

Number 527



**UNIVERSITY OF
CAMBRIDGE**

Computer Laboratory

Architectures for ubiquitous systems

Umar Saif

January 2002

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<http://www.cl.cam.ac.uk/>

© 2002 Umar Saif

This technical report is based on a dissertation submitted by the author for the degree of Doctor of Philosophy to the University of Cambridge.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

<http://www.cl.cam.ac.uk/TechReports/>

Series editor: Markus Kuhn

ISSN 1476-2986

To my parents and Behen ji

Abstract

Advances in digital electronics over the last decade have made computers faster, cheaper and smaller. This coupled with the revolution in communication technology has led to the development of sophisticated networked appliances and handheld devices. “Computers” are no longer boxes sitting on a desk, they are all around us, embedded in every nook and corner of our environment. This increasing complexity in our environment leads to the desire to design a system that could allow this pervasive functionality to disappear in the infrastructure, automatically carrying out everyday tasks of the users.

Such a system would enable devices embedded in the environment to cooperate with one another to make a wide range of new and useful applications possible, not originally conceived by the manufacturer, to achieve greater functionality, flexibility and utility.

The compelling question then becomes “what software needs to be embedded in these devices to enable them to participate in such a ubiquitous system”? This is the question addressed by the dissertation.

Based on the experience with home automation systems, as part of the AutoHAN project, the dissertation presents two compatible but different architectures; one to enable dumb devices to be controlled by the system and the other to enable intelligent devices to control, extend and program the system.

Control commands for dumb devices are managed using an HTTP-based publish/subscribe/notify architecture; devices publish their control commands to the system as XML-typed discrete messages, applications discover and subscribe interest in these events to send and receive control commands from these devices, as typed messages, to control their behavior. The architecture handles mobility and failure of devices by using soft-state, redundant subscriptions and “care-of” nodes. The system is programmed with event scripts that encode automation rules as condition-action bindings. Finally, the use of XML and HTTP allows devices to be controlled by a simple Internet browser.

While the publish/subscribe/notify defines a simple architecture to enable interoperability of limited capability devices, intelligent devices can afford more complexity that can be utilized to support user applications and services to control, manage and program the system. However, the operating system embedded in these devices needs to address the heterogeneity, longevity, mobility and dynamism of the system.

The dissertation presents the architecture of an embedded distributed operating system that lends itself to safe context-driven adaptation. The operating system is instrumented with four artifacts to address the challenges posed by a ubiquitous system. 1) An XML-based directory service captures and notifies the applications and services about changes in the device context, as resources move, fail, leave or join the system, to allow context-driven adaptation. 2) A Java-based mobile agent system allows new software to be injected in the system and moved and replicated with the changing characteristics of the system to define a self-organizing system 3) A subscribe/notify interface allows context-specific extensions to be dynamically added to the operating system to enable it to efficiently interoperate in its current context according to application requirements. 4) Finally, a Dispatcher module serves as the context-aware system call interface for the operating system; when requested to invoke a service, the Dispatcher invokes the resource that best satisfies the requirements given the characteristics of the system.

Definition alone is not sufficient to prove the validity of an architecture. The dissertation therefore describes a prototype implementation of the operating system and presents both a quantitative comparison of its performance with related systems and its qualitative merit by describing new applications made possible by its novel architecture.

Acknowledgments

I would like to thank my supervisor David Greaves whose encouragement and advice enabled me to undertake an ambitious thesis in a new field and finish it in time. I would also like to thank Daniel Gordon who developed most of the AutoHAN architecture with me and was a source of unstinted technical support.

My thesis benefited greatly by the lively discussions in the weekly AutoHAN group meetings. Comments from Alan Blackwell, Peter Robinson, Andrew McNeil, and Gavin Bierman helped me identify many interesting research issues.

I thank all members of Systems Research Group past and present for many helpful discussions, in particular Steven Hand, Austin Donnelly, Dave Stewart, Ian Pratt, Tim Harris, Kier Fraser and James Bulpin. I also have to thank Jean Bacon and other members of Opera Group for many technical discussions, especially about event-driven architectures. I have to thank Pawel Wojciechowski and Peter Sewell for their timely criticism and advice regarding the use of mobile agents in my operating system.

I am grateful to David, Jean and Steve who read my several ambitious papers with extraordinary sympathy, encouraging me and holding me to the very highest standards.

I have to thank Naeem Khan for all his sincere advice and help, and for being a good friend in a foreign country.

Many thanks to system administrators in the Computer Laboratory for providing all the support I needed to prototype and test my operating system.

I owe a special debt of gratitude to my parents and my sister. They have been, more than anyone else, the reason I have been able to get this far. I cannot express in words what their support, love and prayers meant to me. They instilled in me the value of hard work and gave me the confidence to overcome life's disappointments. This difficult and rewarding process would not have been possible without their support.

Contents

<i>Abstract</i>	5
<i>Acknowledgments</i>	6
<i>Contents</i>	7
<i>Figures</i>	13
<i>Tables</i>	15
<i>Chapter 1</i>	17
<i>Introduction</i>	17
1.1 Motivation	17
1.1.1 Scenario	18
1.2 Economic Considerations	19
1.3 Problem Statement	21
1.3.1 Focus.....	21
1.4 Requirements	21
1.4.1 Network Interface	21
1.4.2 Discovery	22
1.4.3 Context-Driven and Application-aware Adaptation	22
1.4.4 Object Mobility and Lifecycle Management	23
1.4.5 Security.....	23
1.4.6 Easy Programmability.....	24
1.5 Challenges	24
1.5.1 Challenge #1: Heterogeneity.....	24
1.5.2 Challenge #2: Longevity.....	24
1.5.3 Challenge #3: Mobility	25
1.5.4 Challenge #4: Dynamism and Context-Awareness.....	25
1.6 Thesis	25
1.6.1 Contribution.....	26
1.7 Case Study	28
1.8 What this thesis is not	28
1.9 Structure of the Dissertation	28

Chapter 2	31
Case Study: Home Area Networks	31
2.1 Overview	31
2.2 Home Automation Industry	32
2.3 Home Area Network Architecture	33
Control architecture.....	34
2.3.1 Physical Media.....	34
2.3.2 Networking Technologies	35
2.3.3 Home Automation Technologies	38
2.3.3.1 Control Interface	39
2.3.3.2 Home Automation Systems	40
2.3.4 Critique of Current Home Automation Systems.....	47
2.4 AutoHAN	47
2.4.1 AutoHan Core Services	48
2.4.1.1 GENA	49
2.4.1.2 Romvets: Resilient Mobility-aware Events	50
2.4.1.3 DHAN: AutoHAN Directory Service	52
2.4.2 AutoHAN Execution Environments and Event Scripts	58
2.4.3 AutoHan Operation.....	59
2.4.4 Internet Access.....	60
2.5 Lessons learnt from AutoHAN	63
2.6 Summary	66
Chapter 3	67
Design Requirements	67
3.1 Overview	68
3.2 Taxonomy of Devices in a Ubiquitous System	68
3.3 Requirements	69
3.3.1 Engineering Requirements.....	70
3.3.2 Requirements posed by Heterogeneity	70
3.3.3 Requirements posed by Longevity.....	71
3.3.4 Requirements posed by Mobility	72
3.3.5 Requirements posed by Dynamism and Context-Awareness	72
3.4 Requirements for Adaptation	74
3.5 Design Goals	75
3.6 Background	76
3.7 Conclusion	78

Chapter 4	81
Related work	81
4.1 Overview	81
4.2 Distributed Operating Systems	81
4.2.1 Amoeba.....	83
4.2.2 Mach.....	85
4.2.3 Plan 9.....	86
4.2.4 Sprite.....	88
4.2.5 Discussion.....	89
4.3 Extensible Operating Systems	90
4.3.1 Dynamically Extensible Operating Systems.....	92
4.3.1.1 SPIN	92
4.3.1.2 VINO	94
4.3.1.3 Fox	96
4.3.1.4 SLIC	97
4.3.1.5 Apertos	98
4.3.1.6 MetaOS	98
4.3.1.7 2K	99
4.3.1.8 Synthetix	100
4.3.1.9 Discussion	100
4.4 Conclusion	102
Chapter 5	105
<i>Context-aware Adaptation in UbiqtOS: A Java-based Embedded Distributed Operating System</i>	105
5.1 Introduction	105
5.2 Contributions made by UbiqtOS	106
5.3 Design Goal	107
5.4 Structure of the rest of the Thesis	108
5.5 System Architecture Overview	108
5.5.1 Layer 0: Extensible Microkernel	110
5.5.2 Layer 1	111
5.5.2.1 SEMAS: Extensible Java Mobile Agent Engine	111
5.5.2.2 UbiqDir	115
5.5.2.3 Romvets	118
5.5.3 Layer 2.....	124
5.5.3.1 Dispatcher	127
5.6 Bootstrap	130
5.7 Summary	133
5.8 Prelude to following Chapters	133

Chapter 6	135
System Components	135
6.1 Motivation	136
6.2 Contributions made by SEMAS	137
6.2.1 Strong vs. Weak Mobility.....	138
6.2.2 Application-specific Connection Management.....	139
6.3 Comparison with Related Work	140
6.4 Structure of the Rest of the Chapter	142
6.5 Mobile Agents	142
6.5.1 Reactive Mobility	148
6.6 Explicit Bindings	148
6.6.1 Context-aware Bindings	151
6.7 SEMAS: Simple, Extensible Mobile Agent System	152
6.7.1 Agent Communication Protocol	155
6.7.2 Extensibility.....	159
6.7.3 Effective Mobility.....	161
6.7.3.1 Extensible Load-balancing	162
6.7.3.2 Application-specific connection management	164
6.7.4 Bootstrap Load-Balancing	167
6.7.5 Disconnected Operation.....	167
6.7.6 Reliability	168
6.8 Summary	171
Chapter 7	173
<i>Extensibility, Dynamism and Context-awareness in UbiqDir: An XML-based Directory Service for Ubiquitous Systems</i>	173
7.1 Motivation	174
7.2 Contributions made by UbiqDir	175
7.3 Ubiquitous Names and Resolution	176
7.4 Design requirements	178
7.4.1 Requirements for Information Model	178
7.4.2 Requirements for Functional Model	178
7.4.3 Requirements for Distributed-operation Model	179
7.4.4 Requirements for Security Model.....	180
7.5 Information Model	181
7.6 Functional Model	184
7.7 Implementation of Romvets	188
7.8 Extensibility in UbiqDir	188

7.9	Distributed-operation model.....	191
7.10	Discovery and Caching.....	191
7.11	Replication and Consistency	193
7.12	Load Balancing and Fault Tolerance: Overlay topologies	195
7.13	Context-aware Adaptation and Production Rules	196
7.14	Security Model	197
7.15	Bootstrap	200
7.16	Comparison with Related Work	200
7.17	Summary	202
	Chapter 8.....	205
	Implementation.....	205
8.1	Background	206
8.2	Implementation of Layer 0	206
8.2.1	Flux OSKit.....	207
8.2.2	Changes made to OSKit.....	207
8.2.3	Support for Dynamic Extensibility	208
8.2.4	Changes Made to OSKit COM model	208
8.2.5	Structure of Layer 0	211
8.3	Implementation of Layer 1	211
8.3.1	Kaffe Port for UbiqtOS.....	212
8.3.1.1	Extensible Scheduling in Kaffe	214
8.3.1.2	Optimizations for Fast Event Handling	215
8.3.1.3	Extensible Protocol Stacks.....	218
8.3.1.4	Connection-oriented Protocols.....	221
8.3.1.5	Adaptation of Protocol Stacks.....	222
8.4	Implementation of UbiqDir	224
8.4.1	Default UbiqDir Extensions.....	224
8.5	Implementation of SEMAS.....	225
8.5.1	Default SEMAS Extensions.....	226
8.6	Default Distributed Services.....	228
8.6.1	Default Dispatcher	228
8.6.2	Default Extensions for Load-Balancing.....	229
8.6.3	Default Extension for Fault-Tolerance	231
8.6.4	Default Extension for High-availability.....	232
8.7	Conclusion.....	232

Chapter 9	235
Evaluation	235
9.1 Evaluation Methodology	235
9.2 Performance Evaluation of UbiqtOS Components	236
9.2.1 Code size.....	236
9.2.2 Cost of extension using UbiqDir.....	238
9.2.3 Cost of Adaptation using Romvets	241
9.2.4 Cost of System Call using Dispatcher.....	242
9.2.5 Inter-component Communication	243
9.3 Evaluation of Context-ware Adaptation in UbiqtOS	244
9.3.1 Follow-me-video Binding.....	244
9.3.2 Flexible Network Support for Mobile devices.....	248
9.4 Conclusion	250
Chapter 10	251
Conclusion and Future Work	251
10.1 Contributions	252
10.1.1 Conceptual Contributions	252
10.1.2 Architectures	253
10.1.2.1 AutoHAN	253
10.1.2.2 UbiqtOS	254
10.2 Future Work	255
10.2.1 Context-specific Protocols and Policies	255
10.2.2 Power-driven Adaptation	256
10.2.3 Security	256
10.2.4 Application Complexity and Backward Compatibility	257
10.2.5 Embedded Device Implementation	257
10.2.6 Active Space Automation Rules	258
10.3 Summary	258
BIBLIOGRAPHY	259

List of Figures

Fig. 1.1 David Tennenhouse (Intel) talk at LCS, MIT, Anniversary talks	20
Fig. 2.1 Expected Home Control Networking Systems Equipment Revenue in the next 3 years.	33
Fig. 2.2 Generic System Architecture of a Home Area Network	34
Fig. 2.3 AutoHAN system Architecture.....	49
Fig. 2.4 An Example Device Description Stored in DHAN	55
Fig. 2.5 Internet Access of a camera in AutoHAN	61
Fig. 3.1 A Typical Middleware Architecture	77
Fig. 3.2 A Typical Micro-kernel Operating System.....	77
Fig. 4.1 Extensible Service Design in the SPIN operating system	94
Fig. 5.1 System Architecture for UbiqtOS.....	110
Fig. 5.2 Context-awareness in UbiqtOS: UbiqDir notifies interested components about changes in the device context using the Romvets Interface.....	118
Fig. 5.3 Installation of new functionality in UbiqtOS using UbiqDir.....	120
Fig. 5.4 Extension of services embedded in UbiqtOS using Romvets.....	120
Fig. 5.5 Romvets Interface of UbiqtOS:	122
Fig. 5.7 An example of Context-aware system-call using the Dispatcher:	129
Fig. 5.8 An example bootstrap sequence in UbiqtOS.	132
Fig. 6.1 Mobet Interface. Components implement this interface to participate as first-class citizens in the system.....	144
Fig. 6.2 Functional Interface Description.....	146
Fig. 6.3 Description of Eventhandler	146
Fig. 6.4 Mbox Interface. Interface Implemented by explicit bindings in UbiqtOS.	149
Fig. 6.5 API for Explicit Bindings. The code fragment shows how an explicit binding is looked-up, parameterized and used to add a string title to a video stream	151
Fig. 6.6 SEMAS Interface. The Interface presented by the UbiqtOS Agent Engine to the System Components (mobile agents).....	153
Fig. 6.7 Bootstrap of an agent in UbiqtOS.....	154
Fig. 6.8 ACP Interface. Communication API in UbiqtOS	156
Fig. 6.9 ACP Authentication Header.	158
Fig. 6.10 Extensibility in SEMAS. Extensions Interpose functionality between SEMAS and ACP	160
Fig. 6.11 Effective mobility in SEMAS:.....	165
Fig. 7.1 An Example Resource Description Registered with UbiqDir	183
Fig. 7.2 This figure shows the algorithm to generate IDs in UbiqDir.....	185
Fig. 7.3 Events offered by UbiqDir to Request Extensible Distributed Operation.....	189
Fig. 7.4 Extensible operation of the directory service.....	190
Fig. 7.5 UbiqDir can support multiple discovery protocols simultaneously, each for a different standard and network interface. These discovery protocols are deployed as extensions to UbiqDir and subscribe to the “looked-up” event, generated whenever a resource description is lookedup in UbiqDir. These extension can then find the	

matching resources in their respective networks and return the descriptions to the mobile agent looking up the resource.	192
Fig. 7.6 An example production rule for events offered by UbiqDir.....	197
Fig. 7.7 Access Control Specification in UbiqDir	198
Fig. 7.8 An Example Group Membership Hierarchy of Principals in UbiqDir	199
Fig. 8.1 OSkit: The COM-based framework and modularized libraries provided by OSKit to implement an operating system.....	207
Fig. 8.2 Structure of OSKit components to implement layer 0 in UbiqtOS prototype. Compare with Figure 8.1 to note that only a minimal set of Oskit components are used. The dotted lines between components signify that the COM indirection has been removed to improve performance. The list based memory manager, FreeBSD packet driver and scheduler were modified to support dynamic extensibility.....	210
Fig. 8.3 Extensible scheduling in UbiqtOS, based on the model proposed in [Harris01]. UbiqtOS implementation uses Romvets to route scheduler activations to extend the model proposed in [Harris01] to provide dynamic extensibility.....	214
Fig. 8.4 An example protocol stack in UbiqtOS.	220
Fig. 8.5 Performance evaluation of the bandwidth of the UbiqtOS TCP/IP protocol stack with traditional architectures. Evaluation was done using two 200 MHz Pentium PCs, connected by a 100 Mbit/sec Ethernet.....	223
Fig. 8.6 TCP one-byte roundtrip time measured with rtpc to compare latency of UbitOS extensible protocol stacks with traditional architectures. Evaluation was done using two 200 MHz Pentium PCs, connected by a 100 Mbit/sec Ethernet	223
Fig. 8.7 UbiqDir default extensions ensure that UbiqtOS devices can interoperate both with IP-based systems and AutoHAN and allow strong consistency for security critical information.	225
Fig. 8.8 Default extensions for SEMAS.....	227
Fig. 8.9 Architecture of standard UbiqtOS distribution.	230
Fig. 9.1 Comparison of ROM Image of UbiqtOS with embedded linux, winCE and QNX operating systems on x86.	238
Fig. 9.2 Comparison of lease traffic for lease update with and without the ID optimization. The traffic generated by ID-based scheme remains fixed, while the traffic generated by transmitting the whole description of the resource to update its description increases with the number of attributes.....	240
Fig. 9.3 Code snippet from Follow-me-Video Application	246
Fig. 9.4 Code snippet from Follow-me-Video Explicit Binding.....	247

List of Tables

<i>Table 2.1 Survey of candidate physical layer technologies for HAN.....</i>	<i>37</i>
<i>Table 2.2 Corresponding MAC layers for physical layer technologies in table 2.1.....</i>	<i>37</i>
<i>Table 2.3 Link layer (and above) for HAN physical layers in Table 2.1</i>	<i>38</i>
<i>Table 3.1 Quantitative comparison of capabilities of embedded devices in a ubiquitous system</i>	<i>69</i>
<i>Table 6.1 ACP frame format. Lingua Franca for inter-UbiqtOS interaction.</i>	<i>156</i>
<i>Table 9.1 Code size of different components in UbiqtOS prototype</i>	<i>237</i>
<i>Table 9.2 Cost of UbiqDir Operations.....</i>	<i>239</i>
<i>Table 9.3 Cost of Romvets operations (in msec) for the un-optimized case</i>	<i>242</i>
<i>Table 9.4 Cost of subscription and notification (in μsec) using the optimizations in Kaffe and Romvets.</i>	<i>242</i>
<i>Table 9.5 Roundtrip RPC in the same address space. Cost measured in cycles on PII 266.</i>	<i>244</i>
<i>Table 9.6 Cost of Adaptation in Follow-me-Video Application.....</i>	<i>248</i>
<i>Table 9.7 Cost of Mobile IP protocol adaptation</i>	<i>250</i>

“The next revolution in computer science is to make computers disappear.”

Anonymous

Chapter 1

Introduction

1.1 Motivation

Advances in digital electronics over the last decade have made computers faster, cheaper and smaller. This coupled with the revolution in communication technology has led to the development and rapid market growth of embedded devices equipped with network interfaces. It has also promoted the development and widespread use of portable computers, allowing users to carry their computation resources and tasks with them.

This presents us with the opportunity to define systems that enable these embedded devices to cooperate with one another to make a wide range of new and useful applications possible, not originally conceived by the manufacturer, to achieve greater functionality, flexibility and utility.

Such a system would allow the computation resources to disappear in the infrastructure to define active spaces [Grimm00][Banavar00][Roman00]; buildings, shopping malls, theatres, rooms, instrumented with embedded devices that collaborate under users' directions to automatically carry out their everyday tasks.

The whole system, therefore, would consist of a multitude of, possibly disconnected, active spaces to provide ubiquitous access to system resources according to the current context of the user.

Such a system promises a future where computation will be freely available everywhere, like batteries and power sockets, or oxygen in the air we breathe. Computation will enter the human world, handling our goals and needs. Devices, either handheld or embedded in the environment, will bring computation to us, no matter where we are or in what circumstances. These devices will personalize themselves in our presence by finding whatever information and software we need. We will not need to type or click, nor to

learn computer jargon. Instead, we will communicate naturally, using speech, vision, and phrases that describe our intent, leaving it to the infrastructure to locate appropriate resources and carry out our intent.

1.1.1 Scenario¹

Karen starts her morning by reading her on-line news service subscription. When her car arrives, she switches to reading the news on her PDA that is equipped with a wide-area wireless connection. In the newspaper, she finds an advertisement for a new wireless equipped camera and calls her friend David to tell him about it.

David's home entertainment system reduces the volume of the currently playing music as his phone rings. Karen begins telling David about the camera, and forwards him a copy of the advertisement that pops up on his home display.

David's decision to buy the camera involves another set of services to be downloaded on his PDA. One acts as a shopping agent to verify that the price is the best possible and another verifies that the camera manufacturer has a dealer in that area. When the purchase of the camera is made, David's bio-metric identification ring authenticates the transaction. His digital camera comes equipped with a short-range radio transceiver as well as a removable cartridge. The data watch he wears can communicate with the camera as well. The two devices informed each other of their protocols and requirements when they were first brought within range of each other. When he snaps a picture, the photo data is communicated to the watch where it is held until a connection can be made to the rest of the world. David's watch acts as a personal storage device and as a gateway to other connections.

Eventually, David enters his home and brings his watch close to his mobile network terminal. At this point, the photo data from the wristwatch can be injected into the wider-area network for the next step of its journey. The pictures find their way through the network to the photo album service. When the camera takes the snapshot it included his encrypted personal ID (from the ring with bio-metric safeguards) in the data packets. Lower-cost data transfer options are always considered automatically based on David's

¹ Adapted from Portolano Project at Washington University

past usage patterns and likely future locations (extracted from his schedule). For example, if David is near his home (and likely headed there), his cellular phone will not place a call to transfer the data, but rather defer the transfer to happen at home where a cheaper Internet connection is available through the home's wireless network and network portal.

1.2 Economic Considerations

A practical system of this nature would inevitably consist of devices and networks manufactured by different vendors, conforming to different, possibly competing, standards. Therefore, at least from an economic point of view, it is natural to ask why different vendors would want their devices to interoperate with one another?

Though this dissertation addresses the technical challenges posed by this heterogeneity, presuming the commercial viability of such a model, this section hints at the feasibility of such a system in the not so distant future.

The computer industry differs from traditional manufacturing in that the costs associated with a given product decline over time due to cheaper implementation. The trend for performance, on the other hand, is relentlessly upward. Hence, the economics of the industry enables manufacturers to periodically introduce enhanced models with richer feature sets at an affordable price.

The result has been a proliferation of devices embedded with appreciable computing power. Fig. 1.1 shows that 98% of the world's processors go into embedded devices, and not desktop computers; this slide was shown as a motivation for DARPA's new research direction in embedded computing, part of the "proactive computing" project.

One natural consequence of the growing complexity of the devices is the desire to connect them together in order to achieve greater functionality, flexibility and utility.

This potential has certainly been realized by the computer industry in the last few years and an increasing number of efforts are underway to enable interoperation of consumer

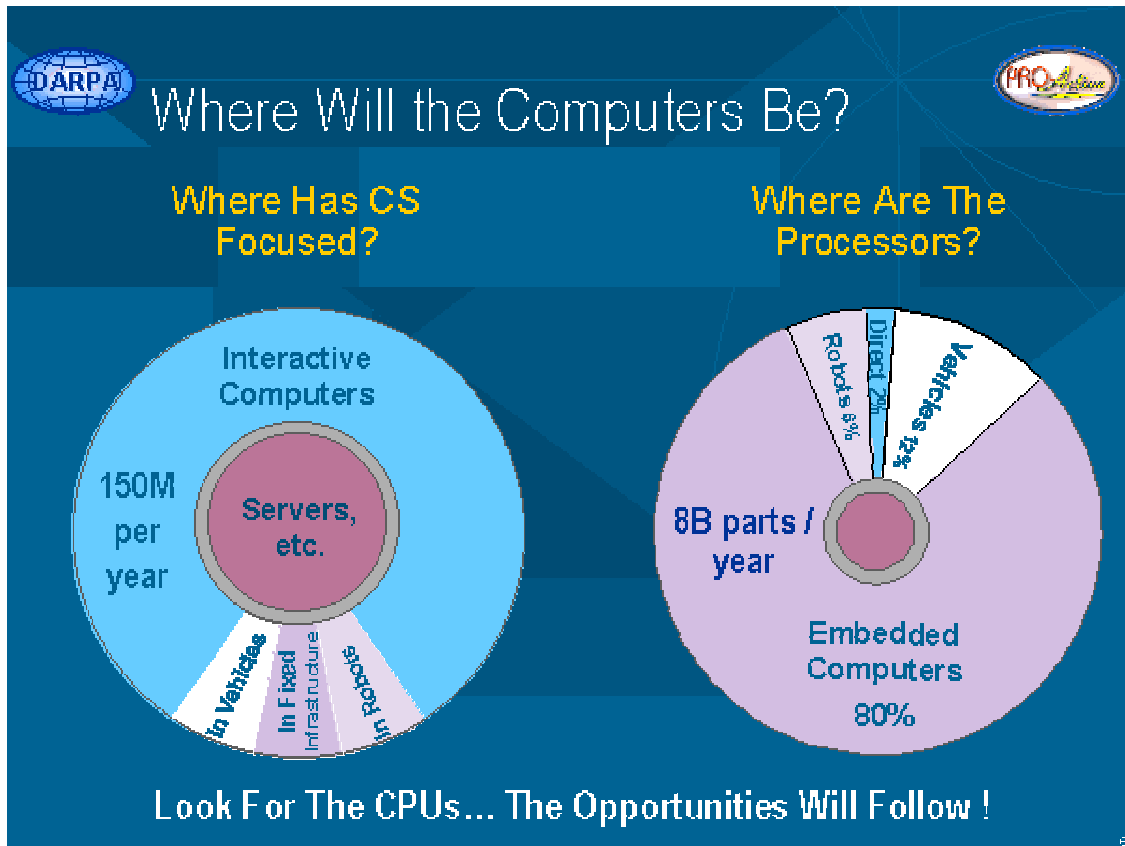


Fig. 1.1 David Tennenhouse (Intel) talk at LCS, MIT, Anniversary talks

devices, especially in the context of home and office automation. In addition to the older systems like X.10 [X10], and CEBus[CEBus], devices conforming to newer systems like Jini [Waldo99], UpnP[UPnP] and OSGi[OSGi] have started to emerge in the market, indicating the industry trend towards enabling a ubiquitous system.

More interestingly, users have also shown keen interest in “programming” their devices as long as they can perform useful functions for them, e.g. the “downloadable mobile phone tunes and games” industry is estimated at 1.30 billion dollars in the UK [Vodafone].

These trends indicate that in future device manufacturers would instrument their devices to enable a ubiquitous system, in order to provide greater flexibility, functionality and utility of their devices.

1.3 Problem Statement

The compelling question then becomes “what software needs to go into these devices to enable such a ubiquitous system”? More specifically, what software needs to be embedded in these devices to allow efficient interoperation with the system? This is the question addressed by this thesis.

1.3.1 Focus

The focus of the thesis is on medium to high-end devices participating in a ubiquitous system e.g. HiFi, Television. Deeply embedded devices like light-bulbs or doorbells are handled by proxies managed by these devices, as proposed by a previous project [Greaves98]. Where an event-based architecture is presented to control, monitor and program deeply embedded devices as part of the AutoHAN project in chapter 2, the main crux of this dissertation is an operating system architecture for medium to high-end devices.

1.4 Requirements

The aim of the software embedded in these devices is to transform these devices from standalone, dedicated pieces of hardware to “universal interactors”, acting as a portal to their resources, to allow other resources to make use of them in a ubiquitous system.

1.4.1 Network Interface

In addition to its physical interface, devices need to support a network interface to allow interaction with other devices on the network.

To enable a ubiquitous system, the software embedded in these devices would need to allow the device to be

- controlled

- monitored and
- programmed

using this network interface.

The control interface allows other resources in the system to change the behavior of the device while the monitoring interface allows other resources to observe and react to changes in the system. Finally, devices that can accommodate extra software need to be able to host novel user applications to program, extend and control the system in ways not originally conceived by device manufacturers.

1.4.2 Discovery

To be able to interoperate with one another, resources first need to be able to find one another on the network.

Resource discovery is imperative, as devices could not know, at time of manufacture, about other devices they would need to interact with to define a ubiquitous system. Instead other devices in the system could only be discovered dynamically at run time. Further, resources frequently fail, join, leave or move in the system and no service can be assumed to be available at all times to be statically linked against. Consequently, applications cannot define static bindings between system resources at compile time, instead useful services can only be composed dynamically using components currently accessible in the system [Esler99].

1.4.3 Context-Driven and Application-aware Adaptation

A ubiquitous system comprises a collection of, possibly disconnected, active spaces each with its own requirements, standards, system idiosyncrasies that define the context of a resource. Resources in a ubiquitous system can range from limited capability embedded devices to high-end servers and can move, fail, leave or join the system. Communication infrastructure can be composed of heterogeneous links [Brewer98] with varying characteristic, prone to network partitions and disconnection. Resources can conform to different standards, and pre-configured support to deploy, locate and manage services may not be available [Winoto99][Esler99].

This heterogeneity, mobility and dynamism coupled with the changing characteristics of wireless links [Brewer98] necessitate adaptation of all those system services that could effect interoperability, efficiency or availability of the system in changing contexts.

The software embedded in the devices needs to be simple enough to be accommodated in impoverished devices, while allowing mechanisms to scale to more privileged devices and to adapt to changing environment conditions as devices fail, join, leave or move in the system, or as the resource is moved from one active space to another.

This dynamism also implies that bindings, established by applications, between system resources cannot be defined statically, instead these bindings need to be able to rebind/re negotiate as available resources fail, leave or move in the system or new resources join the system.

Finally, new applications made possible by the system require new services to be deployed and adapted with the changing characteristics of the system.

1.4.4 Object Mobility and Lifecycle Management

A system can adapt both by altering existing services and by deploying new services in a host to enable it to efficiently interoperate in a new context. This requires support for object mobility and life cycle management of services. Object mobility lets context-specific services be deployed in a device to configure it to interoperate in a new context while life-cycle management provides support for installing, upgrading and removing software from a device. Object mobility is also required to move services around a network for better load balancing, fault tolerance, high-availability.

1.4.5 Security

Resources in such a ubiquitous system would come from different, possibly competing, vendors. This openness of the system means that the level of trust would be low in the system and strong security measures have to be in place to protect devices from one another. Further, components dynamically deployed to configure a device for a specific context can corrupt system state, violate system security or cause denial of service

[Seltzer94]. Therefore, adaptation of software embedded in a device needs to be type safe, to ensure system integrity, and secure, to protect against malicious components.

1.4.6 Easy Programmability

Finally, the software embedded in the devices should leverage easy programmability of applications using the device functionality. It should be easy to control and monitor the device using the embedded software, and program and extend the device functionality to make new applications possible to meet user requirements.

1.5 Challenges

These requirements highlight four fundamental challenges to the design of a ubiquitous system.

1.5.1 Challenge #1: Heterogeneity

As stated above, a ubiquitous system would consist of devices having varying capabilities, connected by networks of different characteristics, conforming to different standards and imposing different requirements on the system. Hence, the software embedded in these devices would need to address this heterogeneity to allow interoperation of resources.

1.5.2 Challenge #2: Longevity

Consumer devices usually have long life times. Therefore, they cannot be manufactured to interoperate with all the newer models of other devices that may become part of the ubiquitous system. Though economies of scale permit increasing computational capability to be embedded in consumer devices to allow additional software to be accommodated, they usually have no programming terminal attached that can be used to upgrade the software embedded in them. This limits their interoperability with newer

devices in the system. Hence, it is desirable to utilize the network interface of a device to automatically upgrade its software to enable it to interoperate with newer devices in the system. Therefore, to enable ubiquitous interaction of devices, the fixed software embedded in the device ROM needs to be minimal that allows suitable extensions to be deployed, using the device network interface, to enable interaction of the device with other newer devices in the system.

1.5.3 Challenge #3: Mobility

The third fundamental challenge is posed by mobility of resources in such a system. Users can carry devices with them from one active space to another or could take them “out of the range” of the system. As different active spaces can have different requirements, standards and system idiosyncrasies, devices might need to be reconfigured as they move between different active spaces. Similarly, varying characteristics of wireless links and the possibility of network partitions warrant adaptation of system services and applications.

1.5.4 Challenge #4: Dynamism and Context-Awareness

Due to the loose structure, mobility and scale of the system, devices in a ubiquitous system join, leave, move and fail more often than in traditional distributed systems. Therefore the resources accessible to a device, which define its context, frequently change in such a system. Hence, every participating resource needs to be able to discover and adapt to its changing contexts to efficiently participate in the system.

1.6 Thesis

The thesis of this dissertation is that these requirements warrant a new bottom-up system design. The challenges posed by the heterogeneity, longevity, mobility, dynamism and context-awareness of a ubiquitous system can only be handled effectively at the operating system level.

1.6.1 Contribution

This dissertation presents the design and implementation of an embedded, extensible, context-aware, distributed operating system. The operating system, referred to as UbiqtOS, enables resources to effectively participate in a ubiquitous system by lending itself to safe, context-driven adaptation and by leveraging context-aware adaptation of applications.

This dissertation makes the following contributions to address the requirements outlined in section 1.4.

- It presents a taxonomy of devices in a ubiquitous system. Although the premise of the dissertation is that embedded devices in future will become increasingly sophisticated and would be equipped with network interfaces, to allow interoperability with other devices, experience with home automation systems showed that devices constituting such a system are either “dumb” e.g. electric kettles, door bells etc. or “intelligent” e.g. hi-fi systems, microwave ovens, handheld computers etc. Dumb devices have limited capability and are only capable of accepting and generating simple control commands, whereas intelligent devices have additional capacity that can be utilized to control, monitor and program other devices in the system as well. Hence, the dissertation presents two compatible but different architectures; one to enable dumb devices to be controlled by the system and the other to enable intelligent devices to control, extend and program the system.
- The dissertation presents a publish/subscribe/notify event-based architecture to manage control-commands for thin devices; devices publish their control commands to the system as typed discrete messages, applications discover and subscribe interest in these events to send and receive control commands from these devices, as typed messages, to control their behavior. This allows applications to automate the system to be structured as sets of intuitive logical assertions embodied as event scripts that subscribe interest in events generated by

devices to trigger condition-action bindings that, in turn, send events to other devices to control their behavior. Finally, the proposed architecture also handles mobility and failure of devices by allowing events to be delivered to “care of” devices in case the device of interest is temporarily disconnected. The “care of” device delivers the event to the destination device if it becomes accessible within a specified time. Moreover, if the mobility pattern of a device can be bound or is known in advance then the event is multicast to all the probable destinations to ensure assured, timely delivery.

- The rest of the dissertation motivates, describes and evaluates the design and implementation of an embedded operating system for medium to high-end embedded devices.
- It shows how the architecture of this extensible operating system lends itself to safe, dynamic and context-driven adaptation to address the challenges posed by a ubiquitous system.
- It describes how the proposed architecture allows components to be injected into the system as mobile agents, extending and adapting the operating system to make new applications possible and to suite the idiosyncrasies of a particular context.
- It proposes a naming framework suitable for describing resources in a ubiquitous system, showing how our architecture provides support to meaningfully capture and indicate changes in context to system services. It describes how these notifications are used to guide a suitable and timely adaptation of the system to suite the changing contexts.
- It introduces effective object mobility and emphasizes explicit, active bindings as a key abstraction to make the system self-organizing for load balancing, fault tolerance and high-availability.

1.7 Case Study

The ubiquity of the system, however, also means that it is difficult to implement and validate a system design without focusing on a concrete example. The dissertation therefore gives a parallel account of the implementation and evaluation of the system on the Cambridge University Computer Laboratory, Home Area Networks Architecture, which has served both as concrete example to guide the system design and as a test-bed to evaluate its performance.

This thesis has benefited greatly from our experience with Home Automation architectures [Saif01]. Work presented in this dissertation is aimed towards the goal of the AutoHAN [Saif01] project to enable a self-configuring home area network, though, as the dissertation illustrates, the system design is relevant for ubiquitous systems in general.

1.8 What this thesis is not

Ubiquitous systems are, indeed, aimed to help users without being intrusive. Therefore, a large portion of research in ubiquitous systems has focused on human computer interaction architectures to allow unobtrusive, natural interaction with the system. This thesis, however, addresses a more fundamental question of what needs to be embedded in these devices that could allow different user interfaces to be supported atop. HCI issues of ubiquitous computing are only discussed when relevant to system design and are, generally, beyond the scope of this thesis.

1.9 Structure of the Dissertation

The remainder of the dissertation is structured as follows.

Chapter 2 describes the problem domain; Home Area Networks. It delineates the problem, presents a survey of some practical home automation systems and presents the evolving architecture of AutoHAN [Saif01]. It concludes with a list of lessons learnt from

the AutoHAN project that serve as the guiding principles for system design presented in the following chapters.

Chapter 3 starts with a taxonomy of devices in a ubiquitous system. It presents the requirements posed by challenges of a ubiquitous system design, and motivates the need for an embedded, extensible, context-aware distributed operating system for medium to high-end devices.

Chapter 4 presents a critique of state of the art in the fields of distributed and extensible operating systems.

Chapter 5 introduces the architecture of UbiqtOS, an embedded, extensible distributed operating system that provides safe, application-specific and context-aware adaptation.

Chapter 6 describes the mobile agent engine that allows context-specific software to be dynamically deployed in UbiqtOS. A binding architecture to address the dynamism of the system is presented. This chapter introduces effective mobility and shows how the mobile agent engine and the binding architecture are used to implement effective mobility and agile bindings.

Chapter 7 describes an extensible, XML-based registry that forms the core of UbiqtOS. A naming scheme befitting a ubiquitous system is presented, along with the philosophy of a context-specific distributed operation of the registry service. The use of the registry service as a meta-interface to extend and adapt UbiqtOS and its role as an orthogonal persistent store is emphasized. This chapter shows how the proposed registry services support context-specific discovery and effectively address the heterogeneity and dynamism of the system better than any other currently available system.

Chapter 8 describes a prototype implementation of the UbiqtOS architecture and chapter 9 presents an evaluation of the UbiqtOS prototype.

Finally, chapter 10 gives a summary of the thesis, outlines the limitations of the architecture and hints at future work to conclude the dissertation.

“To boldly connect what no one has connected before”

AutoHAN Research Group, University of Cambridge

Chapter 2

Case Study: Home Area Networks

Home Area Networks have been a subject of our research for the last five years [Greaves98]. During this time, we have designed, prototyped and deployed several home automation architectures [Greaves98][Saif01]. Our research has spanned from the hardware design level to the issues of user interaction, leading to our current project “AutoHAN” [Saif01]. AutoHAN aims to define a bottom-up design of a self-configuring home area network that allows ubiquitous access to home devices. This thesis is based on the experience with the AutoHAN project, which has served both as a concrete example to motivate and guide the system design and as a test-bed to validate the proposed architecture.

2.1 Overview

The rest of the chapter is organized as follows. Section 2.2 justifies the selection of Home Area Networks as a case study for the thesis, both commercially and technically. Section 2.3 covers different aspects of Home Automation and reviews existing home automation systems. Section 2.4 describes the architecture of AutoHAN. Section 2.5 presents the lessons learnt from the AutoHAN project that led to the requirement to design a new architecture for an embedded operating system. Finally, section 2.5 presents a summary of the chapter.

2.2 Home Automation Industry

Home Automation has received considerable attention from the computer industry in the past couple of years. The home control networking systems market is undergoing a significant transition from closed-loop solutions to open, IP-aware solutions. The result is that the US home automation and controls equipment market is expected to grow from \$1.1 billion in 1999 to \$3 billion in 2005. This is according to the Applied Business Intelligence's (ABI) report on "Home Automation Systems and the IP-Based Control." (fig. 2.1).

The report cited three factors contributing to why the industry may now be ready to begin realizing its true potential. First, the Internet is the leading catalyst to a change in both system designs and business models. IP-aware home control systems not only provide greater value to consumers, but also represent a means for service providers and appliance vendors to create new revenue streams. Utility service providers are particularly interested in the possibility of providing an energy-centric bundle of services, while appliance vendors are looking to market intelligent appliances that have additional functionality and can be managed remotely.

Second, the immense interest in high-speed home networks is spilling over into control-oriented applications and services. Key players are looking to enable a more complete vision of the intelligent home that extends beyond high-speed data and entertainment networks.

Lastly, there has been a renewed effort to develop and to improve technologies for home control applications. New control networking protocols such as the Microsoft-led Simple Control Protocol (SCP) effort and emWare's EMIT architecture promise to enable more reliable, lower cost solutions. Additionally, LonWorks technology is meriting a serious second look as it has gained impressive traction, particularly in the European residential market. These and other home networking technologies are discussed in detail in section 2.4

The heterogeneity, dynamism and context-awareness in a home area network design makes it a good choice to explore design challenges in the wider scope of ubiquitous

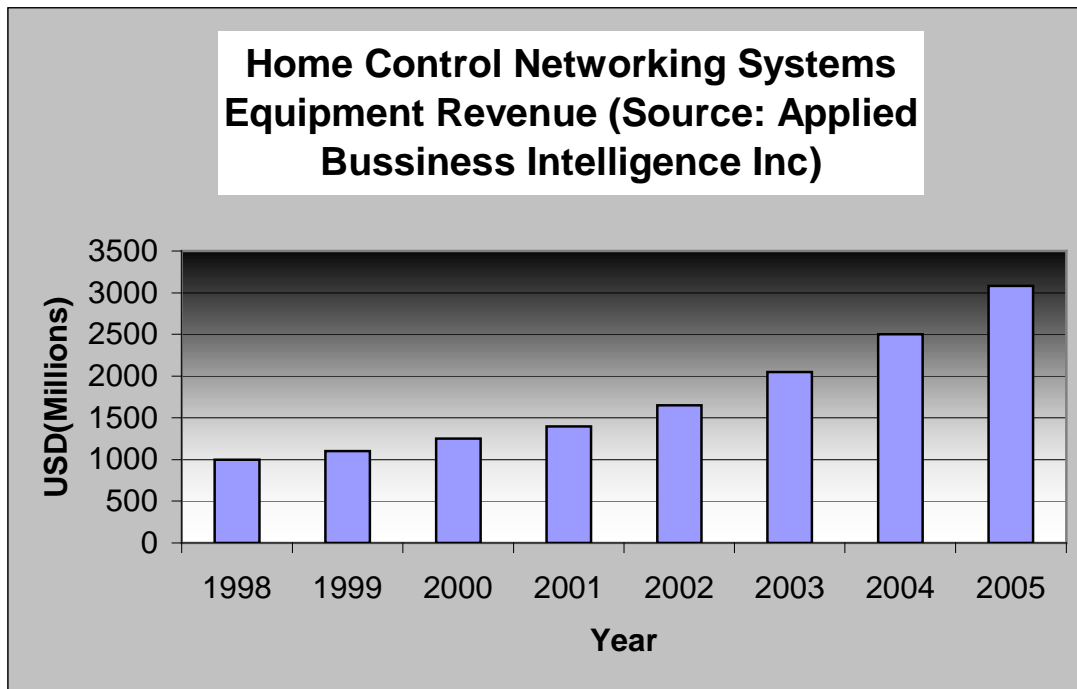


Fig. 2.1 Expected Home Control Networking Systems Equipment Revenue in the next 3 years.

systems. Home Control Networks allow embedded devices in a house to cooperate, under user direction, to assist users in everyday tasks. These devices and the networks connecting them could be both mobile and fixed, would have varying capabilities and could be manufactured by different vendors. Different locations in a house can have different requirements, standards, system idiosyncrasies e.g. garage, living room, garden would each define an active space with different characteristics.

Therefore, home automation networks not only provide a commercially feasible example of a ubiquitous system, they provide a good case study to get a taste for the technical challenges posed by a ubiquitous system design.

2.3 Home Area Network Architecture

Home Area networks, like most computer networks, have an inherent layered structure, with higher layers augmenting and abstracting the functionality of lower layers. The

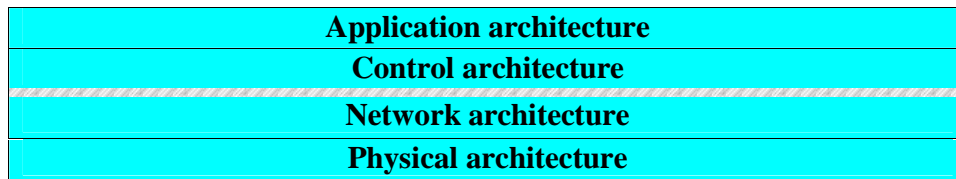


Fig. 2.2 Generic System Architecture of a Home Area Network

conceptual structure of the protocol stack for a generic Home Area Network is as depicted in Fig. 2.2.

This model serves as an architecture which in practice is further refined into sub-layers depending on the specific technologies used in different layers. There will be multiple implementations of some sub-layers: for instance both HomePNA and X.10 might be used at the physical layer.

The reason for this basic architecture is to decouple the technologies of networking from the control and application architectures. The design issues at all levels are different and, in a good design, one technology can be replaced without affecting others.

2.3.1 Physical Media

Home Area Networks are enabled by both wires and wireless media. Wired media can be either newly installed wire, like Coaxial and Plastic Fiber, or existing old wire like POTS (Phone line) wiring and power line. Wireless media can be Infrared and radio (RF).

Wired media is cheap, fast and more reliable than the wireless media. It will be used for fixed and portable devices to provide high bandwidth. Among the wired media Plastic Optical Fiber (POF) may offer the most promising solution but requires new installation in homes. POTS wiring, on the other hand, can be used for bandwidth up to 25 Mbits/sec and does not require new installation. Power Line solutions like X.10, might soon be eliminated from the race of broadband networking, and would at most be used for simple control functions for devices which need a power cable but not high data rate: for example, a standard lamp.

Use of wireless technologies is inevitable in an environment of mobile devices. Both radio and infrared technologies would be part of the network, with radio better suited for multimedia applications and infrared for control channels. Our testbed at the Computer

Laboratory [Greaves98] uses category 5 Unshielded Twisted Pair (UTP), POF and Infrared (IR).

The only conclusion that can be derived from the above analysis is that there is no clear winner, at least in the short run. Therefore a practical network would have to support all kinds of heterogeneous media in its higher layers.

2.3.2 Networking Technologies

Several networking technologies have emerged in the domain of home networking, each suitable for different traffic characteristics and end-host complexity. Tables 2.1, 2.2 and 2.3 compare the characteristics of different home networking technologies.

These technologies differ in the traffic streams supported, network topologies used, addressing schemes defined, and security provided by the.

For example, the specific addressing scheme used depends on a number of factors. A fixed address can only be allocated if the device is complex enough to hold a permanent fixed address, and there are sufficient addresses available, in the address space, to allocate a fixed, unique address to every device. Ethernet and all modern LANs use fixed 48-bit MAC addresses, whereas, a dynamic address can only be allocated either if there is a central service available that could be contacted any time by any device to acquire a dynamic address, or all the devices are ready to cooperate to help each other decide on a unique address, thus increasing the complexity of the network. Finally, topology-based addressing can only be used if some device is ready to act as a root or point of reference e.g. Warren. This device can mediate as a third party for start-of-day connection setup. This technique effectively moves the complexity from a device and to the network.

Similarly, the network topology depends on the traffic requirements and the complexity that the network can afford. Switched schemes, like Warren[Greaves98], require additional complexity in the network whereas broadcast networks like HomePNA [HomePNA] require arbitration protocols to be embedded in end-hosts. Technologies like Bluetooth [Haartsen00] and USB [USB2.0], require a root device in the network to act as a master to manage the network, while technologies like HomePNA and FireWire [Wickelgren97] define a decentralized peer-to-peer network. Further, serial cables, like

FireWire and USB, and short-range wireless technologies like Bluetooth and IrDA [Williams00], have limitations on the length of network links they can support, and hence require bridged topologies to support networks spanning a home area.

Finally, these networking technologies differ in the bandwidth, traffic streams and the levels of quality of service supported by them. Where all the technologies in the home networking domain include support for multiple traffic requirements, they vary in levels of services provided for synchronous, asynchronous and isochronous streams. Technologies like Warren [Greaves98] and FireWire provide support for both asynchronous control commands and multimedia isochronous streams for a range of traffic requirements, whereas simpler technologies like HomePNA and IrDA provide limited bandwidth and support fewer traffic requirements.

Additionally, technologies like Bluetooth and IrDA are more than networking technologies. These technologies support mechanisms for encryption, compression, and authentication, as well as higher-level services like resource discovery and data marshalling.

We believe that all the technologies discussed above would be used in a home area network. HomePNA and HomeRF provide sufficient range to cover a typical home area network, whereas Bluetooth and fast serial buses would be used within individual rooms to define short-range networks. IrDA is finding its use as short-range peripheral interconnect for devices that cannot afford the complexity of Bluetooth. Devices conforming to all these standards had started to come out at the time of writing of this thesis. Apart from the appreciable attention HomeRF and HomePNA have received in the home networking domain, FireWire has been chosen by the VESA Home Network Group as the link layer for their standard [DiGirolam99] and Bluetooth consortium envisages Home Networking to be a sizeable market for their standard.

Technologies	Media	Electric Specs	Device Distance	Bandwidth
MediaWire	Wired CAT-3, 5, Coax	Modulation: Proprietary	CAT 3 : 33 m CAT 5: 100m Coax: 500 m (max 4 km)	100 Mb/sec
Warren	Wired CAT-5, POF Wireless : IR	ATM25 network Encoding: 4b5b		
USB 2.0	Wired 2 STP for Signal 1 UTP for power	Encoding: Differential NRZI	5 m	1.5 (low) ,12(normal) Mbits/sec, 480 Mbits/sec (USB2.0)
FireWire	Wired 2 STP for Signal 1 UTP for power		4.5 m	100,200, 400Mb/sec
HomePNA	Wired POTS CAT-3, CAT5	-Dual Carrier Frequency Network -Frequency Range: 2,32 Mbauds/sec -Modulation: QAM 256	500m (max 2.5 km)	10 Mb/sec
IrDA	Wireless Infrared	-IR range : 0.85 – 0.9 μ m -Optical encoding: RZI, 4PPM -Modulation: SIR, MIR, FIR	0-1 m	SIR: 9.6 – 115.2Kb/sec MIR: 0.576, 1.152 Mb/sec FIR:4.0 Mb/sec
Bluetooth	Wireless Radio	-Frequency Hopping Network: 1600 Hops/sec -Frequency Range: 2.402 GHz ISM Modulation: GFSK	0-10 m	1 Mb/sec
HomeRF	Wireless Radio	-Frequency Hopping Network: 50 Hops/sec -Frequency Range: 2.402 GHz ISM Band -Modulation: 2FSK, 4FSK	0-50m	2 FSK: 0.8 Mb/sec 4 FSK: 1.6 Mb/sec

Table 2.1 Survey of candidate physical layer technologies for HAN

Technologies	Algorithm	Peer-to-peer	QoS	Addressing
MediaWire	Token Ring	No	Yes	Fixed: 64 bit GUID
Warren	Point-to-point: N/A Token tagging	No	Yes	Topology based
USB 1.0	Host (PC) generated Token bus	No	Yes(limited)	Dynamic: 7 bits
FireWire (P1394)	Synchronous bus, frames generated by cycle master	No	Yes(no VBR support)	Dynamic: 64 bits
HomePNA	CSMA/CD with 256 priority levels	Yes	Limited	Fixed: 48 bits
IrDA	NDM: listen for 500 msec, tx when idle NRM: Master/slave (primary station passes token for max 500 msec)	No	N/A	Dynamic: 32 bits
Bluetooth	TDMA (Polling – negotiable - master/slave) ~802.15 TDD provides full duplex comms within each channel	No	Yes	Fixed: 48 bits (piconet ID) Dynamic: 3 bits AMA 8 bit PMA
HomeRF	SWAP (hybrid of ~802.11 i.e. CSMA/CA and ~DECT)	Isochronous: No (DECT) Asynchronous:yes (CSMA/CA)	Optional: depending on mode of operation	Fixed: 48 bits (network ID) Dynamic: 7 bits

Table 2.2 Corresponding MAC layers for physical layer technologies in table 2.1

Technology	Traffic types	Framing	Link Management	Compression	Security
MediaWire	Isochronous, Asynchronous	Fixed sized (per BW) frames generated by Clock Master every 20.83μsec	Token ring (802.5) style management	No	No
Warren	Isochronous, Synchronous, Asynchronous	ATM25 53 bytes cells	ATM management	No	Nouce based Authentication
USB	Isochronous Asynchronous: Bulk transfer (12mb/sec mode), interrupt transfer	No well-defined framing	Command packets from the host	No	No
FireWire	Isochronous, Asynchronous	Fixed sized (per BW) frames generated by Cycle Master every 125μsec	Command packets from the host	No	No
HomePNA	Asynchronous	802.3 frames appended by an extra header and 16 bit CRC, 3 bits of priority	No central management	No	No
IrDA	Asynchronous, Synchronous	Asynchronous: byte stuffing with 16 bit CRC (~UART) Synchronous: ~HDLC, ~SDLC	Command packets from master	No	No
Bluetooth	Isochronous, Asynchronous	Fixed sized frames generated by Primary every 0.625μsec Extended frames supported	Command packets from Primary	LZRW3-A compression	Authentication + Encryption
HomeRF	Isochronous, Asynchronous	Hybrid TDMA/CSMA frame of 20msec	Command packets from CP in DECT mode		128 bit encryption

Table 2.3 Link layer (and above) for HAN physical layers in Table 2.1

This emphasizes the fact that many networking standards would co-exist in a ubiquitous system, both due to technical and commercial reasons. Therefore any ubiquitous system design would need to support heterogeneity of underlying networks.

2.3.3 Home Automation Technologies

The Physical and Network architectures, discussed above, provide the infrastructure for information to be delivered from the source to the destination. It is the control architecture that allows both humans and devices to communicate and control one another's behavior to achieve greater flexibility, functionality and utility.

The control architecture defines the control interfaces and a corresponding control protocol for the devices wishing to interact with one other over the network.

For this to happen every device needs to be able to

- Find other devices on the network, given a set of attributes

- Once found, send them commands to control their behavior.

Devices in such a loosely coupled system often do not know the location of the device being looked-up. Indeed, the location of a mobile resource changes frequently in such a system. Therefore, location-based lookup protocols, e.g. DNS, are not sufficient to discover resources in the system.

Instead, attribute based lookups need to be supported to allow yellow pages style lookup.

2.3.3.1 Control Interface

A device is controlled by its control interface. It could be as simple as a physical binary switch, or as complex as a device driver.

Home devices usually come equipped with a physical interface e.g. buttons and indicators, designed to be controlled by humans. For a device to interact over the network, there needs to be an additional software interface, much like the upper API of a device driver found in an operating system, to expose its functionality to the network.

The control interface is based on the functionality of the device. This functionality of the device is, in turn, a manifestation of its capabilities. The complexity of the real world devices (e.g. HiFi, VCR, security systems etc.) means that attempting to model a whole device as a single entity is not a workable solution due to two reasons:

- 1) The model would be complex, and difficult to manage.
- 2) It would not be reusable and there would be a new model required for even a slightly different device.

In order to make the problem manageable and efficient, home networks need to model the devices at a finer level of granularity, thereby taking a more modular approach. Each module has a corresponding set of well-known functions that could be invoked by any other module. For instance, an answering machine might be modeled as a telephone set and a recording and playback module etc. The advantage is threefold. A) Any device expressed as a collection of these well-known modules could be recognized by other

devices without even having the complete knowledge of the device as a whole. B) The model of each module is smaller and therefore simpler. C) Depending on the level of granularity, there may be scope for reuse of modules across many different devices.

The first point is one of the keys to interoperability between devices which have no prior knowledge of one another.

After defining an interface for a device, which allows it to be manipulated by outside world, the next step is to define a network protocol for sending control commands to the device.

Clearly, it could be as simple as register read/write instructions [Greaves98] to as complex as a Turing complete language [CEBus].

2.3.3.2 Home Automation Systems

The chapter so far has systematically identified the issues involved in the design of a home area network. Using these as the criteria, we now review and discuss different technologies targeted for Home Networking.

X.10

The first of its kind, and still very successful, X.10 [X.10] was designed to provide a simple control network for electrical appliances.

X-10 was designed to use the electrical wiring as the physical medium.

Based on then newly developed Power Line Control (PLC), it used a very simple digital encoding scheme to provide a bit rate of 50bit/s (60 bit/s in US) on the most ubiquitous wired medium in the house. X-10 used an 8-bit addressing scheme, which is implemented by setting DIP switches in the devices.

X.10 was only designed to control simple devices like light bulbs and did not define a device model. It uses a simple control protocol of a set of eight basic commands, compact, but enough for its purpose.

These commands are broadcast on the shared media (power line) and devices respond to the commands addressed to their 8 bit unique identifiers. No registry, transaction or asynchronous events service was defined.

Echelon LONWORKS

LONWORKS is another proprietary standard, based on using an application specific integrated circuit called the Neuron Chip [LONWORKS].

It caters to 2kbit/s to 10kbit/s on dedicated UTP cabling, in addition to 5400 bit/sec bandwidth over power line.

LONWORKS can be viewed as a direct evolution of X.10. Its command set drives the same purpose as X.10 but provides a richer repertoire of 32 commands. The devices respond to the commands addressed to their 15-bit unique identifiers on a broadcast medium (UTP, powerline, Coaxial ect). Like X-10, no support for registry, transactions or asynchronous events has been provided.

Both X-10 and LonWorks were just aimed to provide a network design to support a simple control protocol, and do not concern themselves with other issues in the design of a general-purpose HAN.

European Home Systems

EHS [Kung95] goes a step further in standardizing a network, which can support high-speed continuous data streams alongside its control streams, therefore taking a more unified approach.

EHS caters to a much wider range of media (IR, Power line, UTP, Coaxial, CT2), while still maintaining the main thrust towards power line medium, which operates at 2.4k bit/s. EHS defines a synchronous framed physical layer for UTP and wireless (using CT2 and DECT), in addition to a contention based CSMA style scheme for UTP and Coaxial cable.

Unlike X.10 and LonWorks, EHS introduced the pioneering concept of the device model. An EHS unit is described as a set of entities, each of which has a 16 bit descriptor. The top eight bits identify the entity's general application area (audio, video etc) and the bottom 8 bits identify a particular application (audio player, video recorder etc). The descriptor defines a minimal set of objects and services, which the unit must support. Orthogonal to this descriptor, entities are further classified as Controllers (clients) or

Resources (servers). There are a number of feature controllers present in the network, each responsible for devices belonging to a particular general application area. Simple devices can be handled by intermediate controllers called coordinators. EHS defines mechanisms whereby complex devices are enrolled by feature controllers and simple devices are contracted to coordinators. A resource can be shared by a token passing mechanism.

The control protocol of EHS also used a different approach than X.10 and LonWorks. It defined a command language based on the simple Remote Procedure Call system operating at the application layer.

This simple command protocol language allows resource entities to define so-called programs, a sequence of commands, which can be remotely invoked by a control entity. There are also provisions for transmitting new programs to a resource entity.

CEBus

CEBus [CEBus] is perhaps the most elaborate standardization effort in Home Networking.

The PLC physical layer for CEBus operates at a higher frequency and provides a variable bit rate of up to 7500 bit/s. However, the power-line carrier, like previous networks, does not support high-bandwidth data stream channels.

Other CEBus physical layers include twisted pair, coaxial, optical fibre, wireless radio and IR. Each medium can support data channels in addition to CEBus control channels. CEBus explicitly defines a resource allocation protocol for each medium. CEBus assumes its media to be shared, with different segments linked by link-level frame repeaters, unlike network layer routers used by EHS.

Like EHS, CEBus has its own device model based on an object-oriented approach.

CEBus devices are modeled at three different levels. At the highest level, a device (node in CEBus) is classified by its general type, TV, CD Player etc. At a lower level, the node is modeled as a collection of logical separate subsystems called contexts: a TV is modeled as a channel tuner (video subsystem) and an amplifier (audio subsystem). These contexts are derived from a well-defined context class. The context class defines a

number of basic objects that define a particular subsystem: the audio subsystem in a TV would be modeled as a multistate switch (the audio source selector) and some analogue controls (such as pitch control, audio gain etc.).

What this approach achieves is a certain degree of reusability, e.g. the audio subsystem in TV would essentially have the same subsystems (made of context classes) as the audio subsystem in a HiFi.

The CEBus devices are presented to the outside world by means of metadata. The metadata itself is represented in object form and the CEBus application layer protocol defines a mechanism for retrieving it remotely.

The control protocol in CEBus is called the Common Application Language (CAL) [CEBus]. The CEBus device model is based on an object-oriented approach and therefore CAL is built around the concept of remote method invocation.

The CAL commands are addressed by context number and CEBus unit address, although commands can be broadcast by using wildcards.

CAL is a Turing complete language allowing assignments, conditions, and iterations. the 24 commands in its command set provide rich functionality. In addition, it supports functions like call back, and conditional execution of remote methods that are very useful in monitoring and sensor devices.

Although CAL defines a language syntax for intercommunication between CEBus devices, it does not offer any semantics for its use. Therefore, a high-level scheme called HomePnP [CEBus] has been defined by different vendors. It is aimed to provide semantics for interoperability between devices from different vendors without a prior knowledge of one another.

Compliant devices have to include contexts from a minimal set of interoperability contexts defined by HomePnP. These contexts become automatically bound by broadcasting generic discovery messages by type. Once bound, a loosely coupled system can be formed in which Status objects report information to Listener objects using the typed broadcast messages. Distributed applications can be built using these loosely coupled Subsystems.

Universal Plug and Play (UPnP)

The systems described above define proprietary architectures to allow control and programmability of networked appliances in a home network. To participate in one of these systems, devices need to support the physical media used by the system, the proprietary addressing scheme defined by it, the control protocol used by it and the programming interface required by it. As all of these systems define their own proprietary standards, incompatible with one another, devices conforming to any one standard can only interoperate with a system that conforms to the same proprietary standard from the physical layer to the programming interface exported by it. In particular, these standard are incompatible with existing Internet standards and hence cannot be used to allow remote access to devices in the home network.

Therefore, except from their tiny niche markets, none of these systems have been able to amass any popular support from home appliance vendors.

These shortcomings led to the Universal Plug and Play [UPnP] initiative, which is the most momentous of all the home automation industrial efforts. Led by Microsoft, UPnP consortium includes the majority of players from the home automation market.

UPnP is based on the philosophy of leveraging the existing Internet standard to control and program devices in a home network. UPnP, therefore, uses protocols like IP, DHCP, DNS, HTTP and XML and extends them to suit an ad-hoc environment like HANs.

UPnP uses IP for addressing devices on the network. Devices dynamically acquire IP addresses by either a DHCP service, if one is available in the system, or use an ad-hoc scheme called AutoIP [UPnP]. Devices using AutoIP simply pick an address from a pool of IP addresses and “ping” to check if any other device with that address exists. If there is no reply then that address is reserved by that device. Otherwise the process is repeated with a new IP, until a free address is found.

Another protocol called Multicast DNS defines an extension to DNS in an ad-hoc environment like a home. In this protocol, the device itself keeps its attribute list and responds to any multicast message looking for a device with those attributes. A URL is passed in response to a matched query, where the interface of the device can be reached. Multicast DNS is used as the basis of SSDP described below.

On top of the IP “dialtone”, UPnP rests on three basic protocols, SSDP for device discovery, SOAP and GENA for device interaction. Extensible Markup Language (XML) is used to both describe device functionality and as a wire format for remote interaction. These protocols and their use of XML are described in the rest of this section.

SSDP [UPnP] is the UPnP resource discovery protocol. It uses IP multicast and extends HTTP. Device descriptions are encoded as special HTTP headers and HTTPUDP [UPnP] is used to encapsulate these HTTP headers in UDP packets that are transmitted using IP multicast. Devices export themselves by periodically transmitting their descriptions on the SSDP multicast channel. Devices listen to these advertisements on the well-known multicast channel to find out about other resources on the network. Additionally, devices can send discovery packets on the multicast channel and the resources with matching descriptions respond with their IP addresses.

Device descriptions in SSDP, placed in extended HTTP headers, are encoded as URIs. Although this allows device descriptions to be structured as attribute-value pairs, SSDP lacks support for advanced yellow-pages lookups.

Generic Event Notification Architecture (GENA) [UPnP] enables the UPnP compliant devices to send and receive notifications using HTTP over TCP/IP and multicast UDP.

GENA formats are used in UPnP to create the presence announcements to be sent using Simple Service Discovery Protocol (SSDP) and to provide the ability to signal changes in service state for UPnP eventing. A control point interested in receiving event notifications will subscribe to an event source by sending a request that includes the service of interest, a location to send the events to and a subscription time for the event notification.

Simple Object Access Protocol (SOAP) [UPnP] defines the use of XML and HTTP to execute remote procedure calls. It is becoming the standard for RPC based communication over the Internet. By making use of the Internet’s existing infrastructure, it can work effectively with firewalls and proxies. SOAP can also make use of Secure Sockets Layer (SSL) for security and use HTTP’s connection management facilities, thereby making distributed communication over the Internet as easy as accessing web pages.

Much like a remote procedure call, UPnP uses SOAP to deliver control messages to devices and return results or errors back to control points.

Each UPnP control request is a SOAP message that contains the action to invoke along with a set of parameters. The response is a SOAP message as well and contains the status, return value and any return parameters.

Together GENA and SOAP allow devices to interact with one another to control and monitor one another's behavior. Where SOAP allows synchronous RPC calls to change the behavior of devices, GENA is primarily used for de-coupled interaction by subscribing interest in events happening in the system to be notified when they happen.

Open Services Gateway Initiative Technology

OSGi consortium, led by SUN Microsystems, has defined a residential gateway standard [OSGi] to allow devices in a home network to interoperate and access service outside the house. The OSGi standard is based on the Java Embedded Server technology.

- It defines a collection of APIs that a compliant service gateway needs to support. These APIs include a set of Core and Optional APIs that together define an OSG compliant gateway. Where possible the OSGi is leveraging existing Java standards, such as JINI and JDBC. Where there are standards that apply that are not Java-based, the group's work focuses on integrating with these standards.
- The core APIs address service delivery, dependency and life cycle management, resource management, remote service administration, and device management. All of the core APIs are either contributed by a member or developed by the OSG technical working groups.
- The optional set of APIs define mechanisms for client interaction with the gateway and data management. In addition, several existing Java APIs are included in the optional services. This includes JINI and several other Java standards. A vendor implementing the OSGi specifications is not required to implement all of the optional APIs and their implementation is certified as such.

In addition, if a vendor is implementing the capability defined by an optional standard it is required use the optional standard to implement that capability.

The Open Services Gateway Initiative is still under development as an open standard based on Java technology. Being a gateway technology, the focus of OSGi is on standardizing the APIs and services to enable external access to the home area network, instead of automating a home area network.

2.3.4 Critique of Current Home Automation Systems

UPnP has emerged as the most successful of all the home automation standards described above. UPnP's use of already established open wire protocols and its compatibility with Internet standards has made it most popular with the Industry in a short time.

Among the older proprietary systems, CEBus defines a comprehensive and well-designed architecture by addressing most of the issues involved in home networking. Unlike X.10 and LonWorks it provides support for data channels in addition to low-rate control channels. It also provides specifications for a range of wired and wireless media, instead of just Power Line. Its object oriented device model and CAL based control protocol is better designed and better supported by industry than that of EHS.

Still, all these networks are incompatible proprietary standards, with low-level designs tightly coupled with high-level designs. This means that these products could only be used "as is", with all their shortcomings and weaknesses.

Therefore, the design of AutoHAN [Saif01] is based on a layered architecture that allows interoperation of different low-level technologies and, like UPnP, uses open wire protocols to leverage compatibility with other system.

2.4 AutoHAN

The goal of the AutoHAN project [Saif01] is twofold.

- 1) AutoHAN enables an auto-configuring home area network,
- 2) To allow ubiquitous access to embedded devices participating in the system.

To achieve this, AutoHAN

- Allows interoperability of heterogeneous networking technologies like HomePNA, Bluetooth, ATM etc,
- It defines a set of middleware services and protocols that allow devices to find and interact with one another on the system, while providing compatibility with Internet standards to enable ubiquitous access.
- Finally, it defines a range of execution environments and user interfaces to allow programmability of the system.

The AutoHAN system architecture is shown in figure 2.3.

2.4.1 AutoHan Core Services

As stated above, to be able to control entities on a network, there need to be mechanisms for the resources to

- a) describe their control interface and attributes (non-functional attributes)
- b) advertise these to the rest of the world
- c) find other resources on the network
- d) interact with other resources on the network.

Additionally, there needs to be support to program the system to meet user requirements not originally conceived by device manufacturers.

AutoHAN devices use an XML-based directory service, called DHAN, to find other resources and use Romvets, an extended version of GENA, to control one another. This architecture provides basic compatibility with UPnP compliant devices, while providing richer functionality than a UPnP system. The use of Internet wire protocols and descriptions also allowed us to provide Internet access to AutoHAN [Saif01].

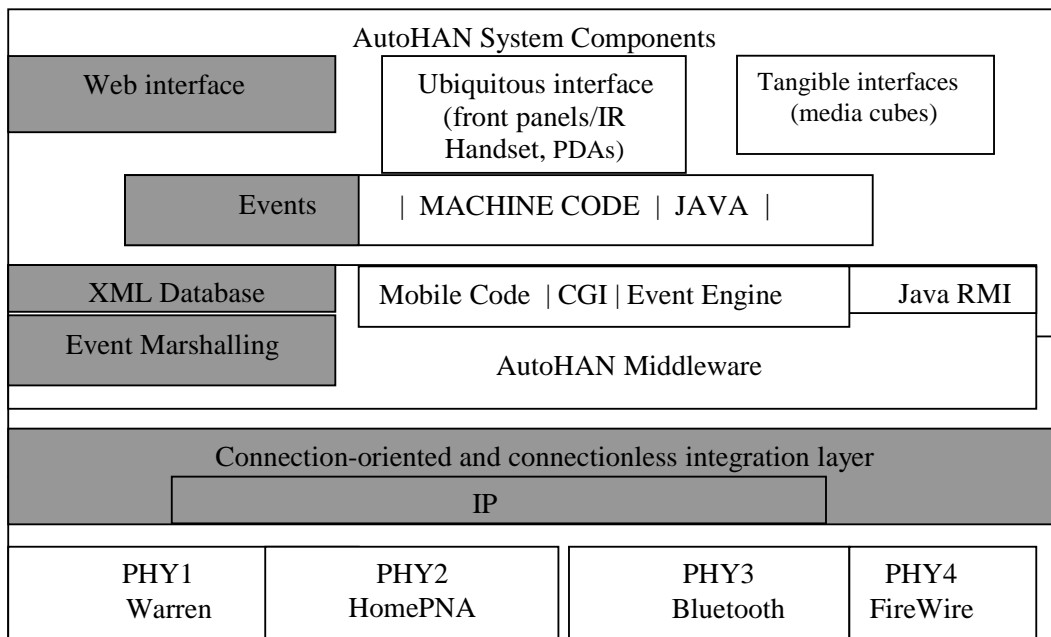


Fig. 2.3 AutoHAN system Architecture

2.4.1.1 GENA

AutoHan devices use an extended version of the UPnP Generic Event Notification Architecture [UPnP] to send and receive events over the network, presenting a unified architecture for monitoring and controlling the device. GENA extends HTTP by the addition of three new methods: SUBSCRIBE, UNSUBSCRIBE and NOTIFY. GENA is an event notification system that runs over HTTP or HTTP-UDP where devices implement a subscription arbiter. Subscription arbiters are used by devices that wish to monitor or control the network, such as the AutoHan event scripts (discussed below). Since the events need to be generated by a wide variety of hardware devices and software entities, not all of which are AutoHan compatible, in our prototype implementation, it is also necessary to have device proxies to convert other forms of event to the extended GENA format.

GENA defines a notification type (NT) and a notification sub-type (NTS), both of which must be URIs. The former is used to specify the type of notification required when subscribing and the latter gives the parameters of the event when it is notified.

The first extension to the GENA architecture in our implementation is the ability to send events to the subscription arbiter. In the standard model, for an entity to receive events, it must actively subscribe to the subscription arbiter and then passively wait for notifications. This does not fit well into the AutoHan model as, for example, an event engine may want to send as well as receive events from the same device, and for simplicity's sake, it would like to use the same subscription. It therefore is desirable to allow the subscriber to generate events and pass them to the subscription arbiter (using the same subscription ID) in the same way as events are received. The alternative would be to have a second subscription arbiter, per device, from which the event engine would offer subscriptions, to which the devices would have to subscribe.

Our implementation of GENA uses HTTP-UDP instead of HTTP, as UDP is sufficient and appreciably more efficient for asynchronous event notifications. Lack of reliable support at the UDP layer is irrelevant, in fact desirable, due to two reasons. First, most of the advanced lower-layers in a HAN are “pretty” reliable e.g. ATM, Bluetooth, even Ethernet (HomePNA), and timeouts and acknowledgements are a big overhead at the transport layer. Second. HTTP (and HTTP-UDP) is itself a request response protocol, and, hence, reliability can easily be supported on top of this mechanism -- a classical example of end-to-end arguments for system design.

2.4.1.2 Romvets: Resilient Mobility-aware Events

Though GENA defines a simple and lightweight events architecture to allow interoperability of thin devices, it fails to address the problems arising from the mobility and dynamism of the system.

GENA does not handle the case when a resource subscribing to an event moves to a different location, disconnects or fails. Romvets architecture address these issues by extending the GENA architecture as follows:-

- GENA subscriptions can only specify a single host to be notified when the event happens. In circumstances where the mobility pattern of the subscriber is known, subscriptions need to be able to specify a list of locations to allow notifications to

be delivered to all the probable locations the subscriber could have moved to. This allows subscribers to be mobile agents with predefined itineraries to carry out their tasks [Wong97]. This technique, usually exploited by QoS reservation protocols for mobile hosts [Talukdar97], is also useful when the mobility space of a mobile node can be bounded or predicted.

- Where the above technique incurs extra overhead of spurious notifications, it alleviates the subscriber from having to change its subscription every time it moves to a new location. However, there are circumstances when the mobility pattern of a subscriber cannot be known, predicted or bound. Hence, there needs to be support to update an existing subscription if the subscriber moves to a new location. This can only be achieved in GENA by first removing an existing subscription and then adding a new one with a different subscriber address. Romvets avoid this inefficiency by introducing a new operation UPDATE that lets a subscriber to change the callback address of an existing subscription. Likewise, Romvets also introduces another method ADD that allows a new callback address to be added to an existing subscription. Address additions are treated as idempotent; similar additions do not cause multiple entries.
- The third addition made by Romvets is to enable GENA to handle subscribers that fail or disconnect temporarily due to a transient malfunction like low battery power or an intermittent network connection. This is handled by allowing the subscriber to specify a “care of” address of a node that is notified along with the subscriber. From the subscription arbiter’s perspective, the care of node is just another member in the subscription address list, except that the “care of” node is notified with a special header “CAREOF: Subscriber address”, which specifies the address of the subscriber. The “care of” node then probes the subscriber and if the subscriber is not accessible then the “care of” node buffers the event. It is the responsibility of the “care of” node to periodically probe the subscriber of the event, from the time it is notified with a CAREOF header for some period t , and deliver the event if the node becomes accessible within that interval. If the subscriber does not become accessible within time t then the event is discarded. This scheme does not require the subscriber to make a separate request to the

“care of” node to take care of its events while it is inaccessible. Instead, the “care of” node implicitly finds about its responsibility when notified with a CAREOF header. This lazy notification also reduces the time to keep the extra state associated with each subscriber in the “care of” node, which takes up its responsibility only when the first event for the subscriber is sent and not when the subscription is made. This allows “care of” nodes to scale, which we expect to be one or two per active space with a wired connection and low probability of failure. The subscriber can, of course, name more than one node in its subscription address list as “care of” addresses to increase the degree of resilience.

These three simple extensions to the basic GENA architecture efficiently address the mobility and dynamism of the system, without introducing undue complexity. This allows thin devices in our system to support romvets while maintaining compatibility with GENA compliant devices.

2.4.1.3 DHAN: AutoHAN Directory Service

DHan is the name of the AutoHan registry service used by different entities to export their resources to the network and to lookup other resources to interact with them. DHan is an XML-based yellow pages directory service. Its **functional model** allows objects of any sort to be registered, unregistered, updated, and looked up by name and/or attribute list. Unlike relational directory services, Dhan’s XML-based **information model** allows objects with varying number and type of attributes to be stored. Each registered object is assigned a lease, which if not renewed within the leased time, automatically deregisters the object from the directory. HTTP1.1 (and HTTP-U) is used as the **Directory Access Protocol**. XLink and MIME External-Body header allows for **Distributed Operation** of the directory service, where lookup operation only returns a link to another instance of the directory service, which might even reside with the object itself. An authentication scheme is used to support a fine-grained access control list based **security model** (see section 2.4).

Why XML? Entities use XML to describe their resources to be registered with the Dhan registry. XML [Bray98] has a number of features that makes it suitable as a meta-language to capture and present the state of a Home environment.

- a) Our experience has shown that most of the entities in a HAN can be grouped in a hierarchical structure. Top-level classes are people, rooms, devices, programs, bank accounts, event scripts, licenses and so on. Within the device class, a potential grouping could be by device functionality or brand name e.g. DisplayDevice/Television/ColorTelevision/SonyColorTelevision. This hierarchical namespace provides a direct mapping to device modeling [CEBus], as well. In this respect XML's hierarchical namespace is a perfect match. However, most of the relationships between different entities in a HAN could be sensibly expressed into alternative tag orderings within hierarchical namespace. Xlink and XPointer technologies cater for exactly that, and their fine-grained linking facilities provided us with an efficient way to describe complex relationships between different entities in a HAN. E.g. Person/Owner/PayPerView/Account could be linked to (otherwise unrelated hierarchy) Event/WatchMovie/PayPerView to pay for a movie a person might have seen.
- b) The competitive and evolving nature of consumer electronics industry leads to product differentiation and a high flux rate of newer models with novel features. Therefore different entities in a HAN have varying number and types of attributes. XML is ideally suited for storing structured but irregular information of this nature, and XML's weak typing and loose matching of tag strings enables interoperation between different generations and versions of devices.
- c) XML is supported by all major browsers and XSL allows multiple views of the same data, allowing devices of varying display capabilities to view the system state.

In AutoHan, an entity can be referenced by a fully-qualified name, called *Point*, somewhat similar to the Distinguished Names (DN) of X.500 [X.500], in its hierarchical

structure. It is a concatenation of all the XML tags, starting from the root of the directory, that identifies the object according to its place in the object hierarchy and a set of attribute/value pairs that delineate the device according to its capabilities. Attributes themselves are XML tags (text nodes), and unlike DN, they can only occur at leaves of a Point and can only assume one value as shown in Figure 2.4.

At times, it is desirable to lookup an entity only by its attributes, whereas the exact position in the device hierarchy would not be of much interest to the client. Therefore the lookup operation permits non-qualified names, as well. A non-qualified name would be composed of only a partial Point name, which could just be an attribute/value set, or even just a single node in the point name in an extreme case, parallel to a Relative DN in X.500. Therefore the above object can be referenced, though not uniquely, by a partial Point name of /ColorCamera and an attribute list of Location/Living Room.

If a partial Point name is used, then the register function itself determines the right place where the object needs to be inserted in the XML tree, by inserting the new element in the same sub-tree as any other already registered element of the same type is registered. A fully qualified Point name, belonging to no already registered hierarchy will, of course, create a new hierarchy. The register function always returns a fully qualified Point name of the object just registered, which can then be used to uniquely refer to the object in future.

Clearly there is no one description schema that can be used to group and represent entities in a HAN. A good description schema would have minimum redundancies and inter-hierarchy relationships for efficient tree search operations. We are currently researching different description schemas for representing the state of the HAN. Having said that, our architecture defines a generic framework that can be used to support any schema standard, which we believe, would emerge with time as the industry agrees on different description standards (e.g. RDF, XRML).

```

An object belonging to the object hierarchy of
    Camera/StillCamera/ColorCamera/

And having the following attributes
    SnapFrequency\10 seconds
    Location\LivingRoom
    Capacity\5 requests
is represented as the Point.
<Camera>
  <StillCamera>
    <ColorCamera>
      <SnapFrequency> 10 seconds
    </SnapFrequency>
      <Location> Living Room </Location>
      <Capacity >5 requests</Capacity>
    </ColorCamera>
  </StillCamera>
</Camera>

```

Fig. 2.4 An Example Device Description Stored in DHAN

DHan itself is a first class resource in the HAN, which means that it registers itself with itself, and interacts with other devices using Romvets events. Hence, it offers events itself corresponding to its access interface, e.g. lease expired, new device registered, device updated. Home control platforms, such as the event engines, will typically register themselves with the subscription arbiter exported by the DHan registry. The registry notifies the event engines of major changes, such as when a new device has been switched on and registered itself, so that event scripts to support these devices can be run. This design decision leverages self-organization of the network.

Suitability of HTTP1.1(-UDP) as the Directory Access Protocol: The next design decision is to specify a directory access protocol to access the contents of the directory service. The home network comprises many resources that may be limited in their capabilities and might come from different manufacturers. This combined with the

requirement of ubiquitous access, made us choose Hyper Text Transfer Protocol (HTTP) [Fielding99] as a directory access protocol.

In addition to being a universally accepted protocol, HTTP-(UDP) has many merits as a directory access protocol for a directory service accessible over the Internet.

- a) HTTP is lightweight, compared to e.g. a full-blown LDAP [Howes97] server, and therefore can be supported in low-end embedded devices.
- b) HTTP works both with connection-oriented and connection-less (HTTP-UDP)[UPnP] underlying protocols, thus satisfying a goal of AutoHan, which is that the protocol can run before and after the network (IP) layer is set up and working.
- c) HTTP1.1 methods fit in rather well with the model adopted by Internet-AutoHAN(IHan) registry service to query and update the system state (discussed below).
- d) HTTP is supported by all web-browsers and thus leverages access to the directory service over the Internet.
- e) It leverages the use of web proxy model to take load off the IHan directory service.
- f) The protocol allows new HTTP methods and MIME headers to be defined, allowing new services like GENA [UPnP] to be supported.

The HTTP GET method is mapped to the lookup function of DHan, it returns the object matching the lookup criterion encoded in the URL, along with its lease in the *Date* header, if it exists and the lease has not already expired. As the Date header contains the time stamp of the time until which the registration is valid, this reply can be cached by a web cache proxy to serve any future requests until the lease expires and the cached reply becomes stale, taking load of DHan. The PUT method is used to register a device with the name encoded in the URL and attributes in the body as XML document fragment. The *Location* header of the response returns the fully-qualified Point name with which the object was registered. The *Date* header gives the lease date until which the registry is valid. The POST method is used to update an existing entry, by changing its attributes

and/or renewing the lease agreement. The DELETE method unregisters an already registered object, and the HEAD method only checks if an object exists. It returns its fully-qualified Point name in the response *Location* header, and the lease time-stamp in the *Date* header for a registered object.

Therefore the five core HTTP methods are mapped to five event types used to interact with the directory services i.e GET, PUT, POST, DELETE, HEAD. The attributes of these events are encoded as XML and MIME headers.

Our design prefers HTTP-UDP to HTTP1.1 inside the HAN multicast network for DHan as well. For Internet access, the IHan adaptation layer provides the conversion between the two.

Advertisement and Discovery: DHAN announces its presence by advertising its IP address every 30 sec on a well-known multicast channel. Devices listen to the multicast channel to be notified about the location of DHAN. Alternatively, devices can send discovery messages on the multicast channel and DHAN replies with its location. In our reference implementation, DHAN runs in a central server i.e. the residential gateway.

Security Model: The security model is based on a fine-grained access control list approach. Every XML-tag in the DHAN directory also contains the access permissions of the group that can access it e.g <ColorCamera Read=Everyone Modify=ColorCamera>. The use of loosely matched hierarchical XML strings as group identifiers leverages a much more flexible security model than e.g a flat bit set, a la UNIX. The group identifiers and membership lists are stored as XML tags, and are managed by the DHAN service, just like all the other state of the HAN. Basically it is just another hierarchy in DHAN, with the root tag <Access Control>.

This group membership directory is initialized by the owner of the house, who can add, modify or delete any entry. All other entities, including people and devices, can add new entries in this database, but can modify and delete only the group membership lists which give them such privileges. By default (when no access control attributes are registered with the tag), only the owner of the house and the entity who had registered the entry can modify or delete it. Therefore even the read permissions to “everyone” have to be given

explicitly; we believe the sensitivity of the information in a HAN warrants this prudent approach. A cascading rule applies to the hierarchical entries which, though, can be explicitly overridden; access control permissions at the root of any sub-tree apply to all the child nodes, but children nodes can override this to have stricter access permissions. This design decision greatly increased the speed of directory access operations, and allowed for shorter access requests.

One issue that has not been mentioned as yet, is how an entity is authenticated? This can, indeed, be done in a number of ways, depending on the execution environment of the DHAN service. For instance, our implementation is done in Java running on Linux. The underlying file system is used by the OS to maintain entity accounts and the entities authenticate themselves by providing an entity ID and password. This identity is passed to the DHAN service for access control.

2.4.2 AutoHAN Execution Environments and Event Scripts

AutoHAN lends itself to programmability by supporting a range of execution environments.

AutoHAN is primarily controlled by event scripts that encode rules to automate the everyday operation of the system. The home automation rules are structured as event-condition-action bindings. These event scripts, written in the Cambridge Event Language [Bacon00], subscribe to the events offered by Romvets event arbiters of different devices and those of DHAN, to be notified about events happening in the system. These events can then trigger actions to automate the home network.

This allows policies like “if someone enters the room (event by the active badge), then switch on the TV (send an event to the TV to switch it on)” to be encoded in a declarative style.

An advantage of using event scripts, which we hope to prove in future work, is that merging of applications which interact can be controlled cleanly and various pathological error cases can be automatically checked using rules of home consistency.

AutoHAN architecture also provides support to adaptively place these event-scripts in the network, using mobile code, for load-balancing, fault-tolerance, high-availability.

The architecture of UbiqtOS, presented in chapter 3, addresses this issue in detail.

2.4.3 AutoHan Operation

Every device that wants to send or receive control events either implements a Romvets arbiter itself or is supported by a proxy that implements a Romvets arbiter.

Any new entity that comes up in the network notifies its event arbiter of the event types it can offer, corresponding to its embedded functionality. The entity then sends a multicast packet on the AutoHan network to locate the DHan service. As the DHan service has already registered itself with itself, like any other entity, it returns its IP address and port address, two of its registered attributes, in reply to this UDP lookup packet. The new entity can then go ahead, authenticate itself, or create a new group, and register its attributes along with the location of its subscription arbiter and the event types it can entertain via that arbiter, with the directory service. The information registered may include location information, but the way in which this information is generated depends on many details, for instance, Warren [Greaves98] devices can tell something about their location from their physical layer address and infra-red exchanges with other nearby devices whose location is already known. The entity may cause the event arbiter to advertise multiple classes of event with various allowable ranges for the parameters associated with an event. Now any entity can lookup this entity by its attributes and, if interested, can subscribe to its events using its event arbiter returned as one of the attributes by the DHan service. These events are handled by embedded event handlers to control the device and/or to setup data channels to and from the device. Because the directory service is itself an entity, it offers a few events itself using its event arbiter e.g. entity registered, lease expired, lease updated, entity deleted etc. The AutoHAN event scripts could subscribe to these events to provide higher order control functions using event scripts. This architecture provides a self-organizing network. Resources can discover one another using XML lookups, and can monitor and control other resources by using event streams, without any human interventions. Use of XML descriptions e.g. instead of strongly typed RMI interfaces, means that any resource can discover and make

use of other resources even if it does not understand all of its functionality (by ignoring some of the XML tags). Conversely, automatic service degradation can be supported by being able to use a resource that supports only a partial functionality expected by another resource. For example, a follow-me-video application can automatically bind itself to a range of display resources varying from LCD displays to monochrome to color TVs depending on what is available in the proximate environment of the user. We have also found this flexibility to be very useful for interoperation of devices from different manufacturers. Soft state in Dhan service, implemented by using time leases, ensures that resources that have failed or moved to a disconnected region are automatically removed from the HAN. The events generated by Dhan can be used to implement different self-organization policies. For example, if the central heating system fails (lease expired), switch off the boiler, as well etc.

2.4.4 Internet Access

The use of established Internet protocols in AutoHAN has also allowed us to extend it to allow Ubiquitous Internet Access to the system, by interposing an adaptation layer between the home network and the Internet.

This layer, called the IHan adaptation layer, extracts AutoHAN XML events from a number of different types of SDUs encapsulated in either HTTP or HTTP-UDP[UPnP] depending on whether the underlying protocol is connection-oriented or connection-less. HTTPUDP events are assigned a unique URI for a request and its corresponding response.

HTTP-UDP is used to receive and send events to the entities connected to the AutoHAN multicast network which only support connectionless communication, whereas HTTP serves TCP/IP Internet connections and other entities supporting connection-oriented communication.

The use of XML and HTTP in the AutoHan design lends itself naturally to web access, except that the current web browsers only implement a selection of HTTP methods and HTML is not sufficiently rich to specify when these functions need to be performed. The current web browsers, of course, do not implement GENA, or Romvets, headers either.

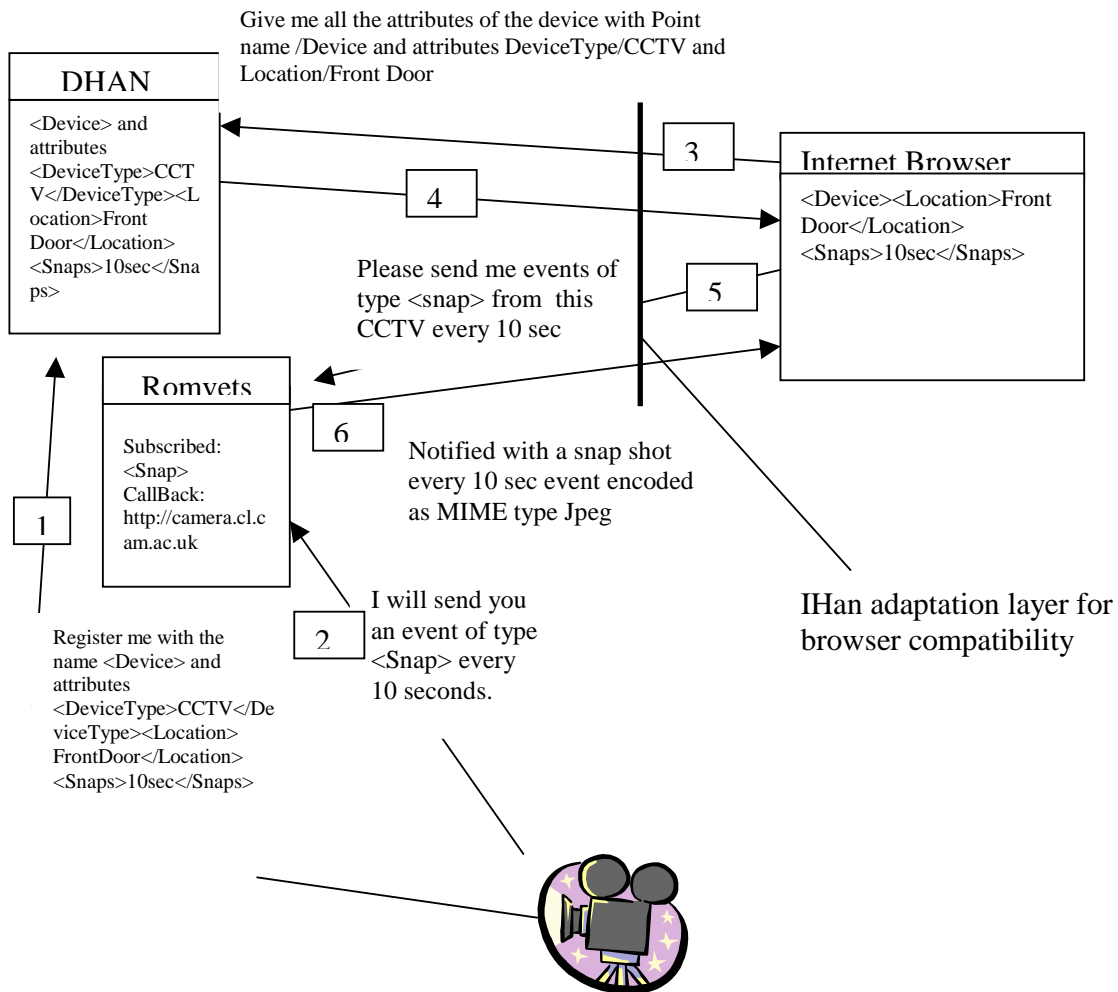


Fig. 2.5 Internet Access of a camera in AutoHAN

To allow a standard Web browser to connect to an AutoHan network, a mechanism is required to encapsulate the HTTP entity header into a URL. The responses from a server likewise need to be encoded in an entity body in XML. A simple scheme is used whereby the HTTP method, path name and some of the MIME headers are included in the path name of a single HTTP GET request. This uses a similar encoding scheme as is used by an HTML form to specify variable names and values to a CGI script.

The GENA draft does not specify the contents of the HTTP body in the NOTIFY messages. In our implementation, the body contains the equivalent XML encoding to the

Notification Type (NT) and Notification Sub-Type (NTS) header fields. This then gives a mechanism for the notification of events to a web browser, in the IHan adaptation layer. As HTTP is a client pull protocol and does not fit in the event subscription and notification paradigm, this is handled by the IHan adaptation module, as well. The IHan adaptation module registers the entity with the attributes which make the DHAN service generate XML pages with the Auto Reload HTML tag set to the frequency of the event notifications. So now event notifications can simply be done by the browser generating an encapsulated GET request to the appropriate event arbiter every n sec, where n is the event notification frequency. Asynchronous events are approximated by very short reload times in the current implementation, but a stock ticker-like java applet is planned for future. The IHan adaptation module provides for tunneling of Romvets events and other HTTP headers as described above, and provides for the event notification using client pull. Some of these IHan pages will be cached by Web cache proxies till the lease date stamp, thus naturally taking the load off the DHAN service for future lookups of the entities with no changes in the registration status.

This scheme makes it possible to control and program the network using nothing more than an Internet browser. Fig. 2.5 shows the interactions that occur as a user instructs the surveillance camera in his house to deliver snapshots to an Internet browser outside the house.

2.4.5 Comparison with Related Work

Though not a home automation standard, Jini [Waldo99] from Sun Microsystems also promises to provide the required framework for home networking. Our work differs from Jini in two important ways. First, AutoHAN uses a language-independent, text-based, XML, directory service which uses an open wire protocol, HTTP, and is therefore not bound to any one API. Second, AutoHAN is tailored for home networking, and its directory service and access protocol provide “just” the functionality for such a network to work. This allows the framework to be supported by highly embedded low-end devices in the home. The choice of already pervasive and agreed upon standards for home

networking allows the infrastructure to scale to the Internet, and this “standards-independent” approach makes this design much more palatable to the industry.

This is exactly what distinguishes our work from HAVi[Lea00] and HomeAPI, as well. HAVi’s design is also limited by its highly FireWire (P1394) centric APIs.

Our work, as mentioned earlier, closely follows the UPnP philosophy of defining modern wire protocols rather than the use of standardized APIs. The use of Romvets provides compatibility with the UPnP design.

AutoHAN provides a more flexible alternative to discovery protocols, e.g. Salutations[Winoto99], SLP[Veizades97] , and, of course, Jini, by allowing loose-matching of XML tags for discovery.

VESA Home networking Committee [DiGiromla98] has proposed a HAN architecture based on XML, as well. In the VESA model, each device has its control interface stored as an “XML page” in it. Whereas in our case, the devices are only required to implement a Romvets interface, and DHAN is used to represent the state of the HAN as XML. This alleviates the devices from costly XML manipulation operations, hence accommodating low-end embedded devices. This approach of pushing the core HAN services in a general purpose high-end node, i.e residential gateway, has also been proposed by OSGi [OSGi], but the OSGi design, like Jini, is highly Java API centric due to its basis in the Java Embedded Server technology.

None of the above mentioned related work has paid much attention to the naming and addressing issues, or provide a detailed fine-grained security model like AutoHan. To our best knowledge, AutoHan is the first architecture to provide a secure framework to control a HAN from across the Internet.

2.5 Lessons learnt from AutoHAN

Our experience with the AutoHAN project taught us a number of lessons and highlighted new requirements for system design.

- Warren and AutoHAN primarily focused on the embedded system design for limited capability devices. Warren defined an extremely simple register read-write ATM cell level protocol, whereas AutoHAN requires participating resources to support nothing more than an event loop and a simple directory access protocol to lookup and interact with other devices in the network.

Over the course of the project, there was a growing realization that most of the devices in our system could support appreciably more complex software and an overly simplistic system design did not make effective use of system resources. Instead of just being able to allow their resources to be controlled by the system, more privileged devices can also support user applications and system services to control the active space. AutoHAN architecture includes support for object mobility to, primarily, export event scripts to the devices, but this can be used to deploy user applications and system services as well. This issue is discussed in detail in the following chapters.

- A home network comprises a collection of active spaces, each built to provide a different utility, e.g. living room, kitchen, lawn etc. Therefore, each of these places can have different requirements and system idiosyncrasies. Further, as mentioned earlier, a home network would inevitably consist of a number of low-level networking standards. This means that no single, fixed software embedded in participating devices can leverage their efficient interoperability. Instead, the software embedded in these device would need to adapt as the device is moved from one active space to another or as new resources join, or existing resources move or leave the active space, changing the characteristics of the surrounding system. Likewise, user applications and external system services residing with the device would need to adapt as the surrounding conditions change.
- Another important lesson learnt from AutoHAN was to avoid single point of failures in the system. In an environment where devices frequently fail, move or leave the system, no single device can be guaranteed to act as a highly available server. A design like Warren, where all the devices are controlled by proxies running in a centralized controller, though presents a feasible design for impoverished devices,

introduces a single point of failure. Moreover, physically separating the state of the device from the proxy managing it introduces issues of maintaining consistency.

- The initial design of DHAN was based on hard-state, but it was soon obvious to us that soft-state is imperative for such a dynamic system to leverage fate-sharing [Clark88]. Where soft-state in DHAN, implemented using time leases, introduced some extra overhead incurred by periodic lease renewal requests, it effectively removed inconsistent state from the system as devices failed or disconnected without warning.
- DHAN lookup service in the reference implementation of AutoHAN is served from a central server. Where this made the implementation simpler by avoiding problems of maintaining consistency that would have arisen by replicating instances of DHAN, it highlighted a novel problem for resource discovery. Some of the attributes associated with a device are only meaningful when computed relative to the device looking up the resource (client) e.g. the latency to access a looked-up resource. If the directory service is physically separated from the client, then these attributes exported by a resource would, implicitly, be relative to the location of the directory service instead of the resource looking-up the resource. Hence, a discovery model where the directory service resides in a physically separated host limits the usefulness of the looked-up name.
- Another novel requirement highlighted by the use of DHAN was dynamism in resource descriptions. Some of the attributes of a resource in an active space change with time, e.g. current load on the resource, latency to access a mobile host. Hence, these attributes are only meaningful when computed on the fly. DHAN's use of soft-state addressed this requirement indirectly as lease refresh time puts an upper bound on the staleness of the view. Still, where smaller lease interval could limit the staleness of the view, it exacerbates the overhead of the lease refresh traffic.
- Finally, we found that asynchronous publish/subscribe/notify events paradigm is well suited to monitor and control such a system. The use of subscribe/notify events architecture leverages loose-coupling of the system, allows multiple parties to be notified about an event happening in the system, can provide support for mobility, handle disconnection and node failures and naturally lends federation of

heterogeneous systems. Cambridge Events Architecture [Bacon00] also provides support for filtering, aggregation, and federation for conditional notification of events. Suitability of events paradigm is presented in the next chapter [Saif01b].

2.6 Summary

This chapter described the problem domain of the thesis i.e. home area network automation. The suitability of home automation as a practical, feasible and representative example of a ubiquitous system is emphasized. A survey of current home networking architectures is presented to motivate the need for a generic and flexible system design.

This chapter presents a detailed description of the AutoHAN architecture and highlights the merit of a simple, layered and flexible design to both provide interoperability of heterogeneous devices and to leverage ubiquitous access to the system. The chapter concludes with a list of lessons learnt from the AutoHAN project that serve as guiding principles for the system design presented in the following chapters.

*“Bringing abundant computation and communication,
as pervasive and free as air, naturally into people's lives”*

Oxygen Research Project Goal [MIT]

Chapter 3

Design Requirements

The previous chapter substantiated the vision of a ubiquitous system by giving a practical example of one application domain: Home Automation. It verified the claim that future consumer devices would be instrumented to interoperate with other resources in the system, instead of being standalone pieces of equipment providing dedicated functionality.

It discussed the challenges arising from the longevity, heterogeneity and mobility of devices in the system and highlighted the proliferation of networking and high-level standards in such a system, each suitable for different requirements and system characteristics.

Finally, it presented the design of AutoHAN to enable interoperability of heterogeneous devices in a home. The AutoHAN design highlighted the suitability of XML to describe the irregular, still structured, description of devices in the system. Further, it presented a simple publish/subscribe/notify architecture to program, control and monitor the system.

Where the AutoHAN event architecture described in chapter 2 catered for limited capability devices, its simplistic design did not make effective use of more privileged resources participating in a ubiquitous system. End devices in AutoHAN only need to support a simple event-loop that handles Romvets encoded events. These events handlers, embedded in the device, are used to control and monitor the device by event scripts, written in CEL [Bacon00], hosted by more privileged devices in an active space.

However, chapter 2 only described architectures to discover, monitor and control the functionality embedded in a device, using events transported by open wire protocols, but did not show how the system can be programmed to meet new user requirements.

This chapter, on the other hand, motivates an architecture that allows device functionality to be augmented with new software, like event scripts, to make novel applications possible, not originally conceived by device manufacturers. For example, in the scenario presented in chapter 1, this architecture would allow David's PDA to be equipped with new software to purchase a camera for him, would allow his newly acquired camera to be configured with additional software to provide interoperability with his wrist-watch, and would allow the pictures from his camera to be injected in the network to be displayed by the networked photo album service.

3.1 Overview

This chapter investigates the requirements for a substrate that can be embedded in medium to high-end devices to allow new functionality to be introduced in the device to program the system with novel applications. It elaborates upon the requirement to allow context-driven adaptation of the software embedded in these devices to address the heterogeneity, longevity, mobility and dynamism of the system.

3.2 Taxonomy of Devices in a Ubiquitous System

The first two chapters of the dissertation have referred to devices comprising a ubiquitous system either as limited capability devices or as privileged devices. Where this gives an adequate indication of the device functionality, there needs to be a precise categorization of resources in the system on which to base the assumptions for system design.

Distributed systems have traditionally modeled resources as servers or clients depending on whether the resource is providing a service or using it. Lately, clients without any

Device category	ROM	Flash	RAM	Disk	MIPS
Thin	< 32K	None	< 8 K	None	< 1
Fat					
Medium	< 300K	<5MB	<5MB	None	< 50
High-end	> 300K	N/A	>5MB	None	> 50

Table 3.1 Quantitative comparison of capabilities of embedded devices in a ubiquitous system

persistent storage, and with limited capabilities, have been described as “thin clients” [Schmidt99]. Our experience with AutoHAN has made us coin the term “thin server” to describe a limited capability device which offers a service corresponding to its embedded functionality, using an event-loop, but lacks support to accommodate additional software. Where thin clients and thin servers, e.g. electric kettle, doorbell, can only send and receive simple control commands corresponding to their embedded functionality, fat devices, e.g. PDA, TV, are capable of supporting additional software to program and control other resources in the system.

Table 3.1 presents a quantitative comparison of the typical capabilities of different devices that would make up a ubiquitous system.

3.3 Requirements

While Romvets define a simple architecture to enable interoperability of thin devices, fat devices can afford more complexity that can be utilized to support user applications and system services to manage the system.

For this to happen, devices need to provide a portable execution platform to execute and manage additional software as well as the capability to manage their embedded hardware resources. Hence, every fat device needs to support an embedded operating system that manages and exports its embedded hardware to the system and lends its additional computing resources to be used by additional software to program and control an active space.

3.3.1 Engineering Requirements

Traditionally, consumer devices have fixed software in their ROM to control the embedded functionality of the device. Lately, devices conforming to emerging standards to allow interoperability of devices, like UPnP [UPnP] and Jini [Waldo98], have additional software embedded in the device ROM that exports the embedded resources of the device to the network and allow them to be controlled over the network using well-known protocols and APIs.

However, to allow additional software to be installed and executed in the device, the device also needs to be embedded with mutable storage, like RAM and Flash, to store the dynamic state required to manage and execute the additional software.

Devices with RAM and Flash memory have already started to emerge in the market.

3.3.2 Requirements posed by Heterogeneity

Chapter 2 highlighted the heterogeneity at three levels in a ubiquitous system.

1. Devices in a ubiquitous system can range from limited capability embedded devices to high-end servers. Therefore, software designed to be embedded in these wide range of devices needs to be simple enough to be accommodated even in limited capability fat devices (medium) while providing mechanisms to scale to more privileged devices (high-end).
2. As mentioned earlier, a ubiquitous system comprises a collection of, possibly disconnected, active spaces. As these active spaces are built to provide different utilities, each has its own requirements, standards, system idiosyncrasies. Hence, the software embedded in a device, to enable it to interoperate anywhere in the system, needs to be able to adapt according to the requirements of the active space of the device.
3. Chapter 2 highlighted the proliferation of physical media and networking technologies for home networks and discussed their suitability for different requirements. It concluded with the note that at least a few of these technologies would co-exist in the near future. Therefore, the communication infrastructure in a

ubiquitous system can be composed of heterogeneous links [Brewer98] with varying characteristic, prone to network partitions and disconnections. Hence, the software embedded in a device to enable ubiquitous interaction in such a system needs to provide flexible communication support to cater to the diverse characteristics of the low-level networking infrastructure.

In addition to this heterogeneity in the infrastructure, resources (both hardware devices and software services) participating in a ubiquitous system are also of several types, each with its own set of interfaces to allow control over its behavior. This poses the following requirements.

- Software objects representing these resources need to describe themselves in a format that allows other objects in the system to discover and use their functionality.
- Further, in order to allow them to be discovered and used by applications in a type-safe programming environment, these objects need to be derived from some well-known generic object type that is any object can be assigned to and introspected for any special behavior.

3.3.3 Requirements posed by Longevity

Consumer devices have long life times. Hence, the software embedded in these devices needs to be able to interoperate with newer models of other devices in the system that would emerge over its lifetime. As the novel features of these new models, and their corresponding interfaces, cannot be known to the software embedded in a device at manufacture time, the software needs to be able to dynamically discover new resources joining the system and allow itself to be upgraded to interact with newer models. This, in turn, also requires that devices describe their features and interfaces in a future-proof way, to allow older devices to recognize and interoperate with them.

3.3.4 Requirements posed by Mobility

1. Mobility of the devices means that they can be moved between different active spaces with disparate characteristics. Therefore, the software embedded in the device needs to be able to dynamically adapt to efficiently interoperate in changing environments.
2. Indeed, the single most important factor towards the vision of a ubiquitous system is the rapid emergence and deployment of untethered, wireless access. As the characteristics of a wireless link are prone to the changes in the physical environment, flexible communication support is required to address the varying characteristics of a wireless link [Brewer98]. Moreover, different wireless technologies could be bridged together in an overlay to provide ubiquitous connectivity [Brewer98], performing vertical handoffs as a device moves from one network to another. Therefore, protocol stacks embedded in the software need to dynamically reconfigure to adapt to the changing characteristics of the communication links as it moves within the same network or between different networks.

3.3.5 Requirements posed by Dynamism and Context-Awareness

The mobility of devices introduces dynamism in the system as devices move, join or leave the system. The loose structure of the system also means that the system design cannot be based on the availability of any particular device to provide a critical service. In fact, no pre-configured support to deploy, locate and manage services may be available [Winoto99], especially when active spaces are spontaneously formed due to the proximity of devices, like in wireless ad-hoc networks e.g Bluetooth [Haartsen00]. The set of resources around a device and the characteristics of the surrounding system that define the context of the device change dynamically.

1. Therefore, in addition to be able to manage and export the resources of the device to the system, the substrate embedded in the devices needs to be able to discover, capture and indicate the device context to applications (that program the system) and

services (that control and manage the system e.g. load-balancing service). Changes in the device context need to be conveyed to system services to trigger adaptation. Context indication needs to be instantaneous and expressive to guide a timely, suitable adaptation.

2. Once notified about changes in resource context, the system could adapt both by altering existing services and by deploying new services suitable for the new context. For instance, discovery protocol could be changed from SLP [Veizades97] to SSDP [Czerwinski99] or a new protocol layer like IPv6 can be deployed to enable efficient interoperation with the new context. This requires support for object mobility and life-cycle management of services. Object mobility lets context-specific services be deployed in a visiting resource while life-cycle management provides support for installing, upgrading and removing services from a device.
3. Further, the changing structure of the system with devices moving, failing, leaving or joining active spaces means that the services and applications to control and manage an active space cannot be configured statically. Instead, they need to be dynamically moved and replicated in the system to provide for load-balancing, fault-tolerance and high-availability in the changing system resources.
4. Dynamically deployed context-specific software, however, can pose threats to system integrity and security. Dynamically deployed components can corrupt system state, violate system security or cause denial of service [Seltzer94]. Especially in a ubiquitous system where components might come from different, possibly competing, sources, security becomes a primary concern. Therefore, system adaptation needs to be safe, to ensure system integrity, and secure, to protect against malicious components.
5. Similarly, context-specific applications, belonging to different sources need to be
 - Accounted for the resources they use and
 - Protected against each other from denial of service (DoS) attacks.
6. Finally, the dynamism of the system implies that bindings between different components in the system cannot be defined statically. Instead, useful services can only be composed dynamically using components currently accessible in the system [Esler99]. Hence, dynamic service composition needs to be supported atop resource discovery. Moreover, highly available services with long life times would need to

reconfigure as constituent components move, fail or leave the system or better component choices become accessible. Consequently, component bindings need to be flexible to allow application specific rebinding when appropriate.

3.4 Requirements for Adaptation

Therefore, the operating system embedded in a device to enable ubiquitous interoperability needs to allow all the software that could effect interoperability, efficiency, or availability of the system to be deployed and adapted dynamically to address the heterogeneity, longevity, mobility, dynamism and context-awareness of the system.

Adaptation, however, takes different forms according to the specific requirements.

- 1) The fixed part of the operating system needs to be simple enough to be embedded in the ROM of a low-end fat device, like a microwave, while adaptation is required to extend this minimal substrate to suit the complexity of more resourceful devices.
- 2) Where the above point can be addressed using a toolkit to construct a family of operating systems [Beuche99], tailored to the complexity a device can afford at manufacture time, this static adaptation is not sufficient to address the dynamism and context-awareness of the system. Context-awareness and dynamism of the system requires these extensions to be deployed and adapted dynamically according to the requirements of the current active space of the device.
- 3) Dynamic adaptation is also warranted to tailor the operating system to suit the requirements of the applications and system services placed with the device to utilize its additional resources.
- 4) Dynamic adaptation is also required as the device is moved from one active space to another, changing the set of resources and system characteristics in the context of the device.

- 5) Finally, dynamic adaptation is required to tailor the policies of the operating system as the utilization of resources embedded in the device changes, such as
- a. CPU cycles i.e. load
 - b. Network link Bandwidth
 - c. Network link latency
 - d. Memory utilization
 - e. Area on display screen
 - f. Battery life for handheld devices

3.5 Design Goals

Therefore an operating system to enable interoperability of fat devices in a ubiquitous environment needs to

- Manage resources embedded in the device
- Export these embedded resources to the system
- Discover the surrounding resources and
- Dynamically adapt according to the characteristics of the surrounding system and applications resident with it. The adaptation requires
 - New software to be installed
 - Existing software to be extended, upgraded and replaced and
 - Dynamically redistributed among the changing set of system resources to allow for load-balancing, fault-tolerance, high-availability.

Adaptation in this distributed operating system needs to be

- Dynamic
- Application-aware
- Context-driven
- Efficient and

- Secure

Finally, the kernel needs to be minimal to be accommodated even in low-end fat devices, while allowing it to be extended to scale to more privileged devices.

3.6 Background

Although the flexibility and modularity in micro-kernel [Liedtke95] and, more recently, middleware systems [Bernstein96] lend them to adaptation, they fall short of addressing the requirements posed by a ubiquitous system design.

The term Middleware refers to the software layer interposed between the application and the operating system to facilitate distributed programming (figure 3.1). Middleware systems define a broker service that abstracts away the problems of distribution and heterogeneity of lower layers from the application developer. Their modular structure has been exploited by recent research on reflective [Roman99] and adaptive [Blair98] middleware designs to allow application-aware [Roman99], and even, context-aware [Blair98] customization of middleware services. However, as middleware systems are layered on top of the operating system, which itself could be of a fixed monolithic design, this adaptation is limited to only a few services in the user space. Hence, functions like network communication, scheduling, caching, monitoring, usually embedded inside the operating system kernel, cannot be adapted by a middleware. Second, in order to present a unified distributed computing environment, the middleware need to duplicate some of the operating system functionality if there is a mismatch in the policies implemented by the operating system and those promised by the middleware to the applications. Further, as the middleware is opaque to the operating system, the mismatch in functionality can lead to an inefficient system design [Aniruddha96]. In an embedded environment, where resources are limited, this duplication and inefficiency is most undesirable.

This makes adaptable middleware systems like 2K [Kon00], and [Blair98] insufficient to address the requirements posed by a ubiquitous system.

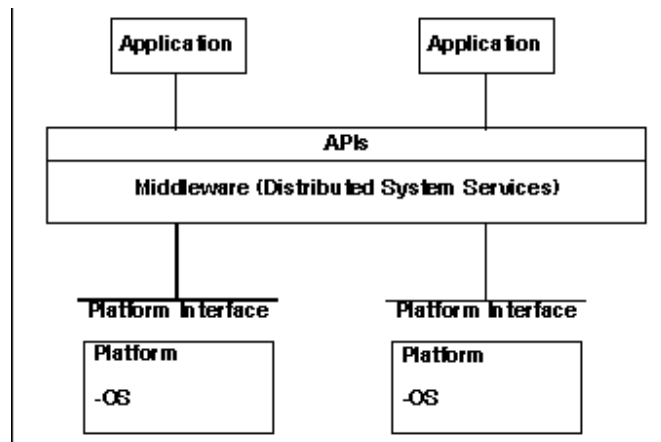


Fig. 3.1 A Typical Middleware Architecture

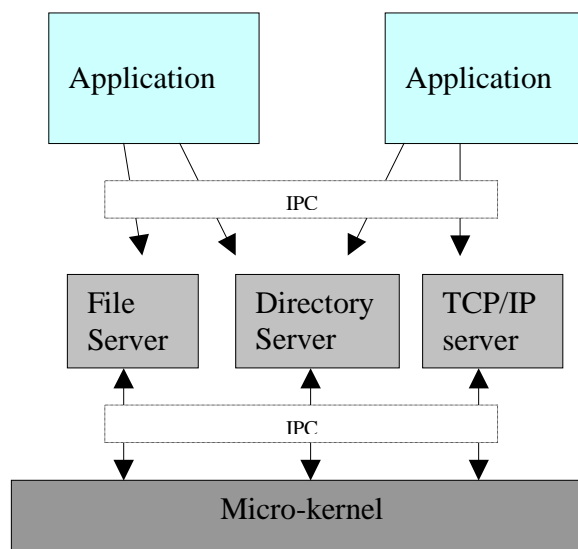


Fig. 3.2 A Typical Micro-kernel Operating System

A well-designed micro-kernel, on the other hand, allows operating system policies to be adapted to avoid the above-mentioned problems, but their strictly layered design introduces considerable IPC overhead (fig. 3.2) [Hsieh93]. Further, micro-kernel adaptation is, primarily, either at build-time [Bricker91], or at link-time [Leslie97], whereas the dynamism of a ubiquitous system warrants run-time adaptation. Although some newer micro-kernel architectures like QNX [Hildebrand92] and modules in Linux [Beck96] allow some of the operating system services to be replaced at run-time, the adaptation is ad-hoc (as described in chapter 4) and does not permit adaptation of core operating system services.

These two shortcomings are addressed by dynamically extensible operating systems [Bershad94].

A dynamically extensible operating system allows software to be linked inside the kernel to avoid the IPC overhead and provides facilities to replace these extensions at run-time [Bershad94]. However, extensible operating systems have only been researched in the past to tailor the operating system policies to suit application requirements [Bershad94]. Whereas, the heterogeneity and dynamism of a ubiquitous system necessitate context-specific adaptation in a resource-constrained environment, as well, to address the requirements outlined in this chapter.

3.7 Conclusion

This chapter argued that the heterogeneity, longevity, mobility, dynamism and context-awareness of a ubiquitous system warrant adaptation of the substrate embedded in fat devices to enable ubiquitous interaction. Further, this adaptation needs to be dynamic, application-aware and context-driven. We argue that the flexibility offered by traditional micro-kernel designs and middleware designs is not sufficient to efficiently meet these requirements. Instead, dynamically extensible micro-kernels [Bershad94] provide a more flexible, efficient and integrated approach to the adaptation warranted by a ubiquitous system. However, dynamically extensible kernels have only been researched for application-specific adaptation of operating system policies, whereas an operating system

for fat devices in a ubiquitous system needs to support context-driven adaptation in a resource constrained environment.

Therefore, we propose an embedded, context-aware, extensible, distributed operating system for fat devices in a ubiquitous system.

“History is philosophy teaching by example”

Dionysius

Chapter 4

Related work

Chapter 3 motivated the need for a substrate that can be embedded in fat devices to transform them from standalone dedicated pieces of hardware to “universal interactors”, acting as a portal to their resources, to extend, control and program the system to make novel applications possible. It derived the requirements for this substrate design from the heterogeneity, longevity, mobility, dynamism and context-awareness of a ubiquitous system and highlighted the need for context-driven extension and adaptation of the embedded substrate to address these requirements. Finally, chapter 3 hinted at the suitability of extensible micro-kernels to enable fat devices to effectively participate in a ubiquitous system.

4.1 Overview

This chapter surveys relevant research in distributed operating systems and extensible micro-kernels. It covers the salient features of pertinent systems and evaluates their suitability against the requirements presented in chapter 3.

This chapter concludes with an analysis of the shortcomings of current systems to motivate the need for a new bottom-up design of an extensible operating system to address the requirements for fat embedded devices in a ubiquitous system.

4.2 Distributed Operating Systems

An operating system designed to participate in a ubiquitous system would serve to enable, control and coordinate resources distributed in a physical space and present them as programmable “universal interactors”, acting as a distributed operating system for the active space [Roman00].

Unlike a time-sharing networked operating system like UNIX where applications are exposed to the details of any operation over the network, making it difficult to program a distributed system, a distributed operating system provides *transparent* distributed operation. However, different distributed operating systems differ in the manner they abstract away the details of distributed operation and the level of transparency and flexibility they provide to the applications. In particular distributed operating systems provide varying degree of support for providing

- Location transparency: clients do not have to know where resources are located to use them
- Migration: resources can move at will and still be referenced by the same name
- Concurrency: multiple clients can share resources seamlessly
- Replication: extra copies especially of heavily used resources can be made without clients knowing.
- Parallelism: the system software will take advantage of the multiple processing units and run jobs in parallel without clients having to program this explicitly.

Below we give an overview of four distributed operating systems, chosen because of their different approaches to providing transparency and flexibility in a distributed environment. We describe how their primitives for process management, naming, communication and resource management allow them to achieve some or all of the goals listed above.

We conclude this section with an evaluation of these systems against the requirements set out in the last chapter.

4.2.1 Amoeba

The amoeba operating system [Tanenbaum90] design caters to the environment where it is affordable and possible to provide 10 or even 100 CPUs per user, an assumption that was based on the ever-declining prices of processors in early 1990s.

A typical Amoeba system consists of four types of machines. First, each user has a workstation for running the user interface. Second, there exists a pool of processors that are dynamically allocated to users as required. Third, there are specialized servers, such as file servers and directory servers that run all the time. Finally, there are gateway machines that allow multiple Amoeba systems that are far apart to be connected together in such a way that to the user, the whole appears to be a single integrated system, rather than a collection of different systems. All these components must be connected by a fast LAN.

Amoeba's model of transparency is based on the concept of processor pools. A processor pool consists of a substantial numbers of CPUs, each with its own local memory and network connection. Pool processors are not owned by any one user. When a user types a command, the operating system dynamically chooses one or more processors on which to run the command. When the command completes, the processes are terminated and the resources held go back to the pool, waiting for the next command, very likely from a different user. Therefore, processes are transparently spawned on a remote host, the least loaded one in the processor pool, to perform their task and terminated when finished.

Amoeba's later versions implement strong mobility to perform load-balancing, allowing a running process to move from an overloaded host to an idle one. However, the development team concluded that additional complexity introduced to support migration offset the flexibility leveraged by it [Tanenbaum90].

Amoeba was designed as a *microkernel* architecture. The kernel supports the basic process, communication, and object primitives. It also handles raw device I/O and memory management. Everything else is built on top of these fundamentals, usually by user space *server* processes. Some of these are user processes, running application programs. Such processes are called *clients*. Others are *server* processes, such as the file

server, that provides read and write access to files, the directory server, that maps files to file handles, and the SOAP server that provides a heart-beat service to ensure that crashed servers can be detected and rebooted to provide fault-tolerance.

The basic Amoeba communication mechanism is the *remote procedure call* (RPC). Communication consists of a client thread sending a message to a server thread, then blocking until the server thread sends back a return message, at which time the client is unblocked. Amoeba has a special language called *Amoeba Interface Language* (AIL) for automatically generating stub routines to marshal parameters and hide the details of the communication from the users.

For many applications, one-to-many communication is needed, in which a single sender wants to send a message to multiple receivers. Amoeba provides a basic facility for reliable, totally-ordered group communication, in which all receivers are guaranteed to get all group messages in exactly the same order. This abstraction simplifies many distributed and parallel programming problems.

Just as there are two unifying concepts in the kernel, threads and communication, there are also two unifying concepts in the user-level software: objects and capabilities.

When an object is created, the server doing the creation constructs a 128-bit value called a *capability* and returns it to the caller. Subsequent operations on the object require the user to send its capability to the server to both specify the object and prove the user has permission to manipulate the object. Capabilities are protected cryptographically to prevent tampering. All objects in the entire system are named and protected using this one simple, transparent scheme.

Objects are looked up in the Amoeba directory server, which returns a capability when presented with an ASCII name of the object. These capabilities will be for files, directories, and other objects. Since directories may contain capabilities for other directories, hierarchical file systems can be easily built.

The directory service exports a unified view of the object namespace, irrespective of their location. Having acquired a capability for an object, applications can send RPC messages to it, without being aware of its location in the system.

The standard Amoeba file server has been designed for high performance and is called the *bullet server*. It stores files contiguously on disk, and caches whole files contiguously in core. Except for very large files, when a user program needs a file, it will request that the bullet server send it the entire file in a single RPC. The bullet server does not provide naming services. It just reads and writes files, specified by capabilities. A directory server maps ASCII strings onto capabilities.

A directory entry may contain either a single capability or a set of capabilities, to allow a file name to map onto a set of replicated files. When the user looks up a name in a directory, the entire set of capabilities is returned, to provide high availability. These replicas may be on different file servers, potentially far apart (the directory server has no idea about what kind of objects it has capabilities for or where they are located). Finally, operations are provided for managing replicated files in a consistent way.

4.2.2 Mach

Though, primarily designed for single and multiprocessor systems (SMPs), the modular design of the Mach micro-kernel [Accetta86] naturally supports distributed computing, as well. The most important contribution made by Mach was a flexible and open design, providing only a minimal basis on which other kernels can be emulated. This emulation is done by a software layer that runs outside the kernel. Each emulator consists of a part that is present in its application programs' address space, as well as one or more servers that run independently from the application programs. Multiple emulators can be running simultaneously, so it is possible to run 4.3 BSD, System V, and MS-DOS programs on the same machine at the same time.

Like all other microkernels, the Mach kernel provides process management, memory management, communication, and I/O services. Filesystem, directories and other traditional operating system functions are handled in user space. The idea behind the Mach kernel is to provide the necessary mechanisms for making the system work, but leaving the policy to user-level processors.

A concept that is unique to Mach is the memory object, a data structure that can be mapped into a process' address space. Memory objects occupy one or more pages and

form the basis of the Mach virtual memory system. When a process attempts to reference a memory object that is not presently in the physical main memory, it gets a page fault. As in all operating systems, the kernel catches the page fault. However, unlike other systems, the Mach kernel can send a message to a user-level server to fetch the missing page.

Interprocess communication in Mach is based on message passing. To receive a message, a user process asks the kernel to create a kind of protected mailbox, called a port, for it. The port is stored inside the kernel, and has the ability to queue an ordered list of messages. A process can give the ability to send (or receive from) one of its ports to another process. This permission takes the form of a capability, and includes not only a pointer to the port, but also a list of rights that the other process has with respect to the port (e.g. SEND rights). Once this permission has been granted, the other process can send messages to the port, which the first process can then read. All communication in Mach uses this mechanism. Every Mach process gets a few system ports that are used by the kernel to communicate with the process, including a process port, a bootstrap port and an exception port. This unified message passing I/O system allows different operating system emulators to be supported, without requiring changes to any interface exported by the micro-kernel. Further, a special interface compiler, Mach Interface Compiler, generates corresponding service interfaces for libraries to relieve the applications from having to handle the complexity of message passing interface.

Initially the Mach micro-kernel supported a network message server, which implemented the TCP/IP communication stack in user space to send messages to non-local processes. However, supporting network communication in user space proved unacceptably inefficient and the later release pushed this component back inside the kernel.

.

4.2.3 Plan 9

Plan 9 [Pike90] started as an attempt to build a system that was centrally administered and cost-effective using cheap computers as its computing elements.

The philosophy is much like that of the Cambridge Distributed System. The motto of the project was to build a UNIX out of a lot of little systems, not a system out of a lot of little UNIXes.

The transparent view of the system is built upon three principles. First, all resources are named and accessed like files in a hierarchical file system. Second, there is a unified, single protocol, called 9P, for accessing these resources. Third, the disjoint hierarchies provided by different services are joined together into a single private hierarchical file name space.

A typical Plan 9 installation has a number of computers networked together, each providing a particular class of service. Shared multiprocessor servers provide computing cycles; other large machines offer file storage.

User terminals all provide access to the resources of the network. When someone uses the system, though, the terminal is temporarily personalized to that user by customizing one's view of the system provided by the software. That customization is accomplished by giving local, personal names for the publicly visible resources in the network. The services available in the network all export file hierarchies. Those important to the user are gathered together into a custom name space; those of no immediate interest are ignored. This is a novel use from the idea of a 'uniform global name space' [Pike92]. In Plan 9, there are known names for services and uniform names for files exported by those services, but the view is entirely local. This could be explained by considering the difference between the phrase 'my office' and the precise address of the speaker's office. The latter may be used by anyone but the former is easier to say and makes sense when spoken. It also changes meaning depending on who says it, yet that does not cause confusion. Similarly, in Plan 9 the name `/dev/cons` always refers to the user's terminal and `/bin/lpr` the correct version of the print command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing `/date`. Plan 9, then, has local name spaces that obey globally understood conventions; it is the conventions that guarantee sane behavior in the presence of local names.

The 9P protocol is structured as a set of transactions that send a request from a client to a (local or remote) server and return the result. 9P controls file systems, not just files: it includes procedures to resolve file names and traverse the name hierarchy of the file

system provided by the server. On the other hand, the client's name space is held by the client system alone, not on or with the server, a distinction from systems such as Sprite [Ousterhout88].

This approach was designed for traditional files, but can be extended to other resources. Plan 9 services that export file hierarchies include I/O devices, backup services, the window system, network interfaces, and many others. Files come with agreed-upon rules for protection, naming, and access both local and remote, allowing services built this way to be ready-made for a distributed system.

4.2.4 Sprite

Sprite [Ousterhout88], much like the amoeba operating system, was designed to provide a single, transparent view of a distributed system. However, sprite is based on a model to transparently utilize the idle machines belonging to other users on the network, instead of using a model of a global pool of resources not owned by anyone, like in amoeba. As a consequence, Sprite also includes elaborate support for process migration, primarily used to transparently evict a foreign process from a machine to his "home machine" when its owner reclaims the machine while the process is still in execution. However, the developers report the support for process migration difficult to implement, fragile to changes in the system, and the cost of migrations to be too expensive for the majority of the tasks [Douglass91].

The Sprite kernel includes support for multithreading, remote communication using RPC, and distributed shared memory to allow easy programmability of distributed resources. Further, though plan 9, and amoeba abstract system resources using a unified abstraction of a file and provide location transparent namespace for accessing files, files are static objects in these operating systems. Sprite operating system, on the other hand, provides both location and migration transparency for files; the kernel allows files to be dynamically moved from one host to another, transparently to the application using the file. Sprite manages the file system with a single hierarchy, a tree structure, which is copied for all the domains. Each kernel has a private prefix-table that maintains and manages the tree structure of the file system. Files, including I/O devices represented as

files, are accessed in the same manner whether they are local or remote. And the notion of Name Transparency allows files in Sprite to be moved from one machine to another without changing their names. Proc_Migrate, a Sprite system call that will move processes from one machine to another machine, is also transparent to programmers. Finally, Sprite employs large variable-sized caches on both client and server to leverage high-performance.

4.2.5 Discussion

The motivation for this thesis is comparable to the motivation for amoeba. Where amoeba was based on the early 90s assumption that several computers would be available to execute a user's tasks, this thesis is motivated by the ubiquitous presence of embedded processors that can be utilized to support additional tasks in an active space. However, where amoeba was designed for stipulated, stable computer systems, the design of an embedded operating system to enable a ubiquitous system needs to address the heterogeneity, longevity, mobility, dynamism and context-awareness in the system.

All three micro-kernels described above, Ameoba, Mach and Sprite, are based on flexible, modular designs, making it easier to maintain and alter the system. However, any modification to these operating systems requires a system reboot at best and a recompilation of software at worst. Similarly, these operating systems provide a fixed set of well-known services and new services, even as user space processes cannot be introduced in a straightforward way. Finally, the policies and protocols supported by these operating systems are fixed, and cannot be changed independently of the mechanisms used to implement them e.g. load-balancing in amoeba.

Further, where sprite and later versions of amoeba provide support for transparently migrating running processes at arbitrary points, developers from both teams suggest that supporting this sort of strong migration involves too much work specific to the underlying platform and the cost of transparent process migration, including all of its related state, is usually too high to be useful for most cases.

Plan-9 abstracts away the heterogeneity of resources in a distributed system by using a unified file abstraction, accessible from a global namespace, much like a UNIX system.

However, it is unique in allowing configurable views of that global space to be exported to different resources in the system. This makes distributed programming simpler by allowing appropriate resources to be easily accessed and allows system policies like security to be imposed by just restricting the view of the global space visible to different resources.

However, the use of files to represent all the resources in a system is overly restrictive and enforces a singular API that is not sufficient for all purposes. Resources in these systems, represented as files, are discovered by presenting a well-known directory service with the unique ASCII name of the file representing it. Though Plan 9 allows that name to be interpreted according to the location of the application, to allow non-qualified names to be used by applications, it fails to provide a general-purpose attribute based discovery of resources required in a ubiquitous system.

Finally, the design of Mach and, to a lesser extent, amoeba provides a unified abstraction for communication, using message passing via protected mailboxes i.e. ports. Where this provides a unified communication paradigm, ports once assigned for the communication have fixed properties and cannot be adapted to suit the changing requirement of underlying network e.g. in wireless environments.

However, none of these systems allow run-time adaptation or extension of services embedded in the operating system to address the longevity, mobility, dynamism and context-awareness of the system.

Next chapter shows how some of the concepts presented in this chapter are adapted to the design of a distributed operating system for a ubiquitous system.

4.3 Extensible Operating Systems

Although the modular design of the micro-kernel operating systems described above provides clean fault isolation, improves performance and maintainability, these systems do not lend themselves well to adaptation and extension.

This shortcoming has been addressed in extensible systems. Extensible kernels allow operating system policies to be adapted to address the special requirements of certain applications [Bershad95] that cannot be efficiently met by general-purpose operating systems.

Extensible operating systems can be divided into three categories, depending on when they can be extended in their lifecycle.

- Build time extensible systems can only accommodate new kernel services while they are being built. Once the system is booted, no more changes can be made unless they are stopped, rebuilt and rebooted.

Recent operating systems in this category employ object oriented technology, structuring operating system functionality as a set of objects that can be sub-classed to specialize operating system policies. Well known systems include Choices [Campbell93], and its micro-kernel version μ Choices [Francisco99], Chorus Classix [Guillemont 97] which is an object oriented version of Chorus OS [Rozier88] and Scout [Hartman94], a communication oriented operating system that allows communication paths to be explicitly declared and scheduled to meet the communication requirements of different applications.

- Link time extensible systems (also known as library operating systems) define only a very low-level functionality in the kernel. This functionality simply virtualizes the hardware in a thin layer of software based protection making it safe to expose to the user level. Conventional operating system functions are then implemented by user level libraries that link into this minimal substrate. By selecting an appropriate library, applications can adapt the system to suit their requirements. However, once a library operating system has been linked to by an application, the system cannot be adapted further.

Well-known link time extensible system include exokernel [Engler95] and its corresponding library operating systems like Aegis, exOS, XOK, PhOS each designed for different application requirements, Cache-kernel [Cheriton94], and Nemesis [Leslie97].

- However, none of the above mentioned systems allow adaptation of operating system at run-time. Run-time extensible systems allow components inside the operating system to be replaced while the system is running. This makes them the closest fit to the requirements outlined in chapter 3. Below we describe the design of the extensible systems that had an impact on the design of our system presented in the following chapters.

4.3.1 Dynamically Extensible Operating Systems

Extensibility in dynamically extensible systems can be divided in two categories, ad-hoc and reflective. Ad-hoc designs are typically functionally oriented and are characterized by the use of different design methods at different points of extension within the kernel. Additionally, there are no clear guidelines as to what a service should look like (i.e. they lack a service design framework). Reflective designs are object oriented and use the principles of introspection and reflection, to standardize the way in which the system extensions are designed and inserted in the kernel. Reflection allows the reification of the operating system internals and the subsequent adaptation of those internals to a specified need. This is achieved through the definition of a Meta Object Protocol (MOP) which provides a well-defined interface to query and modify the reified objects inside the kernel.

Allowing user defined code inside the privileged kernel address space clearly presents integrity issues. Some of the projects described below implement schemes to address this security threat while others simply require those application-specific extensions be trusted.

4.3.1.1 SPIN

SPIN [Berhsad95], from the University of Washington is the most well-know dynamically extensible operating system.

In SPIN, application defined system services are decomposed (shown in fig 4.1) as follows: -

- SPIN Dynamically Linked Extensions (SPINDLEs), that are dynamically downloaded into the kernel address space,
- Libraries, that exist in the same address space as the application and may execute autonomously or contact kernel SPINDLEs via systems calls, and
- User level servers that maintain long-lived state about the extensible service i.e. state that outlives the thread of application that instantiated the service.

Applications dynamically augment the default set of SPINDLEs when they introduce new services. The role of a SPINDLE is to define policies over the resource usage abstractions implemented by the resource controllers part of the fixed kernel substrate. Once a SPINDLE has been inserted inside the kernel, its installing application can benefit from reduced latency in responding to kernel events, as compared to user level servers, and fine-grained hardware access capabilities enjoyed by kernel level code.

All SPINDLEs are attached to event lists and are invoked when the corresponding event occurs. Interest in these events is registered as call-backs and SPINDLEs are upcalled to notify them of a service request. However, these events are not asynchronous as in the conventional definition of events [Ma98], instead the event handling routines block and return a value once finished executing. The kernel includes, what it calls, a dispatcher module that allows SPINDLEs to subscribe to the events offered by the kernel substrate and routes the events to the corresponding SPINDLEs when they happen. Extensions can download event guards inside the kernel to filter unwanted notifications. The SPIN operating system includes a set of core operating systems services, such as virtual memory, networking, scheduling, and storage management, that have interfaces amenable to an event-based extension model.

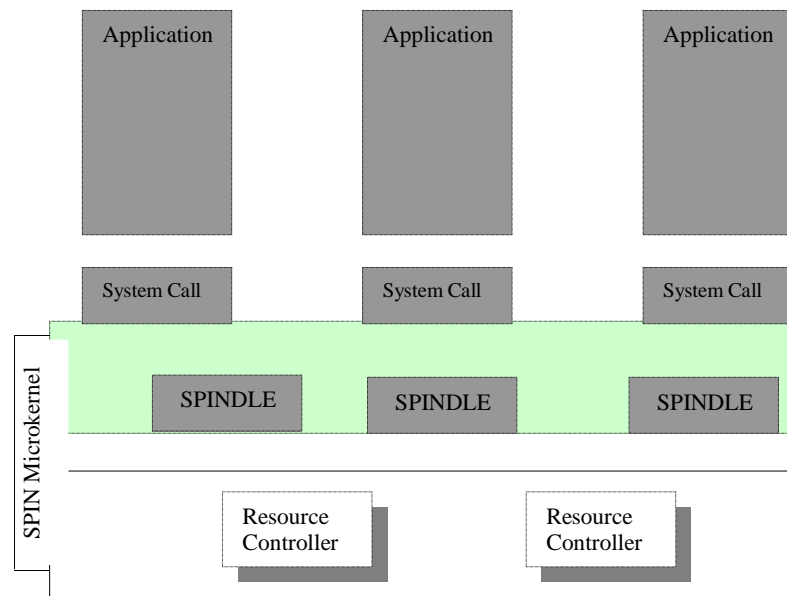


Fig. 4.1 Extensible Service Design in the SPIN operating system

SPIN enforces kernel integrity by requiring that all SPINDLEs must be written in a type safe language, Modula-3. Modula-3 has been used to implement much of SPIN kernel itself to demonstrate the expressiveness of the language and to enforce clean development. SPIN encapsulates kernel data structures in a secure and well-defined objects and the modula-3 compiler verifies (by type checking references) that SPINDLEs do not access interfaces that they do not have permission for.

4.3.1.2 VINO

VINO [Sletzer94] from Harvard University is of the same maturity as SPIN. It introduced the notion of grafts as the unit of kernel extension. Written in C++, grafts are dynamically linked into the VINO kernel address space by applications to adapt the kernel policies according to their requirements. Grafts are usually installed on a per-application basis, but it is possible for an application to replace a global policy. For example, an application can just replace the read-ahead caching policy for file access only for its open files or globally for the whole system. However, special privileges are required to replace a

global policy (c.f. the UNIX superuser). Grafts are linked into the kernel at designated graft points, which are of two kinds. The first is called the method overriding graft point that replaces a method inside the kernel. The second, called the event handler graft point, allows the insertion of application-defined handlers for kernel events, much like SPIN.

The kernel is structured as a set of objects encapsulating resources and exporting fundamental services such as virtual memory mappings and global scheduling algorithms for adaptation by inserting grafts.

VINO design has paid a lot of attention to identify and address threats to system integrity posed by dynamically installed extensions.

VINO kernel protects against two kinds of integrity violations, unprivileged access and resource hoarding.

The first threat with any form of in-kernel extension is the ability to access code and data that, if misused, can corrupt the kernel data structures and cause abnormal termination of applications belonging to other users. This issue is addressed in VINO at the memory protection level and is prevented using Software Fault Isolation (SFI) [Small96] techniques that sandbox grafts into dedicated logical regions within the kernel address space from which the code cannot arbitrarily escape without going through the pre-defined interfaces which limit the access of a graft. The VINO team have developed a compiler tool (MiSFIT) [Small96] for sandboxing arbitrary graft code at the assembly level. The code is also protected from external tampering by encryption.

The other integrity threat posed by extensions is the hoarding of resources that the grafts acquire. Malicious grafts can deny service to other applications via resource starvation. Two types of resource hoarding has been identified; time constrained, involving the holding of a resource for too long e.g. a mutex, and quality constrained, in which a graft attempts to hold enough shared resources to starve out other applications. Time constrained hoarding is handled by timeouts that are instantiated when a resource is contended and the holder is required to release the resource before the timer expires. For quality constrained hoarding, experimentally determined physical limits are placed on the amount of resources that can be held by a graft. To protect the system when such limits are disregarded, each graft invocation is executed within the context of a kernel maintained transaction log recording each grafts actions. A transaction rollback protocol

can be subsequently executed that deletes the graft and undoes its actions, thus restoring the system to a consistent state.

Though VINO's elaborate techniques to protect against misbehaving extensions are effective, they introduce significant complexity within the kernel and incur their own performance overhead. It is possible to develop a graft only to find its benefits are outweighed by the overhead of protection.

4.3.1.3 Fox

The Fox project [Harper98] at Carnegie Mellon University is aimed at applying advanced language techniques to system software in a variety of fields. Although the primary problem domain for the project is active networks where the integrity of the routers needs to be protected from the code carried by active packets, the protection techniques developed by Fox are also applicable to extensible operating systems. The Fox project has explored a number of language-based techniques for securing the actions of untrusted code but the approach most applicable to extensible operating system architectures is that of Proof Carrying Code (PCC) [Necula97]. Proof Carrying Code relies on (simple) formal theorem proving, where the task is split in two parts. The producer of the code attaches a proof (in first order logic) with the code to certify its validity, while the client is only required to verify the attached proof to ensure the validity of the code. Although the Fox team has not deployed the PCC support in their own operating system, PCC has been proposed as a generic way of securing arbitrary code execution in extensible operating systems.

A kernel that employs PCC, defines a safety policy that application-defined code must adhere to through the provision of an attached proof that conforms to the safety policy. The system derives a set of logical predicates for the code by inspecting it at run-time and verifying that this predicate is satisfied by the attached code. The novelty of this scheme is that it does not rely on encryption; if the proof is tampered with, it cannot satisfy the predicate, if the code is tampered with, the predicate generated will not conform to the proof. In addition, the PCC verification is done before the code startup; there is no run-time overhead.

However, the expressiveness of PCC language is limited and proofs for more complex analysis to protect against denial of service, like discovering loop-invariants, need to be inserted manually. We believe that PCC has not reached the maturity where it can be used to as a practical, general-purpose solution for resource hoarding problem.

4.3.1.4 SLIC

The SLIC [Ghormley98] (Secure Loadable Interposition Code) project from the University of California Berkley, is based on the GLUnix [Ghormley97] operating system layer to support NOW [Ghormley97]. In order to unify diverse operating systems, GLUnix requires the ability to interpose its own functional units over the underlying operating systems. Interposition is the insertion of trusted binary code onto an existing kernel interface i.e. within the kernel address space.

SLIC takes a pragmatic view of the operating system extensibility. The design of SLIC is based on the observation that commodity operating systems (UNIX in the case of SLIC) will not disappear overnight and hence a new design from scratch would be of little practical importance. SLIC uses well-known interfaces exported by UNIX and interposes functionality atop these interfaces to extend the operating system.

The architecture to support extensibility in SLIC uses three components: -

- Dispatchers that intercept events crossing kernel interfaces, e.g. system calls, signals and selectively redirects these events to extensions,
- extensions implement new functionality over existing kernel interfaces and
- utility functions that provide extensions with an abstract view of the system internal interfaces by hiding platform dependent details.

SLIC demonstrates good performance but suffers from lack of flexibility due to its commitment to provide backward compatibility; extensions in SLIC only build upon the interfaces provided by the underlying operating system.

4.3.1.5 Apertos

The systems described above only provide ad-hoc or semi-structured extension models, Apertos [Yokote92], on the other hand, is based on reflection.

Apertos introduces the concept of meta-object for introspecting and implementing the non-functional aspects (including persistence, replication, message handling etc.) of the base objects embedded inside the Apertos micro-kernel, metaCore. This separation of concerns allows base objects to be developed independently of the orthogonal behavior listed above. Each base object is associated with a group of meta-objects providing a meta-space for the execution of the base object. As meta-objects are themselves objects, they are also associated meta-objects in a potentially infinite tower of recursion. At the base of this tower is a meta-core, which offers microkernel-like primitives providing fundamental services available to each meta-object. Base objects communicate with meta-objects via methods that they export in their Meta Object Protocols (MOPs).

Apertos defines reflectors to represent a group of meta objects (i.e. meta space) implementing system services with which base objects can associate themselves. The process of meta-space migration is the central abstraction for extensibility with Apertos; any object can request to have its meta-space descriptor updated by transferring or replacing its current meta-object space with that of a new meta space offering different non-functional services. In order to migrate to a new meta-space, the source and target spaces must agree on a their compatibility levels for the base object to succeed in its request.

However, the immense amount of management information required by the system runtime in order to support meta space abstraction and all its involved processes and indirection incurs a large overhead (even though the implementation is done in C++). In addition, each object is associated with its own address space context, this implying a high inter-object communication overhead.

4.3.1.6 MetaOS

MetaOS [Douglas95] addresses these problems in Apertos. MetaOS attempts to adhere to the principles of meta level programming whilst enhancing performance and proposing a

way of assigning authority over who may introduce extensions to the system. MetaOS primarily improves performance by limiting the tower of recursion to just two levels, arguing that further levels are inefficient and rarely required. This two level hierarchy is split into local meta-spaces and global meta spaces. Local meta spaces allow per-object adaptation whereas global meta space can be used to change services that have an effect on all object in the system, e.g. the scheduling policy. Access control lists are used to ensure that only the designated objects can change the global meta space.

4.3.1.7 2K

As middleware systems are layered on top of an operating system, they, generally, cannot allow adaptation of services embedded inside the operating system. However, 2k [Kon00] from University of Illinois Urbana Champaign provides an integrated design, in which a reflective middleware (2K) is layered on top of a dynamically extensible kernel, (Off++, a later version of μ Choices) to allow dynamic adaptation. Although this strict coupling of the 2K Object Request Broker (ORB) with Off++ means that it cannot be used with other, non-adaptable, operating systems with different interfaces, the design of 2K provides an effective solution to address the requirements of dynamic adaptation.

The key to dynamic adaptation in 2K is architectural awareness i.e. the reification of structure, state and the behavior in order to enable the controlled introduction of new services. Adapting to changes in the environment is the responsibility of the middleware as implemented by a reflective ORB. The ORB admits changes to the system structure and uses the interfaces exported by the underlying operating system to adapt the system policies. An important part of the reification in 2K is the explicit representation of interdependencies between objects. This information can be used to change relationships between objects as part of system adaptation.

2K is being designed with similar goals to the ones presented in this dissertation; active space control and automation. Its reflective properties can be used to change internal system state to address the changes in the distributed environment.

4.3.1.8 Synthetix

The extensibility in above-mentioned systems is application-driven, where applications deploy extensions to tailor the operating system policies according to their requirements. Synthetix [Pu95], from the Oregon Graduate Institute, on the other hand, defines a self-extending operating system. The system self monitors the events to determine how applications are using resources within the operating system and then incrementally specializes the system code (system calls) that interact with those resources according to the usage pattern of the applications. To achieve this, Synthetix makes use of partial evaluation, using the TEMPO tool [Belkhatir94]. With this technique, system developers identify static and quasi-invariant conditions within system control paths according to the runtime status of these conditions and differing control paths can be selected (specialized) to exploit the optimizations made possible by the invariant conditions (partial evaluation). For example, the file write() system call in UNIX, can be optimized when the file is not being shared; all the concurrency code can be ignored. If, at a later stage, the file becomes shared then the system notes this change of invariant and replaces the specialized control path with the original version that supports concurrency. Synthetix also defines special replugging protocol that handles the case when the path being specialized is already being used by a system call.

4.3.1.9 Discussion

Dynamically extensible kernels have been a subject of appreciable research in the past few years. Other systems that have the provision for dynamically loading code into the kernel address space include Kea [Alistair96], Paramecium [Doorn95], Fluke [Ford99], MMLite [Helander98] and even the mainstream systems like Linux and Windows, through the use of application defined modules to extend the kernel functionality. Other systems that employ reflection techniques include DECADE [Kourai98a]. DECADE is unique in its use of an in-kernel interpreter to sandbox kernel extensions but its implementation suffers from large overhead of run-time interpretation. A weaker scheme, relying on the sparseness of a large address space, has been proposed in the recent version [Kourai98b].

SPIN's model of extensibility based on the use of synchronous events provides an effective scheme to allow the kernel primitives and user extensions to be decoupled from one another, enabling extensions to be added and removed as the kernel runs. However, it lacks support for reflection and its use of event guards to route events to appropriate extensions introduce extra complexity in the kernel and incurs performance overhead.

VINO provides a more flexible model than SPIN and supports method grafts that allow kernel functionality to be adapted at a finer granularity of an individual method, in addition to event handlers. However, both SPIN and VINO focus on extension and adaptation of primitives embedded inside the kernel, but do not provide support to introduce new services inside a kernel. Indeed, the services expected of a conventional uni-processor operating system are well-known and this model of extensibility is sufficient to adapt operating system services like virtual memory, scheduling and network communication. However, services required to manage an active space depend on the characteristics and requirements of a particular active space and the applications residing with a device. Hence, an operating system designed to manage and control an active space requires additional support to allow new services to be introduced in the kernel. Consequently, it needs to provide mechanisms to allow applications to discover and utilize these dynamically deployed services. Finally, it would need to allow even these dynamically deployed services to be extended to adapt their behavior according to the characteristics of the context of the device and applications residing with it.

Reflective architectures allow interfaces to introspect and discover the functionality of components and their relationships with each other e.g. Apertos, MetaOS, 2K. However, the level of reflection needs to be carefully balanced, or its overhead and complexity of use can outweigh the flexibility leveraged.

Similarly, the techniques to protect the system integrity need to balance between security and performance overhead arising from complex techniques e.g. transaction-based processing in VINO.

Apart from SLIC and 2K, none of the systems described above focus on adaptation of services in distributed systems. Among these, SLIC only focuses on adaptation of legacy systems (UNIX) to make them suitable for tightly coupled parallel processing. 2K is the

only system that allows for dynamic adaptation to suit the changing requirements of a distributed system.

Synthetix provides an interesting approach to automatic system-driven adaptation but only explores adaptation of system call control paths in a uni-processor environment. This approach of monitoring system events and adapting system code to address the changing requirements of the system holds promise for a dynamically changing distributed system as well.

However, none of the systems described above allow context-specific extension and adaptation warranted by longevity, mobility and dynamism in a ubiquitous system. Finally, none of the systems mentioned above has been designed for a resource constrained distributed, embedded environment.

4.4 Conclusion

This chapter reviewed the state of art in distributed and extensible operating systems and evaluated their suitability against the requirements outlined in chapter 3. This chapter leads to the following key observations:-

- The transparency provided by distributed operating systems is desirable to manage, control and program an active space, but the cost of transparency needs to be balanced by application requirements and system characteristics to allow efficient operation e.g. transparent process migration.
- Only dynamically extensible operating systems provide the flexibility to adapt the system according to the changing characteristics of a ubiquitous system
- Synchronous events allow existing operating system services to be extended, while reflection is imperative to allow new services to be introduced in the system.
- Protection of system integrity can incur appreciable overhead. However, it can be handled efficiently by limiting and authenticating extension privileges instead of imposing runtime checkpoints and recovery.

The next chapter presents the design of UbiqtOS, an adaptable, embedded, distributed operating system that builds on the observation made in this chapter to meet the requirements presented in chapter 3.

"It is not the strongest of the species that survive, nor the most intelligent, but the ones most responsive to change"

Charles Darwin

Chapter 5

Context-aware Adaptation in UbiqtOS: A Java-based Embedded Distributed Operating System

5.1 Introduction

Chapter 3 delineated the requirements for an operating system to enable fat devices in a ubiquitous system to control, manage and program an active space. It concluded that the heterogeneity, longevity, mobility and dynamism of the system warrant dynamic, application-aware and context-driven adaptation of all the services that could effect the interoperability, efficiency or availability of the system. Chapter 4 evaluated the state-of-art in distributed and extensible systems and motivated the need for a new bottom design of an embedded, extensible, distributed operating system to address the unique requirements posed by a ubiquitous system design.

This chapter presents the design of an embedded operating system, UbiqtOS, to enable fat devices to control, manage and program an active space with novel applications.

UbiqtOS comprises: -

- A small fixed part embedded in the ROM of fat devices, that is extended by
- Context-specific extensions to enable the device to efficiently interoperate in its environment.

By doing so, UbiqtOS offers context-aware adaptation to address the heterogeneity, longevity, mobility and dynamism of a ubiquitous system. The context of the device comprises

- Resources accessible to the device and
- Characteristics of the system that provides access to those surrounding resources

UbiqtOS adapts to the changes in its context by allowing

- Introduction of new services in the operating system according to the requirements of its context and
- Adaptation and extension of
 - Existing services embedded inside the operating system and
 - Applications residing with it

as the context of the device changes.

5.2 Contributions made by UbiqtOS

Chapter 4 presented an overview of extensible operating system designs to enable application-specific adaptation of operating system services. However, none of the systems provided support for context-aware extension and adaptation necessitated by the requirements of a ubiquitous system.

We propose UbiqtOS, which provides context-aware adaptation and extension using four basic constructs: -

- A registry that captures and exports the resources in the device context as a global namespace. Components residing with an instance of UbiqtOS export themselves to their context by registering themselves with the registry and, conversely, query the registry to discover other resources in their context. Therefore, by allowing components to discover and extend UbiqtOS, the registry provides a reflection

interface over the resources in an active space. More importantly, the registry generates events to notify interested components about changes in the device context to guide context-driven adaptation.

- A dispatcher module that imposes context-specific views over the global namespace, exported by the registry, to allow context-aware programmability.
- A subscribe/notify events architecture that serves to route synchronous events (c.f. SPIN) between components. Components can subscribe to events of interest happening in the system to monitor, extend and adapt the system functionality, and a
- Mobile agent engine that
 - Allows context-specific extensions and applications to be deployed at a host as mobile agents through its network connection and
 - Provides support for distributing services and applications among the resources in an active space to provide for load-balancing, fault-tolerance and high-availability according to the characteristics of the active space and the application requirements.

Context-specific extensions and applications are executed inside a Java Interpreter embedded in UbiqtOS, which provides a portable and safe execution environment for dynamically deployed code.

5.3 Design Goal

The operation of a time-sharing distributed operating system can be divided into two parts: -

- A local part that controls and securely multiplexes the hardware resources embedded in a device between different applications and
- A distributed part that provides, usually transparent, access to non-local resources

These two parts are tightly coupled in a distributed operating system, as compared to a middleware architecture, to provide an integrated, efficient design.

Though local resource management need to be adapted to tailor the operating system according to the requirements of the applications residing with it [Bershad95], it is, primarily, the distributed operation that needs to adapt to provide context-aware adaptation. Therefore, the fixed part of UbiqtOS only provides the mechanisms to control and securely multiplex the hardware resources embedded in the device, while allowing local resource management policies, like scheduling and caching, and distributed operation to be deployed (and adapted) dynamically according to the requirements of a particular context. Viewed differently, UbiqtOS is designed to be a simple extensible kernel, small enough to be embedded in the ROM of embedded devices, that lends itself to be dynamically extended into a distributed operating system according to the characteristics of the current context of the device and the applications residing with it.

5.4 Structure of the rest of the Thesis

This chapter gives an overview of the architecture of UbiqtOS and explains the interoperation of different components to enable context-aware extension and adaptation of the system. However, this chapter does not describe in detail the internals of different components. This is done in the next two chapters that respectively describe the architecture of the mobile agent engine and the registry embedded in UbiqtOS. Chapter 8 describes a prototype implementation of the UbiqtOS design presented in this chapter and chapter 9 evaluates the efficacy and performance of UbiqtOS.

5.5 System Architecture Overview

Figure 5.1 shows the architecture of UbiqtOS. The system architecture is divided in three layers. Layer 0 comprises an extensible microkernel, which contains platform dependent code to manage and export the embedded hardware resources to be controlled and programmed by higher layers.

This microkernel supports the following three components at Layer 1 to allow dynamic, application-aware and context-driven adaptation.

- A Java Virtual Machine, including Java core class libraries [Lindholm96]. The Java interpreter runs an extensible Java mobile agent engine, called SEMAS, to allow new software to be deployed in a device as mobile code. SEMAS serves to distribute software among the resources in an active space to enable new applications, provide interoperability, load-balancing, fault-tolerance and high-availability.
- An extensible registry that serves as a repository of all the hardware resources embedded with the device and the software installed with the corresponding instance of UbiqOS, and as a directory (yellow pages service) to find other resources in the context of the device. Further, this registry generates events to notify interested components about changes in the set of resources constituting the device context, to guide context-driven adaptation.
- A subscribe/notify events architecture, Romvets (introduced in chapter 2), that routes the events happening in the system to interested components. Extensions to UbiqOS use this interface to subscribe to the events offered by kernel primitives (c.f. SPIN) at layer 0, as well as by SEMAS and UbiqDir, to dynamically tailor the policies and distributed operation of the operating system. In particular, it is used by UbiqOS schedule, memory manager, network protocol stacks, SEMAS and UbiqDir to lend themselves to dynamic extension and adaptation. The events offered by UbiqDir, besides lending UbiqDir to dynamic adaptation, are used to notify interested applications (and network bindings, as described later) about changes in the context of the device as new resources become accessible or existing resources become inaccessible or properties of some of the resource in its context change. Further, it is used to monitor, control and program the system e.g. by using event scripts in AutoHAN.

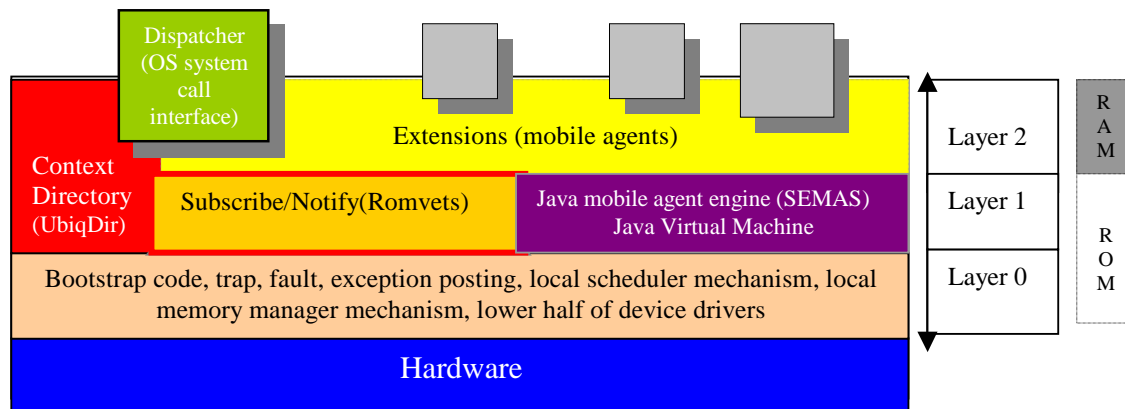


Fig. 5.1 System Architecture for UbiqtOS

This architecture allows new applications and context-specific extensions to be dynamically deployed at layer 2, to enable an instance of UbiqtOS to efficiently interoperate in its context.

The colored region in figure 5.1 represents this dynamically deployed software. Among it is a special component, called dispatcher, which, though can be replaced, needs to be present in an instance of UbiqtOS for proper operation. The dispatcher agent implements the default view exported by UbiqDir i.e. it makes the selection of resources, among those registered in UbiqDir, when requested to provide a service. Hence, it can be replaced to impose a view most suited to the requirements of the current active space of the device.

Below we introduce the three layers of the UbiqtOS architecture in more detail.

5.5.1 Layer 0: Extensible Microkernel

Layer 0 provides a minimal extensible kernel to support a Java virtual machine with a subset of core class libraries. The extensible kernel is structured much like SPIN [Bershad95]. When asked to provide a service, the kernel generates a corresponding synchronous event (c.f. SPIN), which is handled by an extension to provide the desired functionality. The kernel sends the events to the Romvets component which routes them to the extensions that had subscribed interest in them.

The minimal kernel may include an extensible scheduler and an extensible virtual memory manager for devices with persistent, mutable storage. Additionally it contains device drivers for embedded resources, and platform specific bootstrap-code. Thus, the extensible-kernel only implements local resource management mechanisms, allowing resource management policies to be deployed dynamically.

The extensible scheduler generates scheduler activations [Anderson92] that are handled by extensions to implement scheduling policies. Similarly, the extensible virtual memory manager generates page eviction events that are handled to implement page replacement policies for virtual memory. These two operating system policies have already been extensively researched in the past to allow application-specific adaptation in systems like Mach [Accetta88], SPIN [Berhsed95], VINO [Seltzer94], Exo-kernel [Engler95], Nemesis [Leslie97] to name a few.

Although the design of UbiqtOS kernel is amenable to application-specific extension and adaptation of scheduling and memory management policies, the main focus of UbiqtOS design is context-specific adaptation of distributed operation supported by layer 1.

5.5.2 Layer 1

The hardware dependent code at Layer 0 supports three components at Layer 1 in UbiqtOS. These three components provide the context-aware distributed operation in UbiqtOS.

5.5.2.1 SEMAS: Extensible Java Mobile Agent Engine

As mentioned earlier, UbiqtOS needs to enable effective utilization of additional resources in fat devices to control, manage and program their context. As embedded devices generally do not have a programming terminal attached to them, UbiqtOS needs to be able to allow new software to be deployed in the device over its network connection. Further, it needs to allow context-specific extensions to be deployed to address the heterogeneity, longevity, mobility and dynamism of the system. Once injected in the system, the system should be able to move the applications where it could best accomplish its task. Moreover, it needs to provide support to discover interfaces of

the newly deployed software in order to use newly deployed services. Finally, it needs to protect the integrity of the system from these dynamically deployed services and extensions.

UbiqtOS addresses these requirements by supporting a minimal Java interpreter on top of layer 0. Java addresses the above-mentioned requirements as follows:-

- The Java interpreter masks the heterogeneity of the underlying hardware to provide a portable execution environment [Lindholm96]. Hence, software can be developed and compiled elsewhere and transferred to the device to deploy new applications and to extend and adapt UbiqtOS to enable interoperation with its context.
- Correspondingly, designed for network programming, Java provides readymade facilities for marshalling and unmarshalling [Lindholm96] of data types to allow code and data to be transferred over the network.
- Java also provides a readymade reflection API, which can be used to dynamically discover and invoke interfaces of newly deployed software.
- Java provides dynamic type safety to protect the system integrity from foreign code. Note, Java's dynamic type checking and verification means that it does not need to trust the compiler, unlike Modula-3, to ensure type safety. Hence, Java bytecode compiled elsewhere can be safely executed on any device. Further, the Java interpreter provides a configurable sandbox, to execute the dynamically deployed code [Lindholm96], that can restrict access to legal resources.

However, Java suffers from the following shortcomings in addressing these requirements:-

1. Being an interpreted language, Java code cannot be executed as fast as native code.
2. Java in itself only supports pull-style mobility, where applets can be downloaded on a host on-demand. However, as we show in chapter 6, the mobility, dynamism and context-awareness of a ubiquitous system require push-style, proactive and reactive mobility as well.

3. Java's serialization protocol works by taking a closure of all the objects accessible from the migrating object and moving them with the object being moved. However, it is not desirable to migrate the whole object closure for every application. Instead, an application needs to be given the option whether the objects used by it are migrated with it, left on the original host or replaced with new objects on the destination host.
4. Though Java's reflection API allows introspection of objects, Java's object interface description is purely syntactic and does not support any semantic description that can be used to elucidate the functionality provided by an object.
5. Java does not support persistence of objects; services can only exist while they are being used by an application. As extensions to UbiqtOS would need to persist independent of their usage, UbiqtOS needs to augment the Java interpreter at layer 1 to allow persistence of services.
6. Finally, though Java's type safety restricts the dynamically deployed software from accessing prohibited resources, Java cannot protect against misbehaving extensions from hoarding resources. UbiqtOS addresses this shortcoming by requiring that all the extensions be authenticated before they are allowed to execute. Though this scheme requires pre-defined trust contracts, it avoids the considerable extra-complexity and performance overhead incurred by more elaborate schemes like transaction-based processing (VINO) or PCC (fox).

The design of UbiqtOS addresses these shortcomings as follows: -

1. We note that execution speed is not of critical importance in a ubiquitous environment where most of the tasks are I/O oriented i.e. control commands to control the embedded devices or soft real-time data streams between different devices. Still, the interpreter at layer of UbiqtOS uses JIT compilation [Tabatabai98] to improve performance.
2. Mobile agents [Lange98] provide the well-known generic component abstraction in UbiqtOS i.e. mobile agent serves as the well-known base class for all UbiqtOS components to allow interoperability. Mobile agents are objects that can be encoded with a task and can move (semi) autonomously between different hosts [Lange98] to

accomplish its task. To support this, the UbiqtOS Java Interpreter runs an extensible mobile agent engine, called SEMAS, to support proactive mobility, allowing software components to autonomously move in and out of the device. Additionally, SEMAS also allows mobile agents to migrate reactively, in response to the changes in the system, as resources fail, move, join or leave the context of the device to address the mobility and dynamism of the system.

3. SEMAS supports a scheme called effective mobility to allow software components to be moved to where they can best accomplish their task. This scheme is implemented by a combination of context-specific load-balancing, fault-tolerance and high-availability policies and application specific connection management that also addresses the object closure problem.

Whenever a software component requests to be connected to another entity, SEMAS notifies context-specific extensions for load-balancing, fault-tolerance and high-availability, that can suggest to migrate or replicate the requesting agent at the destination host. Further, system components (i.e. mobile agents) in UbiqtOS use explicit bindings [Leslie91] to communicate with one another and SEMAS informs all the bindings associated with the requesting agent whenever it requests to be moved to another host. The bindings can then decide whether they move with the agent, rebind to another agent or just invalidate themselves to deny access to the agent being accessed with the binding. The bindings can also deny the migration if they feel that the binding requirements would be violated with the migration. Therefore, effective mobility provides a framework to support a self-organizing system in which components, once injected in the system, can be automatically moved around to best accomplish their task.

4. SEMAS protects the system integrity by authenticating all the incoming agents. Sending SEMAS signs the outgoing agent with its private key allowing the receiving SEMAS to verify the incoming agents before they are allowed to execute in the system.

SEMAS uses a special protocol, called Agent Communication Protocol (ACP), to move agents to other hosts. SEMAS itself is extensible and the protocols and policies for

migration, load-balancing, reliability, disconnected-operation, fault tolerance etc. are dynamically deployed and adapted to suit agent requirements and the characteristics of the context of the device. To support extensibility, SEMAS generates events, using Romvets, whenever requested to migrate or replicate a mobile agent. Extensions subscribe interest in these events and are notified to implement context-specific distributed operation. The operation of SEMAS is described in detail chapter 6.

Lack of support for meta-data and persistence in Java are addressed by another component, UbiqDir, at layer 1 in UbiqOS, as described below.

5.5.2.2 UbiqDir

The mobility, dynamism and context-awareness of the system mean that applications and services can only be dynamically composed from the resources available in the, changing, context of a device. Moreover, applications cannot be expected to know the location or exact functionality of the resources with which it would need to interact to accomplish its task. Hence, applications need to be able to dynamically discover resources in its context, based only on *intent* [Winoto99]; an approximate description of the resource sought. Consequently, resources need to export themselves to the system such that other entities in the system can discover them by only expressing intent, and software resources, once introduced in the system, need to be able to persist independent of their execution time. UbiqDir addresses these requirements in UbiqOS.

UbiqDir, embedded in every substrate at layer 1, serves as the central repository of all the software installed with an instance of UbiqOS and as a directory (yellow pages service) to access all other resources in the context of the device. UbiqDir is essentially a simple XML database, allowing XML descriptions of entities to be registered, deleted, updated and looked up. UbiqDir allows components to look-up one another by just quoting an intent [Winoto99] of the service required; components can look-up one another by knowing only a subset of the functionality required. This property is critical to address the longevity and heterogeneity of a ubiquitous system where resources cannot be expected to know about the location or exact functionality of all the other resources in the system that it could be required to interact with to provide a service.

A special routine in Layer 0 registers the description, in XML, of both hardware resources embedded in the device and software installed with an instance of Ubiquitous, with UbiquitousDir as part of the bootstrap sequence. Subsequently, software components can be dynamically added to an instance of Ubiquitous by registering a reference to them with UbiquitousDir. Ubiquitous is dynamically extended and upgraded by, possibly unregistering older versions and, registering new components with UbiquitousDir.

These software components, structured as mobile agents, export themselves to their context by registering their properties, encoded in XML, and a handle to their functional interface with the local instance of UbiquitousDir. UbiquitousDir, in turn, exports the descriptions of the software components (mobile agents) residing with it (and the hardware resources embedded in the device) to other instances of UbiquitousDir (embedded in their respective instances of Ubiquitous) in the context of the device, using a protocol suited to the characteristics of a particular context. Conversely, components residing at a node can find other components in their context by looking them up in the local registry. Hence, UbiquitousDir reflects the state of the context of the device and lets both existing services be replaced and new components be introduced in an instance of Ubiquitous. Its interface, thus, serves to provide reflection over the resources in the context of a device.

As services are installed with an instance of Ubiquitous by registering a reference to them with UbiquitousDir, they cannot be garbage collected unless this reference is removed from UbiquitousDir. Hence, services can exist in the system independent of their execution time. This coupled with the property that all the components in the system are accessed using the interface of UbiquitousDir, independent of their type or location, UbiquitousDir serves as an orthogonal persistent store [Jordan98] for components registered with an instance of Ubiquitous.

Where the local operation of Ubiquitous allows descriptions of entities to be registered, deleted, updated and looked-up, its distributed operation exports these resources to the system by disseminates these descriptions to other instances of UbiquitousDir in its context. Conversely, UbiquitousDir allows non-local components in the context of the device to be discovered over the network. However, the scope of a meaningful context and the mechanism to discover it depends on the system idiosyncrasies of the current active space. Hence, the distributed operation of UbiquitousDir is extensible and the policies and

protocols to discover resources in the device context are dynamically deployed and adapted to suit the characteristics of the current context of the device.

5.5.2.2.1 Context-awareness

UbiqDir uses the Romvets architecture (described in section 5.5.2.3) to generate notifications for interested components whenever resources are registered, updated or deleted with it, indicating a change in the context of the device (shown in fig. 5.2). Extensions to UbiqDir subscribe to these events to implement the distributed operation of UbiqDir, as described in chapter 7. Further, applications, network-bindings and operating system services register interest in these events to be notified whenever the corresponding change happens in the context of the device. These notifications about changes in the device context are used to provide context-driven adaptation in a variety of ways in UbiqOS.

- The mobility and dynamism of the system means that the suitability of a resource to provide a specific service changes as devices move, leave or join the context of the device, changing the properties of the resources and making new choices available. Hence, applications subscribe to the events offered by UbiqDir to be notified when better choices become available and rebind their bindings accordingly.
- Similarly, network bindings need to adapt as the properties of wireless connections change as devices move in the network. Network bindings can adapt the fidelity (quality) of data when notified by UbiqDir about changes in the bandwidth and latency of the connection to access a device.
- Finally, operating system services, dynamically deployed at layer 2, can use these events about changes in the context of the device to move and replicate mobile agents to provide load-balancing, fault-tolerance and high-availability.

Context-aware adaptation leveraged by UbiqDir is covered in more detail in chapter 7.

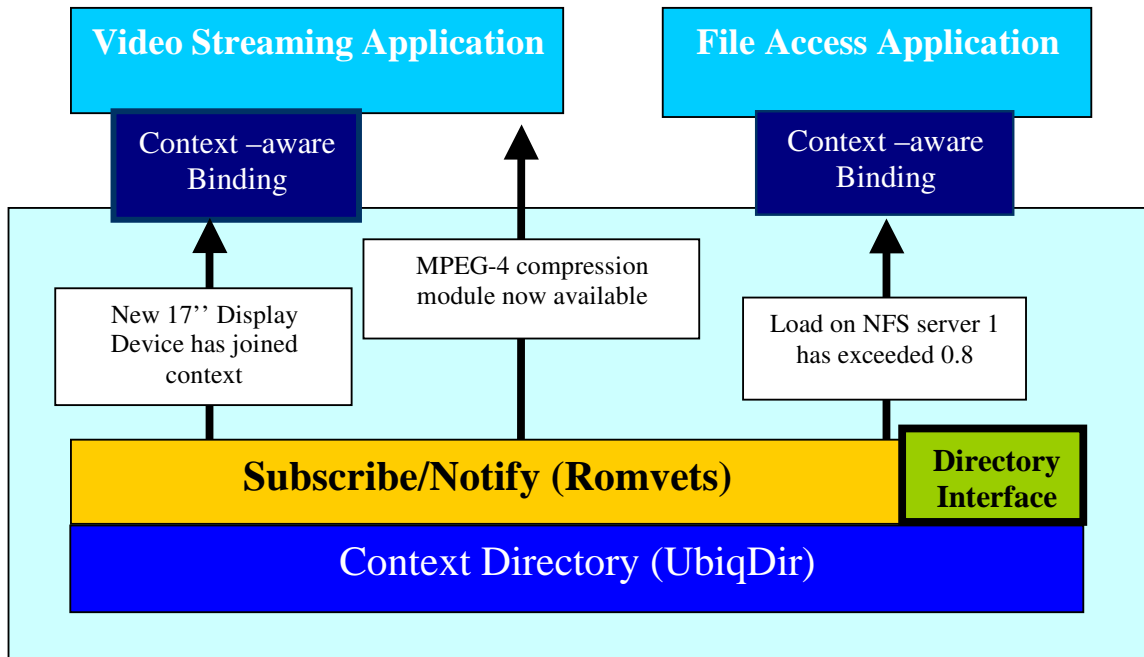


Fig. 5.2 Context-awareness in UbiqtOS: UbiqDir notifies interested components about changes in the device context using the Romvets Interface.

5.5.2.3 Romvets

New functionality is installed with an instance of UbiqtOS by registering new mobile agents with the registry (UbiqDir) embedded in UbiqtOS. Mobile agents, installed with an instance of UbiqtOS, find one another by querying the registry, which returns references of the components matching the description. These references are then used to invoke methods on the looked-up agents, using explicit bindings, to request a service.

However, where this scheme allows operating system functionality to be extended by installing new components with it, it, alone, is not sufficient to extend the behavior of the existing components embedded within the UbiqtOS substrate e.g. drivers at layer 0, UbiqDir, SEMAS.

These components use a modified version of Romvets architecture, implemented by UbiqDir, to generate synchronous events whenever requested to provide a service (c.f. SPIN). Extensions for these components are written as event-handlers and register themselves by subscribing to these events using the Romvets interface. Hence, context-

specific extensions for these services are deployed as mobile agents that subscribe to relevant events to implement and extend the behavior of the service. The extensible service, therefore, just serves to redirect the requests, whenever invoked to provide a service, and Romvets routes these requests to the appropriate extensions. In this manner, the services that lend themselves to extensibility and their context-specific extensions are decoupled from one another, enabling extensions to be added and removed as the kernel runs. Further, as UbiqtOS allows new services to be introduced, that might be extensible themselves, the events offered by a service are described in its description in UbiqDir. Extensions lookup the service description to find out the events offered by it, and subscribe to the appropriate events to extend its behavior. Therefore, UbiqDir serves to publish the events that Romvets serves to route and, hence, provide a general-purpose mechanism to discover and extend, even the dynamically deployed, services. This is different from traditional extensible kernels like SPIN, where the events generated by the kernel are expected to be well-known to extensions and, hence, which only support extensibility of a fixed number of services like scheduling, memory management and network communication. UbiqtOS, on the other hand, allows new services to be introduced in the system, to address the growing requirements of the users and changing characteristics of the system, and the publish/subscribe/notify architecture provided by Romvets and UbiqDir provides a general-purpose architecture to allow extensibility of any service residing with UbiqtOS.

These two schemes to support extensibility in UbiqtOS are shown in Fig. 5.3 and 5.4 respectively.

The Romvets interface supported in UbiqtOS, shown in figure 5.5, is a modified version of the Romvets architecture used in AutoHAN. Though it stores subscriptions in XML and implements mechanisms to address the mobility and dynamism of the system, like AutoHAN Romvets, it supports synchronous events (c.f. SPIN) instead of asynchronous events. Events in UbiqtOS are basically (decoupled) upcalls into subroutines, and the Romvets architecture serves to route these procedural invocations to the interested event handling routines. The use of XML to publish (using UbiqDir) and subscribe interest in events allows extensions to subscribe interest in the events using any

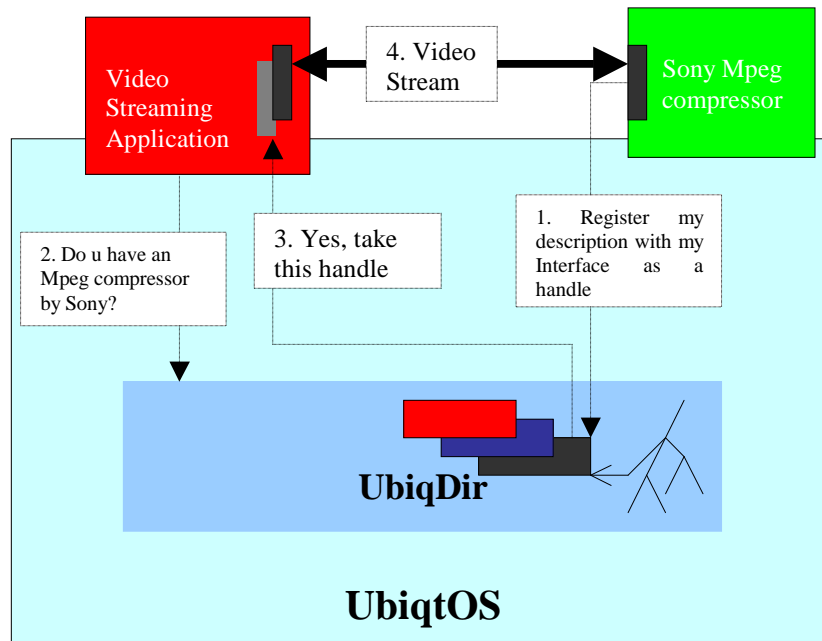


Fig. 5.3 Installation of new functionality in UbiqOS using UbiqDir

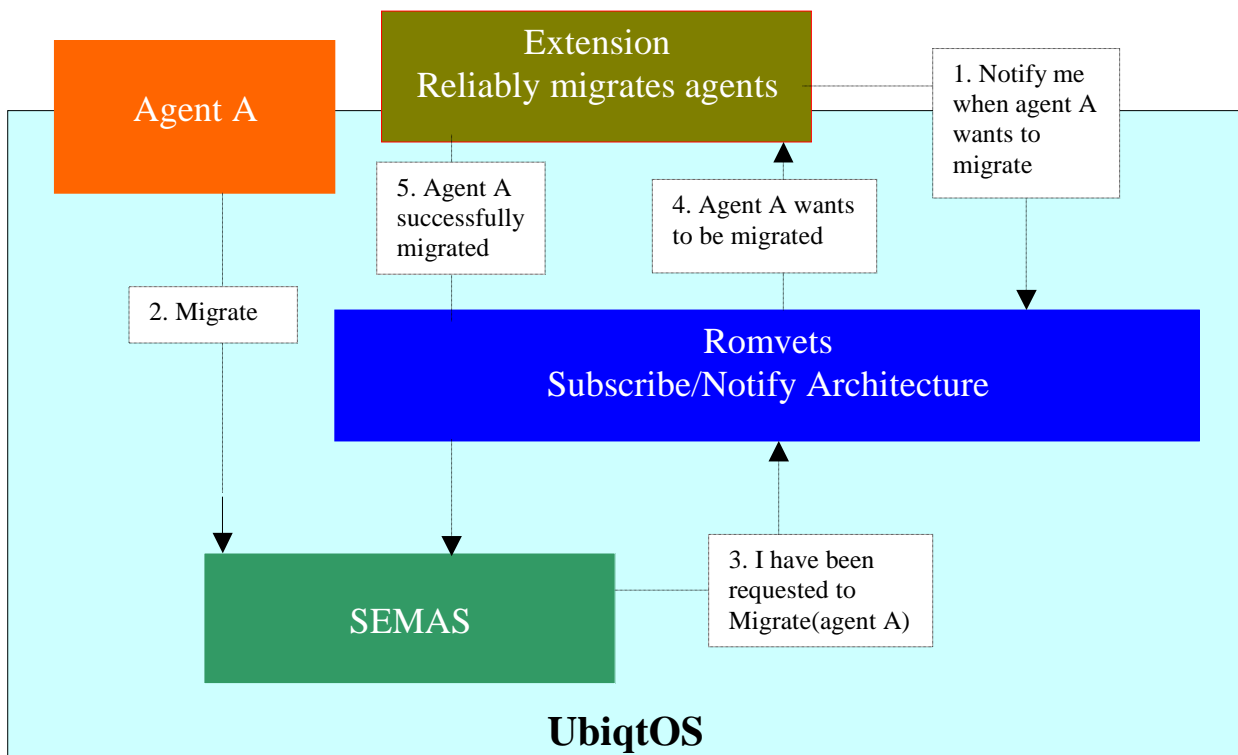


Fig. 5.4 Extension of services embedded in UbiqOS using Romvets

subset of arguments of the up-call (without having to specify all the arguments as in SPIN). This allows event handlers to be installed even if some of the arguments of the up-call are irrelevant for the event handler. The benefit of this flexibility is shown in chapter 8, where protocols (written as event handlers) can handle packets by subscribing interest in any one of the attributes of the packet like destination host or address type etc.

The synchronous events in UbiqtOS mean that extensions block when notified, perform the requested operation and return a result for the requesting service. Romvets invokes all the extensions that had subscribed interest in an event one by one, collects the values returned from all the extensions and passes the results on to the requesting service in a collection data type (Vector). It is up to the requesting service to interpret the results returned. This scheme is different from traditional event systems where the event service either does not support returned values (asynchronous events) or just takes a logical AND (even logical OR) of the returned results and returns the resulting boolean value to the requesting service [Bershad95]. Our scheme, clearly, offers more flexibility, allowing a wider range of services to be extended, as shown in chapters 6, 7 and 8. For instance, our scheme can be used by extensions to anonymously vote in response to an event, and the service generating the event can decide on the success or failure of the operation by counting the votes. Chapter 6 shows how this facility is used to implement effective mobility.

The extensions, indeed, can propagate these events on the network with the appropriate header set to address the dynamism of the system i.e. CAREOF. This allows events to be delivered to remote hosts as well, allowing programs residing in other devices to be notified about events happening in a device. Hence other devices can also control, monitor and program the functionality of the device by, per say, using event scripts.

However, this means that the time spent by an extensible service to perform a requested operation actually depends on the number and complexity of extensions deployed to handle its corresponding event. Worse, these extensions can block forever either maliciously or by error to cause denial of service.

UbiqtOS addresses this problem as follows.

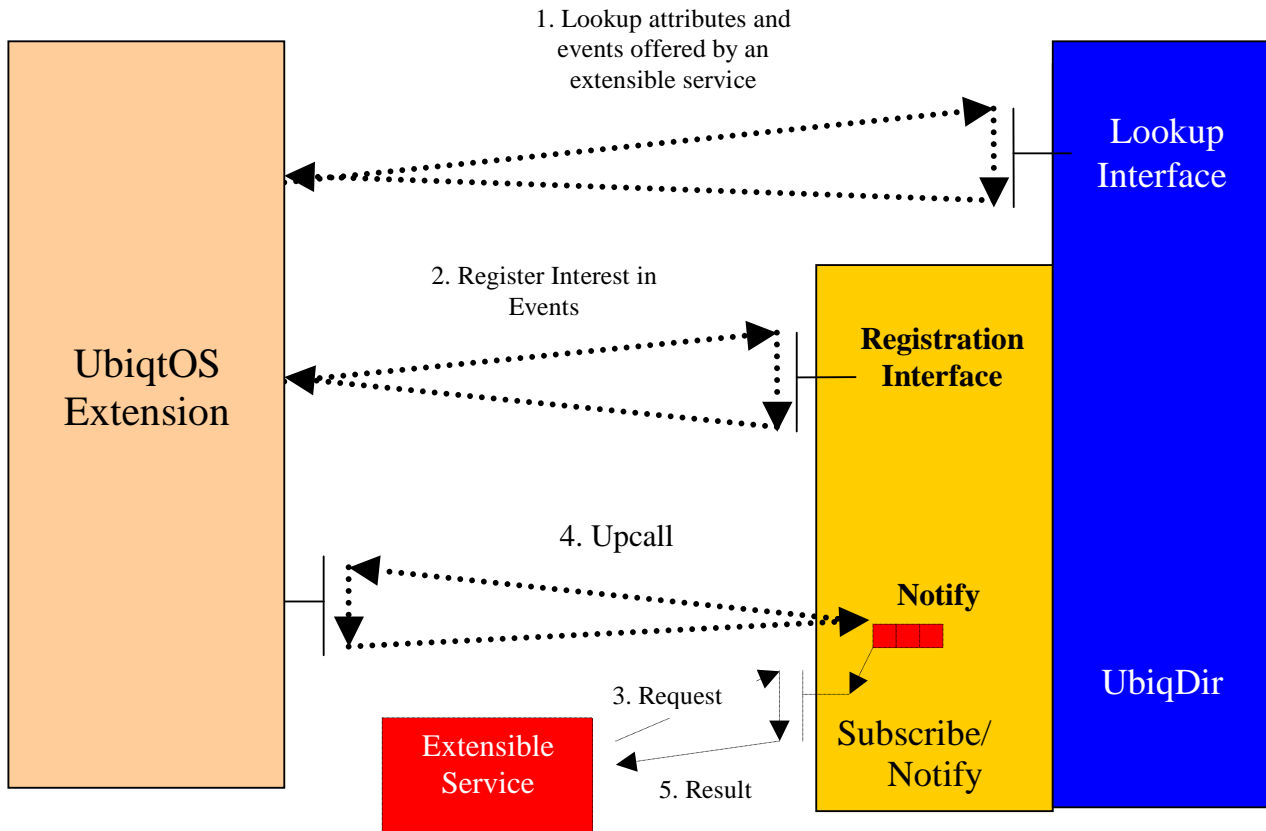


Fig. 5.5 Romvets Interface of UbiqOS:

Extensions to UbiqOS lookup the appropriate service in UbiqDir and subscribe interest in the events offered by it using Romvets' subscribe interface. When the extensible service requests a function, Romvets makes an up-call to the extension and returns the value returned by the extension to the extensible service.

Extensible services specify an access control list along with every event they publish with UbiqDir. SEMAS authenticates all mobile agents, and Romvets allows only those mobile agents to subscribe interest in the events offered by a service for which it has been granted permission by the service itself. Where this scheme ensures that only extensions from trusted sources can extend and adapt the behavior of the operating system, it is based on the assumption that extensions coming from trusted sources will be well-behaved citizens of the community. Well-behaved extensions are expected to return control as soon as they are finished processing the upcall, instead of hoarding resources and computation cycles. Further, extensions that do not actually implement the requested

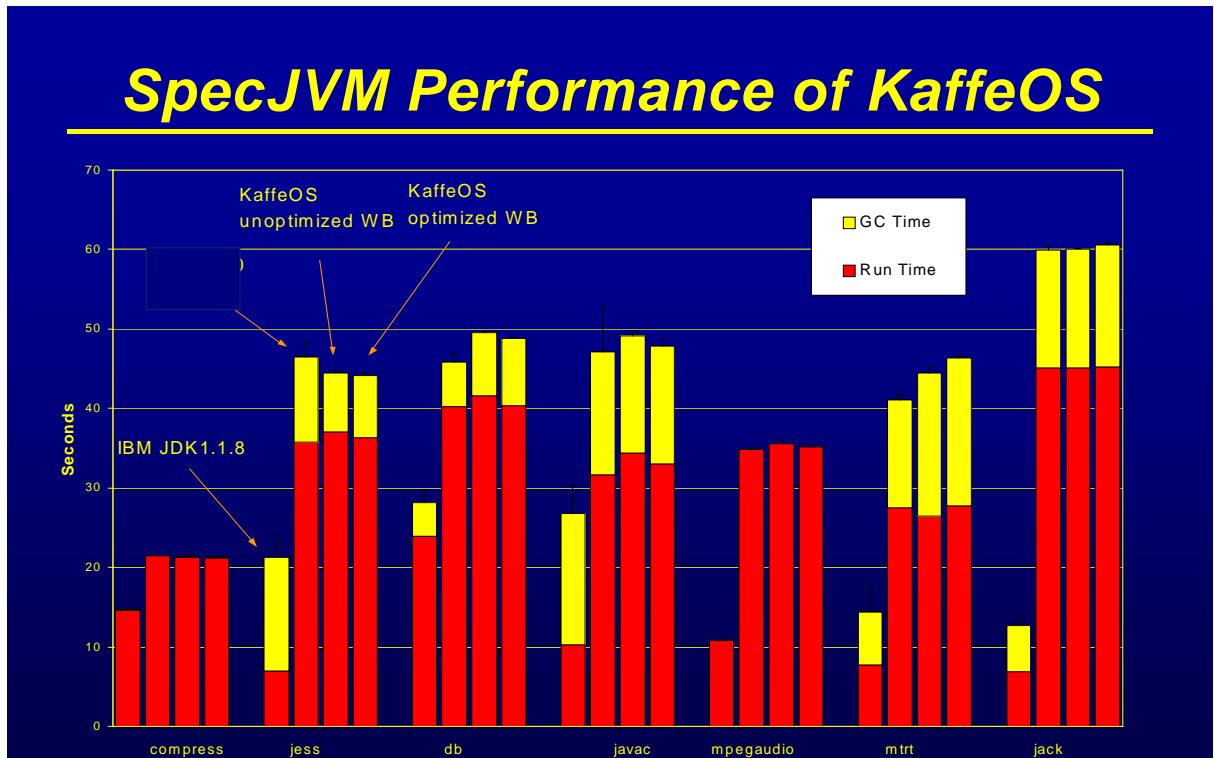


Fig.5.6 Performance comparison of JanosVM with IBM JDK 1.1.8(leftmost bars): JanOS process abstraction incurs as much as 50% extra cost to protect against memory hoarding attacks. (Flux Research Group, University of Utah)

operation, but are deployed just to monitor the system e.g. just to collect statistics, are expected to return control immediately upon notification, and perform their additional functions using a separate thread, instead of incurring the cost of their operations in the critical path of service provision.

Where a more stringent security model, like the one proposed in [Small96], would relax, to some extent, the assumption of good behavior from extensions, it would incur significant overhead in monitoring and terminating the non-cooperating extensions, as concluded in chapter 4. We believe that the hoarding problem in embedded devices is handled more feasibly at the authentication level.

5.5.3 Layer 2

The three above-mentioned constructs in layer 1 support applications and context-specific extensions at layer 2 in UbiqtOS. Applications and extensions are written as Java mobile agents and migrate or replicate themselves on the device to use its resources. New functionality is installed with the device by registering a reference with UbiqDir embedded at layer 1, whereas existing services are extended by registering event handlers for events offered by extensible services using Romvets. Finally, components at layer 2 subscribe to the events offered by UbiqDir to be notified about changes in the context of the device to allow context-aware adaptation.

All of the dynamically deployed components execute inside the single instance of the Java Interpreter as Java threads. This scheme provides a lightweight execution environment, as opposed to heavyweight processes, and allows efficient inter-component communication by avoiding context-switches between different address spaces i.e. intercomponent-communication in this scheme is essentially subroutine calls within the same address space. Where threads provide a lightweight model for sharing and accounting computation cycles between concurrently executing components, Java threads share the same pool of resources, e.g. heap, making UbiqtOS more prone to hoarding attacks. The alternative is either to execute every layer 2 component in its own instance of the Java Interpreter or to extend the Java Interpreter to support a process abstraction, as proposed in J-kernel [Eicken99] and Alta[Tullman98]. Spawning a new Interpreter for every new component is not feasible in resource constrained devices with limited RAM and computation power and negates the advantage of rich sharing of data in the lightweight threads model. Whereas, adding a process abstraction to the Java runtime, though a more economical approach [Back00], introduces considerable extra complexity within JVM to allocate and reclaim memory, and incurs appreciable overhead for intercomponent-communication. This overhead arises from the cost of managing a separate heap for every process, to protect against memory hoarding, and by enforcing namespace protection to restrict the process from accessing illegal resources in the system. Figure 5.6 shows the performance comparison of the JanosVM that supports a process abstraction, with a traditional VM, used in UbiqtOS. JanOS VM is as much as

50% slower than a traditional VM (compared to IBM JDK1.1.8 in this case). Where this extra complexity and performance penalty might be acceptable for high-end nodes, like active network routers, we believe that solving the hoarding problem at the authentication level is a better approach for resource-constrained environments, where, once authenticated, the dynamically deployed software can be trusted not to deny service to other components without any runtime overhead. SEMAS places all the mobile agents belonging to the same source in a single Java thread group. This allows threads belonging to the same source to be managed together. These thread groups can create further child thread groups that inherit the properties, and restrictions, of their parents as per the Java model.

However as components can lookup and use other resources in the system by querying UbiqDir with only a subset of their attributes, Java's type safety is not enough to restrict the dynamically deployed components from accessing illegal resources. This problem is addressed by the security model of UbiqDir. Every resource description registered with UbiqDir includes a list of resources that are authorized to access, modify or execute the component. UbiqDir authenticates the requesting component's identity and denies access to resources that are not in the access control list of the looked-up resource. This is explained in more detail in chapter 7.

Although the whole purpose of UbiqOS design is to allow new applications and services to be deployed and adapted according to the application requirements and characteristics of the current active space of the device, a typical instance of UbiqOS needs to execute some services at layer 2 that implement the distributed operation of the operating system to allow effective participation of the device in its context. At layer 2, these services can be adapted and replaced, using the artifacts at layer 1, to suit the characteristics of the current context of the device and applications residing with it.

The services provided by these components at layer 2 can be classified in 4 broad categories.

1. **Interoperability:** Components deployed at layer 2 allow resources embedded in the device to be efficiently used by other resources in its active space. This interoperability is provided by dynamically deploying a common set of context-

specific services and protocols in the all devices participating in an active space. As these protocols and services are dynamically deployed to suit the standards and characteristics of an active space, they reside at layer 2 and can be adapted and replaced in response to the changes in the context of the device. This means that protocols used to communicate with other devices on the network e.g. IP, SLP, which are fixed inside a traditional kernel, are placed at layer 2 in UbiqtOS, allowing, for instance, IPv4 to be replaced by IPv6 or a new protocol like Berkeley Snoop [Balakrishnan95] to be dynamically introduced to suit the standards and characteristics of an active space.

2. **Load-Balancing:** Software deployed at a device can be moved around in an active space, using SEMAS, to balance load among the devices participating in the system. However, the factors influencing the decision of where to place a software component to incur least cost depends on the characteristics of the active space and the application requirements. Hence, load-management services, which are fixed in traditional distributed operating systems, are placed at layer 2 in UbiqtOS. These services subscribe to the events offered by the extensible scheduler and memory manager at layer 0, to be notified about changes in the local load, and by UbiqDir and SEMAS, to be notified about changes in the context of the device, and move mobile agents between different devices to balance load in the system.
3. **Fault-Tolerance:** Similarly, services for providing fault-tolerance in a system are placed at layer 2 in UbiqtOS. Like load-balancing services, these services subscribe to the events generated by SEMAS and UbiqDir to be notified about changes in the context of the device, and implement functions like “heart-beat”, to indicate well-being of a host, and migrate and replicate software components to leverage fault-tolerance.
4. **Availability:** Likewise, policies for ensuring availability of the system depend on the size and characteristics of the system and the requirements of the applications

residing with the operating system. A typical instance of UbiqtOS supports services to provide reliability, consistency and disconnected operation to ensure that the operating system can reliably meet application requirements.

5.5.3.1 Dispatcher

UbiqtOS, by having UbiqDir as a layer 1 service, provides a platform where resources can be dynamically discovered, based on their properties, and, hence, applications and services can be dynamically composed from the resources available in the system. Where this addresses the dynamism and context-awareness of the system, it introduces extra complexity within the applications that are now, first, required to 1) dynamically discover the resources that provide the desired service and 2) select the most appropriate resource among those available in its context, before they can use the desired service. Instead, applications should only be required to present only those attributes of the desired service that are relevant to their operation, and the system should be able to interpret that description in the current environment (context) of the application to make an appropriate selection (c.f. Plan 9). Further, selection of the most appropriate resource could depend on factors peculiar to the characteristics and the standards of an active space that a general-purpose application, written to execute anywhere, could not be expected to know. These issues are addressed by the Dispatcher module at layer 3, which, though can be replaced, needs to be present in an instance of UbiqtOS for proper operation. The dispatcher agent implements the view exported by UbiqDir i.e. it makes the selection of resources, among those registered in UbiqDir, when requested to provide a service.

Hence, where UbiqDir exposes the properties of the resources constituting the context of the device to the applications, the Dispatcher module masks some of this additional complexity from the applications by allowing them to provide only the attributes that are relevant to them and selects the resource, among those returned by UbiqDir, that could best satisfy the application requirements given the characteristics and standards of a particular active space. This allows applications to be decoupled from the peculiarities of a particular active space; an application can be written to execute in any active space by

going through the dispatcher whenever it wants to invoke a resource providing a specific service.

Viewed differently, the dispatcher module provides the context-specific system call interface for the operating system. When invoked, the dispatcher looks-up the matching components in the registry, chooses one according to its configured policy, invokes its pertinent procedure (using Java reflection API), and returns the result to the calling component. Components at layer 2 send messages to the dispatcher and it routes them to the appropriate services. Individual applications, customized to an active space, can of course override the default system-wide policy of the dispatcher by accessing UbiqDir directly to find the appropriate resource and making its own selection as to what gets invoked to provide a specific service. As UbiqtOS is a distributed operating system, the services invoked by dispatcher might not be available on the local host, but distributed on various nodes within the environment. Dispatcher gives the illusion of a single ubiquitous operating system by finding a reference to the desired service (in the local instance of UbiqDir) and invoking its pertinent procedure either locally or by making a remote procedural call (using ACP, as described in chapter 7). As dispatcher agent implements the default view exported by UbiqtOS, it can be adapted to impose context-specific views. For example, when an application, executing in a handheld device running UbiqtOS, requests to “turn off the light”, the dispatcher can interpret this as “turn off all those lights which are located in the current room”. Or if an applications requests to “sound an alarm”, the dispatcher can choose the speakers that are placed at a location where their sound is audible in most number of rooms in the house -- such information cannot be expected to be known to the application or even to the resource itself, and only the dispatcher, deployed by the active space itself, can impose such a view on the namespace exported by the active space.

Figure 5.7 shows the use of the Dispatcher module to perform the operation “turn off the light”. The dispatcher module is invoked using its (static) method “call” that takes as parameters the description of the device, the method to be called and the arguments for the method, and returns a boolean value indicating the success or the failure of the operation.

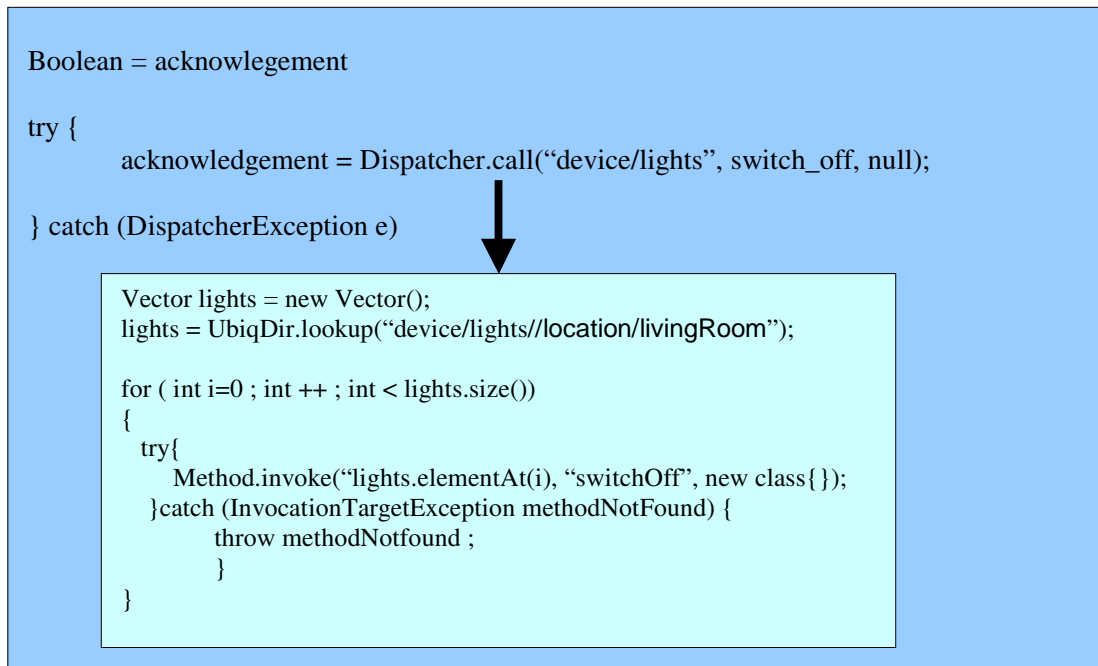


Fig. 5.7 An example of Context-aware system-call using the Dispatcher: The application only requests the lights to be switched_off, leaving the context-aware interpretation to the Dispatcher that carries out the request by switching off all the lights in the current active space i.e. livingRoom. This Dispatcher, deployed by a pre-configured device in the living room, is encoded with the policy that if the application does not specify the location of a device to be controlled, then invoke the operation on all the devices in the current active space by looking them up in UbiqDir and invoking their corresponding interfaces.

As the requested service might not be available in the device context, the dispatcher, instead of completing the system call, can throw an exception to indicate that the requested service cannot be provided. Consequently, layer 2 components invoking the dispatcher need to declare exception handlers (try catch blocks) before they can be compiled, which is natural as dispatcher uses the Java reflection API to dynamically discover and invoke the services (and by default throws the Java reflection exception). Finally, the dispatcher can cache references to avoid the cost of looking up in UbiqDir every time the service needs to be accessed, allowing references to frequently looked-up services to be cached to improve performance as described in chapter 9.

5.6 Bootstrap

When a device joins a specific context, it needs to be configured with the appropriate policies and protocols, according to its capabilities, to enable it to efficiently interoperate in the new context. Moreover, components in the system need to be moved and replicated to allow for load-balancing, fault-tolerance and high-availability.

In our architecture, a device is only embedded with layers 0, 1 with its hardware resources described in the local instance of UbiqDir. The bootstrap sequence of UbiqOS registers the drivers for the hardware resources embedded in the device with UbiqDir as mobile agents. However, these device drivers are special fixed, mobile agents, and deny all migration requests. Similarly, the bootstrap sequence also registers SEMAS and UbiqDir itself with UbiqDir, as fixed mobile agent. This synergetic model allows components in the system to deal with just one component abstraction i.e. a mobile agent; all mobile agent operations are parameterized with a single type regardless of whether they want to connect/migrate/replicate to a layer 0, layer 1 or layer 2 component.

Every embedded device wishing to participate in a ubiquitous system is, of course, also equipped with a network interface and a corresponding low-level networking technology. This interface is used to bootstrap the device to configure it to interoperate in a specific environment.

The presence of a new device in a network is detected by mechanisms part of the low-level networking technology supported by the device. For technologies with dynamic addressing, this bootstrap process also assigns a link layer address to the device. For example, FireWire [Wickelgren97] networks can automatically discover and assign addresses to new devices attached to its bus, devices in Warren [Greaves98] announce their presence by an ATM cell level beacon, Bluetooth[Haartsen00] devices use SDP and so on. When the link layer discovers a new device on the network, UbiqOS generates a corresponding event using the subscribe/notify architecture of UbiqDir. Special mobile agents, called bootstrap agents, subscribe to this “Bootstrap(String Address)” event and request the mobile agent engine to replicate them on the new device. Once replicated on the new device, these mobile agents query its UbiqDir to retrieve the descriptions of the resources embedded in the device and export these descriptions to the rest of the system

according to policies and protocols suitable for the system. Hence, this bootstrap allows other devices in the system to be notified about the resources of the newly joined device. Additional mobile agents can then be deployed to configure the device to make effective use of its resources. These mobile agents subscribe to the events offered by UbiqDir and by those by the mobile agent engine to implement context-specific distributed operation. Further, these mobile agents are used to implement appropriate policies for the *Dispatcher* to tailor the view exported by UbiqOS to applications and are used as explicit bindings to provide context-specific caching, transcoding, aggregation, and customization of traffic streams.

Note, in this scheme only the configured devices on the same link-layer-network can bootstrap a new device. In active spaces where more than one link-layer technologies co-exist, e.g. Firewire and bluetooth, this scheme can be extended to allow devices on different link-layer networks to configure one another. In this case, the bootstrap process proceeds in two steps. In the first stage, a configured device on the same link-layer network as the new device detects its presence and replicates a bootstrap agent to deploy a network layer, IPv4 in our case. This layer provides an inter-network addressing scheme and a well-known packet format. Once the network layer is in place, a special ACP (described in chapter 6) “advertisement” packet is broadcast on the IPv4 network to request for bootstrap agents. Hosts on another link can then replicate the bootstrap agents, using IP, on this new device. These agents export the resources of the device, on the IP network, using protocols and policies suitable for the system and allow more software to be deployed to configure the resources.

Where the first scheme relies on the plug-n-play mechanisms of the underlying link layer, the second scheme uses explicit “advertisement” messages to announce its presence to devices on another network. These advertisement packets, and indeed the bootstrap agents, are encapsulated in ACP (described in chapter 4). Chapter 6 shows how ACP can be configured to use the newly deployed IPv4 layer to proceed with the second stage of the bootstrap procedure.

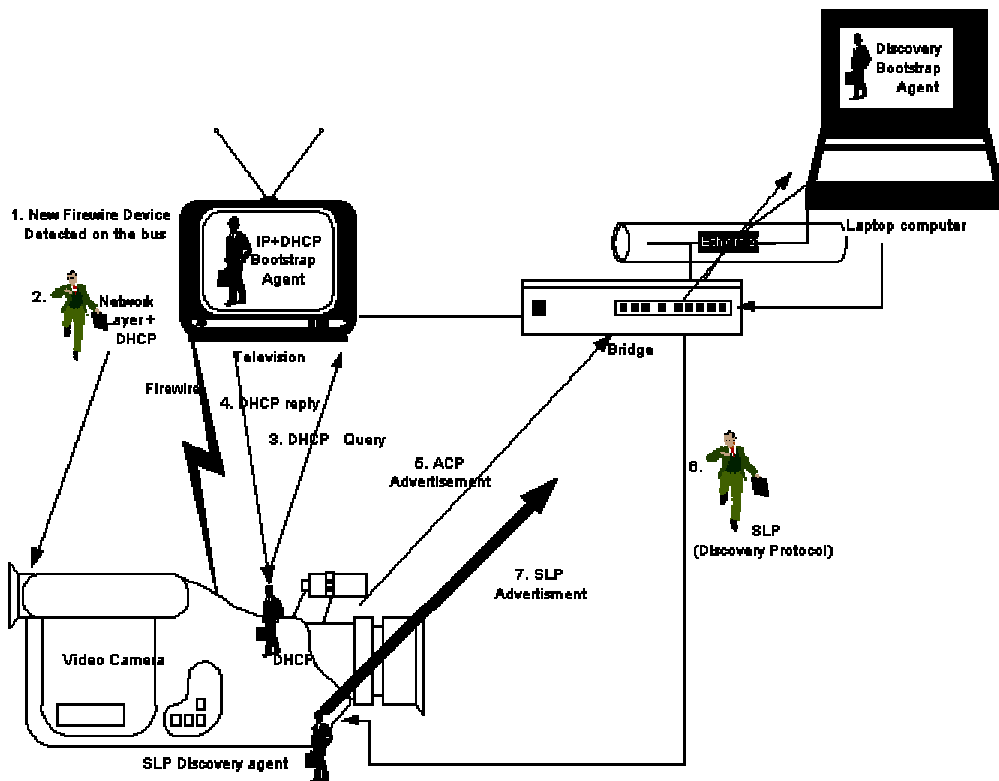


Fig. 5.8 An example bootstrap sequence in UbiqOS.

A video camera quipped with a firewire network interface joins an active space with a firewire equipped TV set that is already configured with IPv4 and DHCP. The IPv4 and DHCP agents residing in the TV set replicate themselves on the video camera, when notified with a “bootstrap” event, to configure the camera with IPv4 and a corresponding network address. ACP advertisements by the camera can then be received by a laptop connected with an Ethernet interface to the active space. The SLP agent in the laptop uses the ACP advertisements to replicate itself on the Video Camera and exports the resources registered with the UbiqDir embedded in the camera to other devices in the active space.

Where the current implementation of UbiqOS uses IPv4 as the network layer, bootstrap mobile agents can, of course, deploy and use any protocol other than IPv4 that suits the requirements of a particular active space. Similarly, different schemes can be used to acquire a unique address for the network layer. Bootstrap mobile agents can use either an ad-hoc protocol like AutoIP [UPnP] or a client-server protocol like DHCP depending on whether any host in the network has the capability to store a set of mobile agents and can

act as a highly-available server to distribute these addresses or not. Fig. 5.8 shows a two-step bootstrap procedure that uses IP and DHCP to configure a new device.

Therefore, once any device is configured on a network, the rest of the devices can be automatically configured to allow their resources to be effectively utilized by the system.

5.7 Summary

This chapter introduced the architecture of UbiqtOS as a substrate to instrument fat devices to effectively interoperate in a ubiquitous system. It outlined the use of mobile agents to extend and adapt this minimal substrate to tailor a device according to the mechanisms and policies suited to the characteristics of a particular active space. Further, it introduced the role of UbiqDir to capture and export the changing context of a resource to the mobile agents residing in it. The philosophy of exposing the changing characteristics of the system to the component bindings is highlighted and the use of explicit bindings to address the object closure problem is introduced.

This architecture leverages a peer-to-peer system architecture, where every device supports an instance of UbiqtOS to participate in the system as a first class citizen. Consequently, availability of any single node is not critical to system operation. Extensibility in UbiqtOS allows it to adapt according to its context. The complexity of the extensions deployed depend on the resources embedded in the device, where more privileged devices can support both a larger number of mobile agents and ones requiring superior system resources.

5.8 Prelude to following Chapters

This chapter introduced the overall architecture of UbiqtOS but did not explain its constituent components in detail. The rest of the dissertation discusses each aspect of UbiqtOS in detail.

Chapter 6 describes extensibility in SEMAS to enable context-aware adaptation using mobile agents and elaborates upon the concept and utility of explicit bindings.

Chapter 7 describes the architecture of UbiqDir and shows how it is used to capture, export and indicate the changing contexts of a resource according to the changing characteristics of the system. Section 8 describes a prototype implementation of the UbiqtOS architecture presented in this chapter. Chapter 9 shows how UbiqtOS is used to support novel applications, and presents an analysis of the performance evaluation of the prototype implementation to validate the design of UbiqtOS.

“Poetry is a rich, full-bodied whistle, cracked ice crunching in pails, the night that numbs the leaf, the duel of two nightingales, the sweet pea that has run wild, Creation's tears in shoulder blades”.
Boris Pasternak

Chapter 6

System Components

The use of mobile agents to extend and adapt UbiqtOS and their role as the sole component abstraction in the system has been mentioned several times.

Agent mobility serves four purposes in our architecture. 1) It is used to inject software into the system and place it where it can best accomplish its task 2) It is used to dynamically extend the capabilities of the participating resources to enable them to participate in the ubiquitous system. 3) It provides for load balancing, and hence better utilization of system resources and 4) provides fault tolerance by replication and/or migration of system services.

This chapter gives a detailed description of the architecture used to support mobile agents in UbiqtOS.

It motivates the need for mobile agents to extend and adapt UbiqtOS before describing the architecture of the mobile agent engine embedded in UbiqtOS. It describes how the mobile agent engine embedded in UbiqtOS supports a) proactive mobility b) reactive mobility and c) effective mobility to allow context-specific adaptation and self-organization of the system.

Further, this chapter elaborates upon the concept of explicit bindings and shows how the use of mobile agents to represent bindings provides a flexible, synergetic system design to address the mobility and dynamism of the system.

Finally, this chapter presents examples of application-aware and context-driven adaptation of distributed operation in UbiqtOS by describing extensible operation for load-balancing, disconnected operation and reliability in SEMAS.

6.1 Motivation

Chapter 3 identified the requirement for context-driven extension and adaptation of Ubiquitous OS to address the heterogeneity, longevity, mobility and dynamism of a ubiquitous system. It also highlighted the requirement for fat devices to support additional applications to allow the system to effectively utilize their additional resources.

As an embedded device, unlike a PC, does not have a programming terminal attached to it, these system extensions and applications can only be deployed remotely on the device through its network connection. This requires applications and system extensions to be modeled as mobile objects to allow them to be passed to the device over the network.

Object mobility is also required to redistribute and replicate this software to provide load-balancing, fault-tolerance and high-availability, as new resources join the system or existing resources move, fail or leave the system.

Further, objects need to be able to migrate and replicate autonomously without human intervention in an automated active space.

This makes the mobile agent paradigm [Lange98] the closest fit to these requirements. Mobile agents are software objects that can autonomously migrate from one place to another to carry out their tasks. Though this *proactive* mobility allows new software to be deployed in a device, to equip it to interoperate in a new active space, traditional mobile agents paradigm falls short of addressing all the requirements posed by a ubiquitous system.

- First, mobile agents in a ubiquitous system need to be able to relocate/replicate themselves *reactively* in response to the changes in the system. Software components residing at a device need to migrate and replicate to balance load, provide fault-tolerance and ensure high-availability as devices fail, move, join or leave the context of the device.
- Second, mobile agents paradigm proposes a model where all the remote interactions are accomplished by moving the mobile agent to the remote host and performing the function locally in that host. Clearly, enforcing this approach for

all interactions is not optimal. For example, other remote connections of the agent can be adversely effected by the migration, or the cost of migration can exceed the cost of remote interaction itself. In fact, the number and importance of factors influencing the decision to migrate or to carry out a remote interaction using RPC depends on the characteristics of a particular active space. Hence, policies providing mobility need to allow context-aware adaptation.

6.2 Contributions made by SEMAS

The simple extensible mobile agent system (SEMAS) embedded in UbiqOS addresses the above-mentioned requirements by providing the following novel features:-

- It provides support for both proactive and reactive mobility. Proactive mobility allows software components to move from one host, to another to accomplish their tasks, according to a pre-programmed itinerary, whereas reactive mobility allows software components to move/replicate themselves in response to the changes in the context of a device (as indicated by UbiqDir).
- Extensibility in SEMAS allows migration and replication policies to be extended and adapted according to the characteristics of a particular active space and application requirements, to allow context-driven adaptation.
- SEMAS, instead of enforcing remote evaluation for every remote interaction, provides a framework to implement effective mobility. This scheme allows migration, and replication, to be performed only when it can positively affect the system. Hence, mobile agents, once introduced in the system, are automatically moved to the appropriate location in the system where they can effectively carry out their task.

Further, SEMAS implements weak mobility and allows application-specific connection management as described in the next two sections.

6.2.1 Strong vs. Weak Mobility

Migration, or replication, of an object executing in a host requires its execution context to be duplicated at the destination host. The execution context includes the code to be executed, any data associated with the computation, snapshot of all the registers, the runtime stack, heap, accounting information, open file and network connection descriptors, signal handlers, environment variables and any other information specific to the executing platform required to resume the thread of execution at the destination host [Tenenbaum90].

Strong mobility refers to thread mobility where the entire execution context is transferred with the migrating object by the executing platform [Douglis91]. The migrating thread can migrate at any arbitrary point during its execution and resume at the destination host.

Weak mobility, on the other hand, only involves migration of code and data, while other execution state and descriptor tables are not migrated with the object [Lange98]. Therefore, on a platform that only supports weak mobility, any state required to resume execution at the destination host has to be saved and extracted explicitly by the application as part of object data [Lange98]. Though the compiler can insert code to save and extract state to mask some of this complexity from the application, this is not possible in a dynamic environment, with support for reactive mobility, where points of migration are not known at compile time.

Hence, strong mobility, though more flexible, introduces complexity within the platform whereas weak mobility pushes most of this complexity to the application itself.

Keeping with the design goal of keeping the core simple, UbiqTOS only supports weak mobility. Weak mobility not only makes the core simple enough to be accommodated in embedded devices, it is sufficient for our purposes due to the following reasons:

- Node failures and disconnections happen without warning in a ubiquitous system and hence do not benefit by the considerable extra complexity introduced by supporting strong mobility. Even strong mobility cannot guarantee that a mobile agent would be able to move out of a node that fails without warning or gets disconnected from the rest of the system.

- Still, strong mobility clearly offers more flexibility for balancing load in the system as it, unlike weak-mobility, allows load-balancing to be performed at any point during the execution of the program. However, recent research has shown that system re-organization decisions for load balancing can benefit from optimistic distributed scheduling [Sokol90][Douglis91], which could be supported on top of weak mobility. Optimistic distributed scheduling proposes that instead of migrating a thread as soon as the load on a machine exceeds a certain value, it is more beneficial to delay the decision. This avoids spurious migrations in response to temporary spikes in load pattern to give a better performance. Especially in a ubiquitous system, where resources join and leave the system frequently to erratically change the system load characteristics, optimistic policies provide a more stable approach to load balancing. As optimistic load balancing does not require instantaneous migrations at arbitrary points, it can be supported on top of weak mobility.

Load-balancing in UbiqtOS is achieved by the local scheduler and memory manager sending notifications to the agents whenever system load exceeds a certain threshold. The mobile agents can then prepare themselves to migrate, by saving any desired computation state, and request SEMAS to migrate them when they are ready. Where this scheme requires cooperation from the mobile agents, to balance load in the system, it makes the UbiqtOS core considerably simpler by avoiding the complexity required to support preemptive migrations [Tenenbaum90].

6.2.2 Application-specific Connection Management

An important component of the execution-state associated with a migrating agent is its bindings with other resources in the system, such as files, network connections and other mobile agents. Current mobile agent systems either do not provide support for migrating the bindings associated with a mobile agent [Lange98] or perform the migration opaque to the migrating application [Wojciechowski99].

Whereas, the decision to preserve, discard or to substitute the binding with an equivalent resource at the destination host depends on the semantics of the application. For example, only the application can decide whether it is better to preserve a binding with a particular

file on the original host, close the file on migration or replace the binding with a new file on the destination host.

SEMAS addresses these requirements by making two contributions:-

- Bindings are *reified* as mobile agents to provide a unified, clean programming model.
- These bindings are notified by SEMAS whenever an agent needs to be migrated to allow the application to adapt the binding appropriately.

6.3 Comparison with Related Work

Mobile agent systems have recently received appreciable attention from the research and industrial community alike [Wojciechowski99] [Lange98]. A mobile agent system allows medium to small sized objects to autonomously move from one host to another to accomplish their task [Lange98]. So, for instance, unlike Java applets that are passive objects, downloaded on demand by an application, mobile agents themselves encode an application and can move from one host to another to accomplish their task. Viewed differently, mobile agent systems offer an application-level alternative to process migration [Douglass91] in distributed operating systems. Where distributed operating systems use mobility to transparently migrate processes to balance load in the system, mobile agent systems expose the mobility API to the applications to allow a wider range of choice [Chess95]. The difference between different mobile agent systems is in the API offered to the agents. Mobile agent systems that support strong mobility, like ARA [Peine97], Nomads [Suri00] and Nomadic-pict [Wojciechowski99], allow agents to `jump()` to another host at an arbitrary point during their execution. However, mobile agent systems that only support weak mobility, like Aglets [Lange98], Hive [Minar99], Concordia [Wong97] and Ajanta [Karnik98], allow migration at only fixed points during the execution of the agent. Systems that support weak mobility come in two flavors. Systems like Aglets [Lange98] and Hive [Minar99] present an event-based API whereas systems like Concordia [Wong97] and Ajanta [Karnik98] provide a subroutine-based

model. The event-based model requires agents to implement an event-handler that is called by the agent system, with an event specified by the agent as an argument to the `migrate()` interface, to notify the agent that it has been relocated to the requested host. It is up to the event handler to resume execution of the agent, by calling a subroutine, depending on the value of the event. The subroutine-based mobile agent systems provide a more direct API, where a migrating agent specifies a subroutine to be called at the destination host to resume execution upon migration. Having programmed with both type of systems, we found it much easier to program with subroutine-based systems, which preserve the logical flow of execution and do not burden applications with the extra complexity of mobility-event-handlers. Hence, SEMAS supports a subroutine-based model to support *proactive* agent mobility.

However, where existing mobile agent systems allow software to be injected and moved in a system to allow them to accomplish their task, they enforce a mobility model where agents encode a fixed itinerary to accomplish their task. Where this might be sufficient in traditional distributed systems, it, alone, is not enough to address the mobility and dynamism of a ubiquitous system. Software components, in a ubiquitous system, need to move as resources move, fail, leave or join the system changing the system characteristics. SEMAS address this by supporting *reactive* mobility. Reactive mobility is supported by using events from UbiqDir to notify agents about changes in the system. However, it is up to the agents to move or replicate when notified about these changes in the system. This combines the best of both worlds; reactive mobility is supported, as in distributed operating systems, while exposing the mobility API to applications, as in mobile agent systems.

Further, traditional distributed systems used RPC [Birrell84] to make remote invocations. The proactive mobility model in mobile agent systems, on the other hand, enforces a paradigm where every remote interaction is carried out by migrating the application to the destination and performing the task locally in the destination host. Whereas, the mobility and dynamism of a ubiquitous system mean that the characteristics of the system change dynamically, and therefore, the decision to accomplish a remote task by migration or remote invocation (RPC) can only be made at runtime depending on application requirements and system characteristics. SEMAS addresses this problem by providing

effective mobility. Effective mobility allows applications to make informed decisions about whether to migrate or make a remote invocation depending on the system characteristics and application requirements.

Finally, existing mobile agent systems implement fixed protocols and policies for migrating and replicating agents, whereas the heterogeneity, longevity, mobility and dynamism of a ubiquitous system require that these policies and protocols be dynamically adapted according to the current context of the device and the applications residing with it. SEMAS is a runtime extensible mobile agent system and allows its distributed operation to be dynamically adapted according to system characteristics and application requirements.

6.4 Structure of the Rest of the Chapter

Section 6.5 describes the mobile agent interface implemented by software components in UbiqtOS to participate in the system as first-class citizens. Section 6.6 describes the role of explicit bindings in UbiqtOS and shows how the mobile agent interface is extended to support explicit bindings as first-class citizens in UbiqtOS. Section 6.7 describes the architecture of the extensible agent system embedded in UbiqtOS and explains how it uses Romvets to lends itself to context-driven adaptation. Moreover, Section 6.7 describes effective mobility and presents examples of how distributed operation is adapted in UbiqtOS to tailor the policies for migration and replication. Finally, section 6.8 concludes the chapter.

6.5 Mobile Agents

Every component implements the mobile agent interface (mobet) interface shown in figure 18 to become a first-class citizen in the system. By first-class citizen we mean that a component implementing the mobet interface can be dynamically

- 1) Deployed as a context-specific extension or application
- 2) Authenticated by the system,

- 3) Discovered by other components in the system,
- 4) Notified about changes in the context of the device, and
- 5) Relocated, replicated and removed in response to the changes in the context of a device.

Hence, a first-class citizen when injected in the system, can discover the resources in its context and move around to accomplish its task despite the changing characteristics of the system.

The mobet interface, shown in fig 6.1, includes methods to support mobility, authentication and life-cycle management of the agent. Additionally, it includes event handlers that are invoked to notify an agent about changes in the context of the device to allow context-aware adaptation. Different components extend this interface to provide specific services.

Agents are launched in the system by calling the launch method of the agent engine embedded in UbiqOS. SEMAS relocates the agent at the appropriate host, to balance load in the system (described in section 6.7), and calls the bootstrap method of the agent at the new host.

The bootstrap() method initializes the agent state and calls another method to start the task of the agent. Therefore, the bootstrap method of the agent is analogous to the constructor of a regular Java Object.

After bootstrap(ing) an agent, SEMAS calls the Get_description() method of a newly received agent. The Get_description() method returns meta-data, in XML, to describe 1) the non-functional attributes of the agent, e.g. location, manufacturer (described in chapter 7), 2) any functional interfaces in addition to the mobet interface implemented by

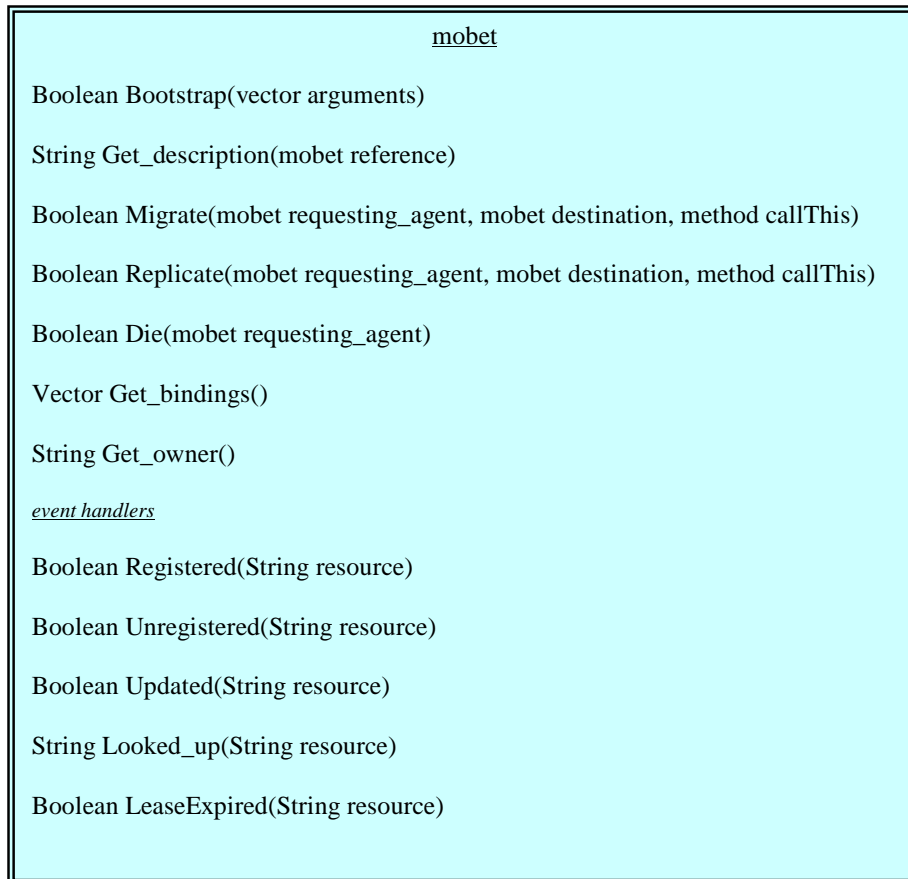


Fig. 6.1 Mobet Interface. Components implement this interface to participate as first-class citizens in the system.

the agent, and 3) any events handled by the agent in addition to the event-handlers for events generated by UbiqDir. The `Get_description()` method also returns a reference to the functional interface of the agent, by setting the value of the value-result argument passed to it by SEMAS.

SEMAS uses this information to install the new agent with the local instance of UbiqOS. To install the new agent, SEMAS registers the reference to the interface of the newly received agent and the XML description of its properties with UbiqDir.

The XML description returned by `Get_description()` method, and registered with UbiqDir, is used to dynamically discover components in the system. As the whole purpose of the UbiqOS design is extensibility, i.e. new components can be introduced in the system to address changing requirements, components cannot be expected to know, a priori, the attributes and interfaces of all the components they would interact with in their

lifetime. Instead, components in UbiqOS dynamically discover one another by their XML descriptions registered with UbiqDir.

Where the non-functional attributes are used to discover the properties of a component, the meta-description of the functional interface of an agent is used to discover and invoke methods on the component. Applications find the agents with the desired properties and use the description of their functional interface to select an appropriate method to invoke the service.

Individual components implement and extend the mobet interface to provide specific services, at layer 2, in UbiqOS. Where, the method and event handlers in the mobet interface are expected to be well-known, any additional methods or event handlers are described as part of the meta-data returned by the the Get-description() method. The signatures of the methods implemented by an agent are described by the name of the method followed by a list of its formal arguments, as shown in figure 6.2. The entity looking up the agent description can use this reflective information to discover and invoke an appropriate method by passing the method name and the type of the arguments to the Java reflection API [Javaref98].

UbiqDir stores the reference to the interface of an agent in a separate data structure and adds the corresponding index, to that data structure, to the XML description stored for the agent, in the “Interface” tag.

The interface index is, however, converted to the corresponding interface reference by UbiqDir, and returned along with the XML description of the agent, when the agent is looked-up in UbiqDir. Hence, the interface index tag only serves as an internal mechanism in UbiqDir to store interface references and is not visible to the entity looking up the agent.

The “Description” tag for an interface reference can be used to store any additional information to explain the semantic operation of the method. It could be anything from a string of words describing the function provided by the method to a more formal representation as a set of pre/post conditions for the method invocation [Kiniry98]. Our current implementation uses a simple scheme where the function provided by the method is described in English, but a more formal description is clearly more desirable to ensure correct operation of the system.

```

<Interface>index_in_interface_vector</Interface>
<MethodName> method_name </MethodName>
<Arguments>
  <Argument Number=argument_number Type=type_of_argument/>
  .....
</Arguments>
<Description> Description of the operation performed by the method
</Description>

```

Fig. 6.2 Functional Interface Description

```

<EventHandlerName> method_name </EventHandlerName>
<Arguments>
  <Argument Number=argument_number Type=type_of_argument/>
  .....
</Arguments>
<Description> Description of the operation performed by the method
</Description>

```

Fig. 6.3 Description of Eventhandler

Any event handlers, in addition to those in the mobet interface, implemented by the agent are also described in a similar format, as shown in figure 6.3.

Finally, the description of the agent includes the description of all the events (upcalls) generated by it to request extensible operation. Other layer 2 components, that implement the extensible operation, lookup the events generated by these extensible services, at layer 2, and subscribe interest in them using Romvets. This allows new extensible services and their extensions to be, independently, added to the system. This is different from extensibility in traditional extensible kernels like SPIN, which only support a limited number of predefined, fixed extensible services embedded in the kernel and hence do not provide a general-purpose mechanism to allow extensibility. UbiqOS allows extensibility of even dynamically deployed services by allowing extensible services to publish their events as part of their description registered with UbiqDir, where extensions

can look them up and subscribe interest in them, using Romvets, to implement extensible operation.

The next two methods in the mobet interface support mobility.

Mobets can be requested to move or copy themselves to another host by calling the Migrate() and Replicate() methods of SEMAS (as described in section 6.7). The agent engine, in turn, calls the Migrate() or Replicate() method of the specified agent, passing it the destination agent, the requesting agent and a value/result argument that is used by the agent to return the method to be called to resume its execution at the destination host, if it agrees to migration. The mobet requested to migrate or replicate can either deny the request by returning a false value, and setting the last argument to null, or honor the request by returning a true value to allow the agent engine to perform the transfer (and setting the last argument to the appropriate method). As SEMAS only implements weak mobility, a call to migrate() or replicate() gives the specified mobet a chance to save any state it needs to preserve across migration or replication to the specified host and to name a method to be called at the destination to resume its execution.

An agent can be requested to remove itself from the system by calling the Kill() method of the agent engine (described later). The agent engine, in turn, calls the Die() method of the specified agent and passes it a reference of the requesting agent. The agent can honor the request by cleaning up any state associated with the agent e.g. active bindings and returning a true value to the agent engine. It also unregister itself from UbiqOS by removing its reference from UbiqDir before returning a true value. The agent can deny the request by returning a false value if the requesting agent cannot be allowed to terminate the agent. Hence, the Die method compliments the bootstrap method to facilitate life-cycle of software components.

The Get_owner() method of the agent returns the name of the owner of the agent signed by his private key to authenticate the agent.

Finally, the agent interface provides a method to return the bindings of the agent with other components in the system. These bindings can be replaced or adapted as the properties of the connected components change, connected components become inaccessible or better choices become available. Section 6.6 describes the role of these

explicit bindings in the system and section 6.7.3 shows how this is used by SEMAS to support application specific connection management to implement effective mobility.

6.5.1 Reactive Mobility

Where agents call the `migrate()` or `replicate()` methods of SEMAS to request migration or replication to another host to carry out their pre-programmed task, agents also need to be moved in response to the changes in the context of the device as resources move, fail join or leave the system.

These changes in the context of the device are indicated by the events offered by UbiqDir. Mobets can subscribe interest in these events to be notified about changes in the context of the device. Therefore the mobet interface includes event handlers for events offered by UbiqDir to be notified about changes in the context of the device (as described in chapter 7). These events are generated by UbiqDir to notify the agents whenever a new resource is registered with the device, or an existing resource is updated or deleted from the device context. This allows mobile agents to move or replicate themselves in response to changes in the device context, to provide interoperability, load-balancing, fault-tolerance, high-availability. This aspect is described in more detail in chapter 7.

6.6 Explicit Bindings

Mobile agents in UbiqOS interact with one another using explicit bindings [Leslie91] [ODP95], which are themselves first-class citizens in the system. In UbiqOS, bindings between mobile agents are themselves special mobile agents that extend the mobet interface to support additional methods for connecting and passing messages between mobile agents. Being executable programs themselves, these bindings allow any computation to be interposed in the communication path between two components. Therefore applications just nominate the binding that provides the appropriate service (by looking up the attributes in UbiqDir) and are decoupled from the details of, possibly changing, characteristics of the underlying system. Further, as first class mobile agents,

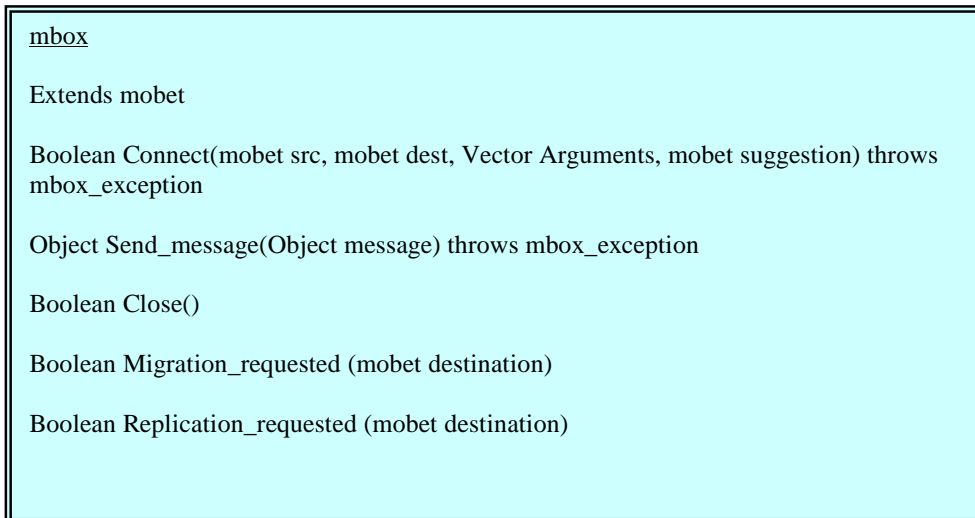


Fig. 6.4 Mbox Interface. Interface Implemented by explicit bindings in UbiqtOS.

bindings are discovered using UbiqDir, can be relocated to other hosts and are notified about changes in the context of the device.

Explicit bindings implement the mbox interface shown in figure 6.4.

Where explicit bindings introduce extra application complexity, which arises from the need to lookup and parameterize the binding, it does not enforce the “best-effort” level of service like an implicit binding, and provides control over the properties of the binding over its lifetime [Leslie91][ODP95]. In a ubiquitous system where properties of the components, and consequently the characteristics of the context of the device, change frequently, the control provided by an explicit binding is most desirable to adapt the binding with the changing characteristics of the system.

The “Connect” method in the mbox interface takes references to two mobile agents and connects them together. This method is used to invoke any signaling/handshake protocol implemented by the binding. Any arguments to the protocol can be passed using the argument vector. If the connection cannot be established then an appropriate exception is thrown to the requesting application. A true value indicates the success of connection establishment, allowing messages to be exchanged between two agents. Therefore, the interface provided by mbox allows a state-full connection-oriented protocol to be

supported to extend (stateless) ACP (described in section 6.7). Chapter 8 shows how the connect method is used to invoke the three-way handshake of TCP.

Further, as processing like encryption and compression performed by a binding needs to be “undone” at the receiving host, such a binding can request SEMAS to replicate its receiving part at the destination host as part of the connection set-up process invoked by a call to connect().

The last argument to the connect() method is only supplied by SEMAS when the agent requests effective mobility from SEMAS, otherwise it is set to null. This argument specifies the name of the least loaded SEMAS in the system and serves as a suggestion to the binding that could request to migrate the requesting agent to another host where it can better satisfy the connection requirements. Effective mobility is described in section 6.7.3.

Once connected, mobile agents can invoke methods on one another using the Send_message method of the binding. The “Send_message” method takes the message to be delivered to the destination specified while connecting. An appropriate exception is thrown if the binding is not connected. Figure 6.5 shows a code fragment illustrating the use of explicit bindings in UbiqtOS.

The binding can interpose any processing in the path of communication by processing the invocation request before calling the Agent Communication Protocol (described in section 6.7.1) to deliver the message.

This allows mechanisms like compression, aggregation, filtering and customization of traffic to be interposed in a connection to suit the characteristics of the underlying network, for example as proposed in the TACC model [Fox97].

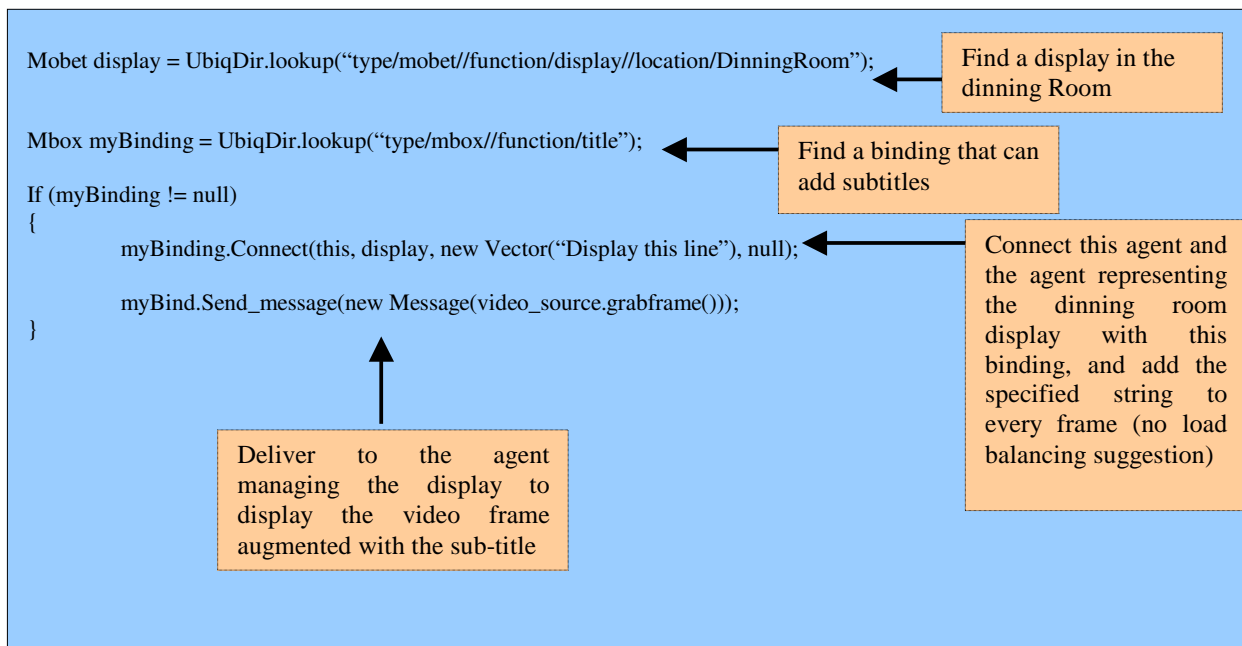


Fig. 6.5 API for Explicit Bindings. The code fragment shows how an explicit binding is looked-up, parameterized and used to add a string title to a video stream

However, the focus of UbiqOS is context-driven adaptation of component bindings to address the mobility and dynamism of the system.

6.6.1 Context-aware Bindings

Chapter 3 outlined the requirement that bindings between components need to adapt as the context of the device changes with resources failing, joining, leaving, or moving in the context of the device. Bindings in UbiqOS, as first-class citizens, implement event handlers for the events offered by UbiqDir to indicate changes in the properties and accessibility of resources making up the device context. Hence, bindings can adapt and/or rebind if a better choice for a resource becomes available or existing resources become unavailable. The definition of “Better” depends on the policy encoded with the binding. For example, chapter 9 shows how a binding was prototyped to provide “follow_me_video” service. This binding rebinds itself to the closet display screen as a

video player is moved from one room to the other to display the video stream on the nearest display.

Optionally, the bindings can be configured to throw an exception to the application whenever notified about changes in the device context, to allow application to perform the adaptation.

6.7 SEMAS: Simple, Extensible Mobile Agent System

The mobile agents are migrated and replicated by the mobile agent system, SEMAS, embedded in UbiqtOS which runs as a thread in JVM. SEMAS presents the interface show in fig. 6.6 to the mobile agents residing with it.

The SEMAS interface allows control over lifecycle management and mobility of agents by exporting methods to bootstrap, move, replicate and destroy agents. Further, it provides methods to list the agents running at a node and to return the corresponding load on the agent engine. Finally, it provides an interface to request effective mobility; this method allows the agents to leave the decision to be relocated to the system.

The `launch()` function is called to bootstrap the specified agent. The agent engine places the agent at the most appropriate agent engine in the system and invokes its “bootstrap” method to start its execution. Application developers develop and compile an agent at a host that has a programming terminal attached to it and then invoke the “`launch()`” method of SEMAS, passing it the agent to be bootstrapped in the system. SEMAS relocates the agent to the most appropriate host in the system and passes the “bootstrap” method, with the specified arguments, to be called at the destination host to initialize the agent in the system. This function returns a true value if the bootstrapping succeeds, whereas an appropriate exception is thrown to indicate that the agent could not be initialized. Figure 6.7 shows the bootstrap of a mobile agent using UbiqtOS.

`Migrate()` and `Replicate()` functions are called by mobile agents to request the agent system to migrate or replicate the specified agent at the specified agent engine. The requesting agent specifies a destination agent with which the agent needs to be collocated and a method to be invoked at the destination host. As SEMAS only provides weak mobility, the specified method is required to resume execution at the destination host. If


```
Boolean Launch (mobet agent, Vector arguments) throws AgentEngineException

Boolean Migrate(mobet requesting_agent, mobet this_agent, mobet destination, Method call_this) throws
AgentEngineException

Boolean Replicate(mobet requesting_agent, mobet this_agent, mobet destination, Method call_this) throws
AgentEngineException

Boolean Kill (mobet requesting_agent, mobet this_agent)

int Current_load()

Vector List_Running_Agents ()

Boolean Do_task(mobet for_this_agent, method task, mobet destination, mbox binding, vector arguments)
throws AgentEngineException
```

Fig. 6.6 SEMAS Interface. The Interface presented by the UbiqtOS Agent Engine to the System Components (mobile agents).

the requesting agent is different from the agent to be transferred, the agent system asks the permission of the specified agent (by calling its migrate()/replicate() method) and the destination system (by using Agent Communication Protocol, as described later), and if they both agree to it, it transfers/duplicates the agent. If the migration cannot be performed due to a network or host failure or because the agent or the destination host did not agree to the transfer then an appropriate exception is thrown to the requesting agent.

A true value is returned to the requesting agent if the transfer succeeds. However, as SEMAS only supports weak mobility, if the requesting agent is the same as the specified agent, the call to migrate and replicate never returns if the transfer succeeds. The execution of the transferred agent proceeds at the destination SEMAS by invoking the specified method.

The destination of a migration or replication request to SEMAS is another agent; SEMAS provides the abstraction of data-centric migrations. Agents only need to know the functionality provided by the destination to be collocated with it, not its location, which could indeed change as agents move around to fulfill their tasks. Viewed differently, this allows intent-based mobility, analogous to intent-based discovery [Winoto99]. Still, an

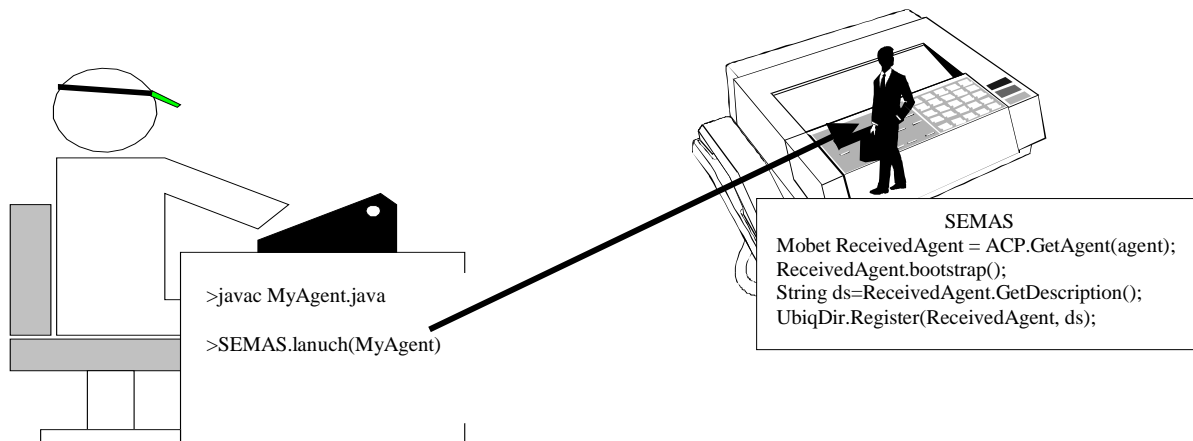


Fig. 6.7 Bootstrap of an agent in Ubiquitous OS.

agent can override this abstraction and request to be moved to a specific host by specifying a fixed component at the host in its migration/replication request i.e. an agent representing the device driver for an embedded hardware resource in the device or the instance of SEMAS or UbiqDir at the destination host.

Further, SEMAS provides an interface to return the current load on the host. The load is returned as the length of the ready queue of the extensible scheduler embedded in Ubiquitous OS as in a POSIX based system.

An agent can also request another agent to commit suicide by invoking the “Kill” method of SEMAS. The requesting agent gives its own reference and a reference to the agent to be killed. The agent engine, in turn, calls the Die() method of the specified mobet and returns true or false depending on whether the agent agrees to commit suicide or not as described in section 6.5.

SEMAS also provides a method to list the mobile agents residing with it. List_running_agents() returns a vector of references to the currently residing agents. These references can be used to invoke methods on the corresponding mobile agents.

Finally, SEMAS provides a method that can be used by mobile agents to request for “effective mobility”. By invoking the method do_task(), the mobile agent requests

SEMAS to perform a specified task and relocate the requesting agent if necessary. The mobile agent engine decides whether the agent needs to be moved to another location to better accomplish the task, finds the best location for it and relocates it before performing the task. Section 6.7.3 describes how SEMAS provides a framework to implement effective mobility to leverage self-organization of the system.

6.7.1 Agent Communication Protocol

The core of SEMAS implements an Agent Communication Protocol (ACP) that is used to transfer agents and messages between different instances of SEMAS.

This protocol provides the basic framework of communication between hosts supporting Ubiquitous. It provides a stateless, unreliable, unordered message delivery protocol that every device needs to support to allow interoperation of resources. It serves to transfer agents, like the aglets Agent Transfer Protocol [Lange98], and, additionally, to exchange messages between hosts. Its simplicity allows it to be embedded even in low-end fat devices. The message format is string based, as opposed to a binary one, to allow flexible processing, with the header fields delimited with special escape characters.

The interface offered by ACP to applications is shown in fig. 6.8. It only has two methods, one for agent transfer and the other for exchange of messages between agents. The return value indicates the success or failure of the transfer request. In the case of a failure, an appropriate exception is thrown to present the cause for failure which could be due to a network failure, destination failure or authentication failure.

The arguments to the methods exported by ACP are used to fill in the fields for the two frame types used by ACP (shown in table 6.1), as described below.

The first field in the protocol header authenticates the source SEMAS. The agent engine receiving the message looks at this header to decide whether to execute the agent transferred, process the message received, or if the source cannot be trusted, throw away the frame.

The second field tells whether the payload encapsulated in the frame is a mobile agent or a message for a mobile agent. The next field gives the method to be called for a mobile agent to resume its execution or the name of the destination agent if the ACP frame

```

boolean agent_transfer(moblet agent, moblet destination, string Method_to_be_called, Vector
arguments_list) throws ACPException

boolean message_transfer(Message message, moblet destination, moblet source) throws ACPException

```

Fig. 6.8 ACP Interface. Communication API in UbiqOS

ATP Authentication header	Agent Message		[Method to be called] [Destination agent]	[Arguments] [Source Agent]	Payload	Checksum
---------------------------------	------------------	--	--	------------------------------------	---------	----------

Table 6.1 ACP frame format. Lingua Franca for inter-UbiqOS interaction.

carries a message. The fourth field gives the arguments to be used for method invocation for the former and the name of the source agent for the later.

The fifth field holds the payload, which could be either an agent or a message, as serialized Java objects, as specified by the frame type. The last field of the ACP frame gives a checksum over the whole frame to ensure the integrity of the message.

On the receiving side, SEMAS receives the ACP frame, authenticates the sending SEMAS (as described below) and determines the frame type by looking at the second field in the header. Agents received from a trusted source are executed by calling the method specified in the third field with the arguments specified in the fourth field of the ACP header. However, if the ACP frame encapsulates a message for an agent, it requires extra processing.

To allow agents to receive messages, ACP, like UbiqDir and SEMAS, offers an event using the Romvets architecture embedded in UbiqOS. This event is generated whenever ACP receives a message. Agents, rather agent bindings, subscribe to this event, parameterized with the handle of the receiving agent. Whenever ACP receives a message, it forwards it to the Romvets interface, which routes it, as event notifications, to the agents that had subscribed to this event. The handler of the event could be an

application mobet or a protocol layer mobet. In the case of the later, it is the responsibility of the protocol-layer to pass the message higher-up in the stack by generating an appropriate event. If there is no event handler for a message, the message is simply thrown away.

An important thing to note in this scheme is the lack of transport layer addressing; ACP frames are addressed to agents by name, and not to a Service Access Point at the transport layer e.g. a UDP port number. Therefore communication in UbiqOS is network independent i.e. ACP works independent of which protocol stack is dynamically deployed to transport the ACP frames over the network.

Where this stateless operation of ACP makes it simple, events generated by ACP can be used to support protocols for reliability, ordering, flow-control and fragmentation according to the characteristics of the system and application requirements. In addition to the “message_received (destination mobet, mobet source, Message message)” event described above, ACP also generates an event when requested to transmit a frame. This “Pkt_transmit(ACP_Frame this, String Address_type, String destination_address)” event specifies the ACP frame to be transmitted along with the address of the destination. The destination address specifies the network specific address, and its address family (like IP), of the destination agent, registered as part of its XML description. ACP resolves this address just-in-time, by looking up the agent description in UbiqDir (described in chapter 7), before transmitting the event to request transmission. This event is subscribed to by network protocols that encapsulate the ACP frame and send it on the wire before returning a boolean value to indicate success or failure. If none of the protocols had registered interest in an event, or the transmission fails, then the request to transfer the agent or the message returns with an appropriate exception.

As the events are parameterized with the destination address and its type, the protocols can subscribe interest at the granularity of a specific destination. More usefully, protocols can handle events containing the address type supported by them. So, for instance,

Sender Private Key(Message, length, from, timestamp)
--

Fig. 6.9 ACP Authentication Header.

messages meant for destinations running an IP protocol, indicated by IP protocol type, can be handled by IP protocol layer to transmit packets.

Conversely, ACP itself handles an event, “packet_arrived(ACP_frame this)”, that is used by networking protocols to deliver a frame to ACP on the receiving side. ACP, in turn, generates the above mentioned, ”message_received(mobex destination, mobex source, Object message)” event if the message contains a message to be delivered to an agent running in SEMAS.

Chapter 8 shows how this scheme is used to implement extensible protocols stacks to suit the characteristics of an active space and the application requirements.

Finally, ACP authenticates the source SEMAS before entertaining a frame, to protect against malicious senders. The source SEMAS adds an authentication header to the ACP frame, shown in figure 6.9, that is used by the receiving SEMAS to ensure that it only entertains frames from trusted sources.

The authentication header specifies the message length, the name of the sender and the time of sending, signed by the sender’s private key. The private key authenticates the sender, the length field ensures the integrity of the message while the timestamp protects against replay attacks [Neuman94].

However, for this scheme to work, the receiving host needs to know the public key of the sender a priori. This key could be burnt in the ROM of the device along with the image of Ubiquitous OS. While this could raise a scalability issue, as a device needs to store the public keys of all the devices it intends to communicate with in its lifetime, we believe that public keys would be manufacturer specific rather than device specific. Therefore, the device would only need to know the public keys of its own manufacturer and those of other compliant manufacturers with which it intends to interoperate. For example, a device manufactured by SONY™ would store the public key of SONY™ (along with its private key of course) and public key of other manufacturers SONY™ can trust, maybe

per say, Toshiba™ and Panasonic™. This scheme guarantees that an ACP frame coming from another SEMAS can be authenticated to safely process the message or execute the encapsulated agent.

6.7.2 Extensibility

The section above presented a brief overview of the interface presented by SEMAS to the mobile agents, but did not cover the details of how the mobile agent engine decides the right location for the agent while bootstrapping or when providing effective mobility. Likewise, no details were given about how the migration or replication is actually performed by the agent engine.

These details, in lieu with the underlying theme of the thesis, depend on the complexity a device can afford, characteristics of the current active space of a device and the application requirements. Therefore policies and protocols for migration and replication need to match the device capabilities and need to adapt as the characteristics of the surrounding active space change or as the device is moved to another active space with disparate characteristics.

SEMAS supports adaptation using the same mechanism as every other component in UbiqtOS: SEMAS offers events corresponding to its functional interface and extensions subscribe to these events to be up-called to implement the distributed operation. SEMAS uses the Romvets subscribe/notify interface to export events and Romvets re-routes them to agents subscribing interest in them by up-calling their appropriate methods.

SEMAS offers 7 events that are used to implement effective mobility and provide context-specific protocols and policies for load-balancing, fault-tolerance, high-availability. Boolean Launch(moblet agent, string method_to_be_called), Boolean Migrate(moblet requesting_agent, moblet destination, string method_to_be_called, Vector arguments) and Boolean Replicate(moblet requesting_agent, moblet destination, string method_to_be_called, Vector arguments) events, corresponding to the three methods in the functional interface of SEMAS, are generated by the sending SEMAS when an agent invokes the corresponding method to request bootstrap, migration and replication respectively.

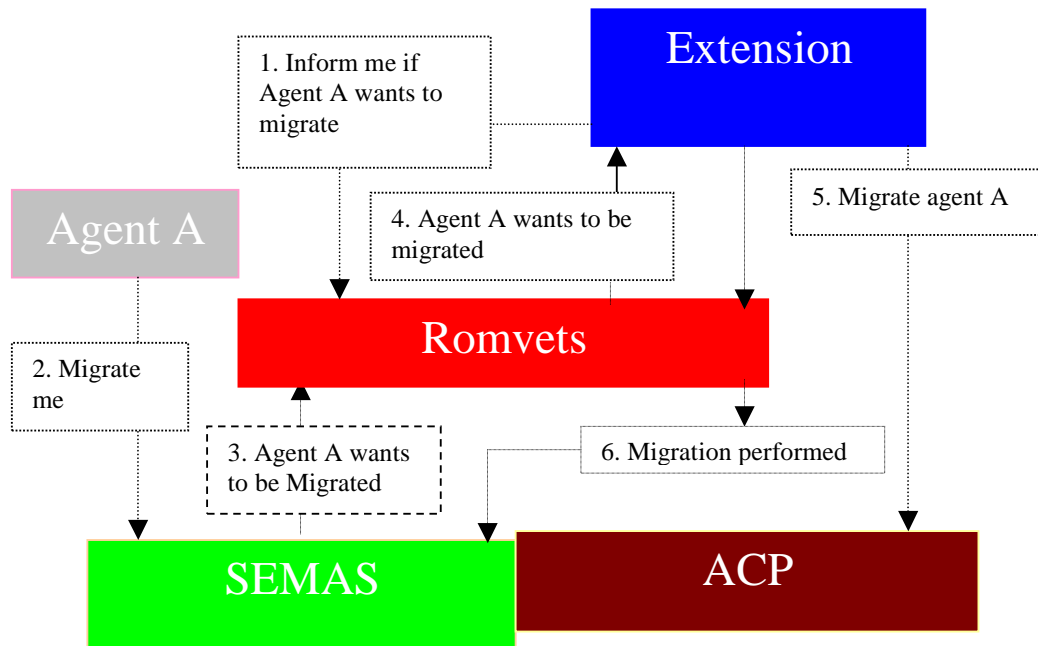


Fig. 6.10 Extensibility in SEMAS. Extensions Interpose functionality between SEMAS and ACP

Context-specific extensions to SEMAS, which are themselves mobile agents, subscribe to these events, by calling the subscribe method of Romvets, and are notified whenever the corresponding operation is invoked to launch, migrate or replicate the specified agent. These extensions perform their respective operations on the agent and call ACP to transfer the agent once they are done. Likewise, corresponding events are generated by the receiving SEMAS whenever an agent is migrated or replicated on it to allow extensions to process the incoming agents. Hence, these extensions allow functionality to be interposed between SEMAS and ACP (as shown in Fig. 6.10). The events offered by SEMAS are parameterized with the agent name requesting the operation, allowing extensions to tailor policies for individual agents. Therefore, different policies can be deployed for different agents according to the requirements of the application encoded by the agent.

It is worth noting that “migrate” and “replicate” events are similar and lead to the same call to ACP, by context-specific extensions, to transfer the agent. The reason to

distinguish between them is to provide flexibility and efficiency by allowing different policies to be supported for migration and replication of agents.

The next section shows how these events can be used to deploy extensions that implement protocols and policies for effective mobility, load-balancing, fault-tolerance, disconnected operation and reliability according to application requirements and system characteristics.

6.7.3 Effective Mobility

The scenario presented in the first chapter led to the requirement that a ubiquitous system should allow software to be injected in the system and placed where it can best accomplish its task.

Applications are injected as mobile agents in the system, and UbiqtOS needs to be able to place them where they can best accomplish their task. This goal has, indeed, been researched by several distributed operating systems like Amoeba [Tanenbaum90], Sprite [Dauglis91] and Mach [Accetta86] to name a few, and several load-balancing algorithms have been proposed to optimize the placement of communicating processes in distributed and parallel systems [Sokol91]. However, as pointed out in [Tanenbaum95], efficacy of a load-balancing policy depends on the characteristics of the system and the requirements of the application. Hence, any one algorithm cannot be expected to satisfy the diverse system characteristics and applications requirements in a ubiquitous system. The most suitable host to relocate the process could be the least-loaded one, the one that minimizes network traffic for the process, or the most reliable host etc., depending on the system characteristics and application requirements. Further, unlike a traditional parallel system, where all communicating processes are known a priori, agents in a ubiquitous system need to dynamically discover and invoke one another. Hence, balancing load for new connections can adversely effect the existing connections with other agents.

UbiqtOS addresses these issues by combining two approaches; context-specific load-balancing and application-specific connection management. Below we describe these two in more detail and illustrate how they provide a framework to support effective mobility in the system. Where proactive mobility allows agents to move from one host to another

to carry out their pre-programmed agendas and reactive mobility allows them to be moved in response to the changes in the context of the device, effective mobility makes the system self-organizing.

In order to provide effective mobility, SEMAS requires that tasks be structured as functional units i.e. as subroutines that perform a function while being connected to another agent. This is analogous to structuring of tasks as mobility units in mobile agent systems that support weak mobility, like Concordia [Wong97], that require that one subroutine be nominated to be called at each hop of the agent.

Mobile agents request effective mobility by calling the `doTask` method of SEMAS (refer to figure 6.6). The requesting agent passes SEMAS the name of the method to be invoked to carry out the task, the destination agent that needs to be contacted to carry out the task, and the binding to use to make the connection.

Once invoked with the `Do_task` method, SEMAS finds the binding in `UbiqDir`, instantiates it and calls its `connect()` method with the arguments provided. It then finds the destination agent and passes its references to the instance of the binding to allow the binding to pass messages to it. Additionally, SEMAS fills in the last argument of the `connect()` method with a handle for the SEMAS that it deems most suitable for the agent to be located to perform its task. It is up to the binding to take this suggestion into the account by asking SEMAS to migrate the requesting agent to the suggested host. The suggested host is selected by context load-balancing policies and the migration is only performed if all the other bindings of the agent agree to it. Finally, SEMAS invokes the specified method of the source agent to start the task. Below we describe these operations in more detail.

6.7.3.1 Extensible Load-balancing

When invoked with the `Do_task()` methods, SEMAS generates an event `mobet Balance_load(mobet src, mobet dst)`, using `Romvets`, that is handled by load-balancing extensions. These extensions talk to their peers on other hosts and return a handle of the SEMAS that they think is the most appropriate to locate the requesting agent. Several load-balancing policies can co-exist, each returning the host that optimizes a specific

aspect e.g. least loaded host, most reliable host, the host which would incur least network traffic between the source and the destination agents etc. Romvets collects all the returned values and passes them on to SEMAS. SEMAS applies a simple counting scheme, and chooses the host with the most number of votes. If there is only one occurrence of each entry, then the first entry in the vector is selected. This simple scheme allows the selection of the host that optimizes the most number of metrics, but relies on the fact that the first extension installed by the active space is the most important metric for the system in the case when each metric is optimized by a different host.

As the `load-balance()` event is parameterized with the source and destination agents, the extensions can tailor their policy on a per-agent basis; different load-balancing policies can be supported depending on the application requirements of the requesting agent.

In the simplest scheme, the extension handling the `balance-load()` event selects the agent at the least loaded host in the system. Extensions use the “`current_load()`” method offered by SEMAS to measure load at a host and use a group communication protocol on top of ACP to exchange this information to decide on the least loaded server. The extensions can use a passive scheme to exchange the load, where idle processors ask their neighbors for work, or an active scheme, where tasks are distributed by their creating processor, depending on the number of processors in an active space registered with UbiqDir. Passive schemes have been shown to be more efficient for medium to small scale systems, typically less than 25 processors, whereas active schemes are known to effectively reduce the time to select an appropriate host in larger systems [Kouichi91].

Further, extensions can present the load as an exponentially weighed moving average, instead of a snapshot of the length of ready queue, to ensure long-term fairness in load-distribution in the system. Similarly, simpler algorithms that only interpret load information as discrete steps, like high, medium, low, can be supported if devices in an active space cannot support complicated algorithms.

Likewise, load balancing policy used by the extensions could be local, where only the local load is compared with the destination, or global where load on all the processors in the system is compared to find the least loaded host to relocate an agent. Where global policies lead to more informed decisions they require interaction between all the hosts in the system and, hence, can only be supported efficiently on networks with multicast

support like HomePNA. Local policies reduce network traffic and can be supported on point-to-point link layers like Warren.

Finally, extensions can use algorithms to optimize other criteria in addition of balancing load in the system. Agents that require network communication could be handled by extensions that place them at hosts with high-bandwidth/low-latency network connections. Similarly, agents providing services that need to be highly-available can be handled by extensions that place them at faster and reliable servers in the active space, as indicated by their descriptions in UbiqDir. Reliability of a server can be judged by the number of timeouts of its description in the local UbiqDir due to lack of renewal of its soft-state entry.

SEMAS selects the host with the most number of entries in the vector returned by Romvets and passes that as the last argument to the connect() method of the binding specified for the connection. The binding takes this suggestion into account when making the connection with the specified destination, as described below.

6.7.3.2 Application-specific connection management

Where load-balancing extensions find the best location for the agent, only the agent binding can decide whether migrating to another host would be beneficial for the new connection. Therefore the system finds the best location for the agent but instead of transparently relocating the agent at the selected host, as in traditional distributed operating systems, it only uses that as a hint to the agent binding. It is up to the agent binding to make use of this hint by requesting migration to the suggested host. However, the binding can choose not to request migration or even request migration to another host. The binding requests the migration by calling the migrate() function of SEMAS, just like a mobet, and specifies the requesting agent as the source agent and its own connect() method as the method to be invoked at the destination SEMAS. Therefore the connect() method of a mbox supporting effective mobility needs to keep some state to detect that its connect method has been entered again, indicating that it has been relocated to a new host.

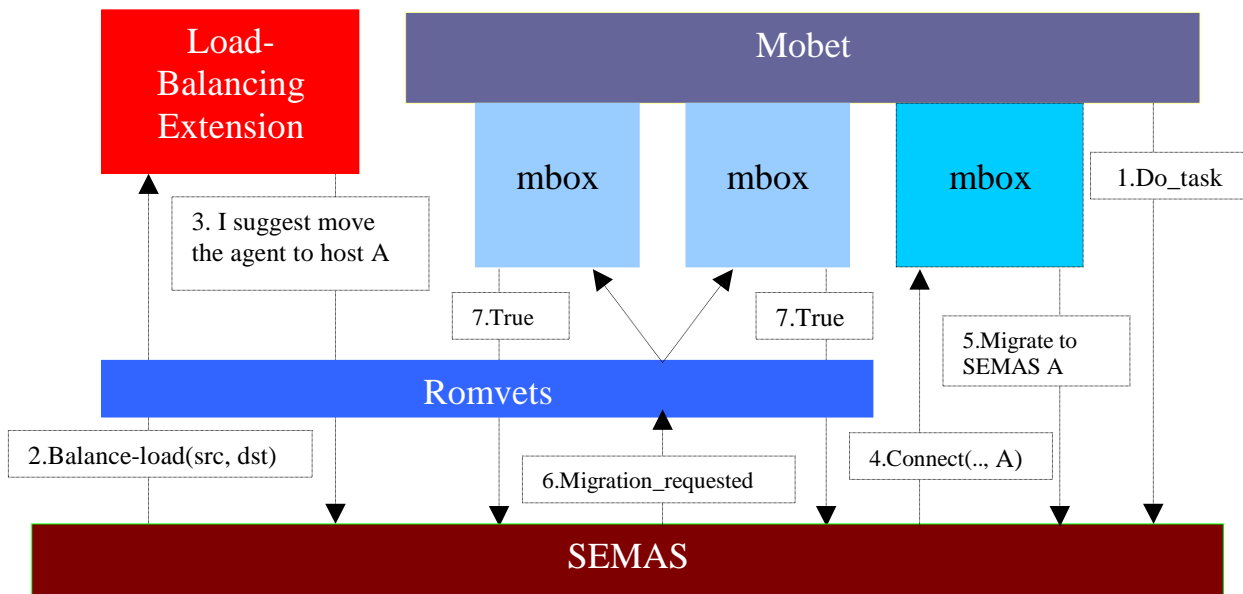


Fig. 6.11 Effective mobility in SEMAS:

When invoked with the *Do_task* interface, SEMAS upcalls context-specific load-balancing extensions and uses the returned least-loaded host as a suggestion for the newly created binding. The binding, in turn, chooses to migrate the requesting agent to the suggested host and calls the migrate function of SEMAS. SEMAS seeks permission of all the existing bindings by generating the *Migration_requested()* event and performs the migration if all of the agent bindings return a true value.

Where the above scheme provides a delicate balance between application requirements and system characteristics in a dynamically changing environment, the migration requested by the new connection can adversely effect existing connections of the agent. This is addressed as follows.

When the agent binding calls the “migrate” function, the agent engine sends a “migration_requested (destination agent)” event to all the other active bindings associated with the agent, acquired by calling its *Get-bindings()* method. When invoked with the *migration_requested()* event, the bindings can either adapt their connection in light of the new migration and return a true value or, if any binding decides that its connection requirements would be violated by the migration needed by the new connection, veto the migration by returning a false value. SEMAS takes a logical AND of the all the returned values and the requesting binding is notified about the decision. Hence, even if one of the

bindings decides that its connection requirements are violated by a migration, the agent is not migrated to the specified host.

Moreover, if the destination agent is not accessible, or the binding specified for the task decides that its requirements cannot be met in the current context, then an appropriate exception is thrown to the agent attempting to make the connection and the migration is not performed. It is up to the requesting agent to recover from it, for example, by changing the binding to a less demanding one or by postponing or skipping the task all together.

Assuming that the bindings used by an agent cooperate with the system and with each other, this framework allows application semantics to be combined with context-specific characteristics to provide effective mobility. Mobility is application specific and used only when it is effective to leverage a self-organizing system. Figure 6.11 shows how SEMAS supports effective mobility.

The other primitive offered by the agent engine is “`replicate(destination agent, method to_be_called)`”. This allows agents to replicate themselves at the node hosting the indicated agent. The agent is replicated and the specified method is called at the destination host. We use this facility for two purposes. 1) To install new components with UbiqtOS joining an active space. The indicated method unregisters the component to be replaced, registers the new component and sits there waiting to be called. 2) To deploy sensor agents. The indicated method is a repetitive task that continuously monitors the system. For example, beacon-sensing agents deployed to sense and select the best network interface for a mobile host in a wireless overlay network. Likewise, “`replication_requested(destination agent)`” event is sent to all the active bindings of the agent requesting to be replicated. The boolean value returned by the binding is taken to be its consent to be replicated at new host or not. A false value stops the agent engine from replicating the binding at the new host. This allows host-specific bindings from not being replicated at the destination host.

The agent can choose to leave the choice of an appropriate binding to UbiqtOS, which defaults to an appropriate context-specific binding. This is indicated by specifying a null binding.

Apart from leveraging effective mobility, extensibility in SEMAS is used to support protocols for context-specific bootstrapping, disconnected operation and reliability, as described below.

6.7.4 Bootstrap Load-Balancing

Mobile agents are bootstrapped in UbiqOS by calling the “launch()” method of SEMAS. SEMAS relocates the agent to the most appropriate host in the system and specifies the “bootstrap” method of the agent to be called at the destination host to initialize the agent in the system.

This scheme of relocating a process to another host, at startup time, has been proposed in distributed operating systems like Ameoba, Mach, and Sprite to provide for balancing of load in the system. However, as pointed out above, efficacy of a load-balancing policy depends on the characteristics of the system and the requirements of the application.

Therefore, SEMAS allows bootstrap load-balancing policies to be deployed as context-specific extensions, as well. Bootstrap load-balancing policies are deployed as event handlers for the launch() event generated by SEMAS when requested to bootstrap an agent.

A user introduces a mobile agent in the system by calling the “launch” method of SEMAS and passing the agent to be launched as its argument. SEMAS, in turn, generates a “launch(moblet agent)” event using the Romvets architecture. Extensions subscribe to this event to be notified when the specified agent requests to be launched in the system and then return the handle of the SEMAS they deem most suitable to relocate the agent. Romvets collects the values returned by the extensions in a vector and passes them on to SEMAS. SEMAS migrates the agent to the host with the most number of entries in the vector, specifying its bootstrap() method to be called at the destination SEMAS to bootstrap the agent in the system.

6.7.5 Disconnected Operation

A major benefit of using mobile agents is their ability to perform remote functions even in the face of network disconnection [Lange98]. However, in architectures that employ

RPC to transfer agents [Lange98][Wong97], migrations can only be performed if the destination host is running and the network link to it is working at the time when the migration is requested. Otherwise, either the migration is denied or the agent engine is blocked until the agent migration can be carried out. This limits migrations and introduces inefficiencies for active spaces where network links fail and hosts go down often.

The events offered by SEMAS can be used to address this problem by deploying extensions that facilitate disconnected operation according to the characteristics of an active space. The extensions deployed to allow for disconnected operation implement a FIFO and subscribe to the “migrate” and “replicate” events generated by SEMAS. When notified with the request of an agent to migrate/replicate it on another host, the extension places the agent request in its queue. Transfer requests queued in the FIFO are tried one by one by attempting a transfer using ACP. If ACP cannot find a route to the destination, indicated by failure exception, the extension tries the next agent and then the next, transferring agents in a round-robin manner. Therefore, the agents waiting in the queue can be served whenever there is a connection available to the destination host without holding up other agents or requiring a re-try by the application itself.

The extension can use any policy to serve its FIFO according to the characteristics of the network links. For instance, agents for the same destination can be scheduled together to avoid unnecessary retries, the time between attempts for queued agents can be adjusted to suit the dynamism of the system etc.

6.7.6 Reliability

The dynamism of a ubiquitous system means that some of the agents can get lost as communication links fail, or devices move, disconnect or malfunction. However, if a mobile agent holds any important information or possesses unique functionality, it is important not to lose the mobile agent. Moreover, the mechanisms to provide this reliability depend on the device capability, the application requirements, degree of dynamism in the system and the characteristics of a particular active space.

Therefore, the extensibility in SEMAS can be used to provide reliability according to device capabilities and requirements of a particular active space. We have used the events offered by SEMAS to support three schemes to protect important agents from getting lost in the system.

The first scheme provides a simple mechanism to recover an agent lost due to remote host or network failure, but only provides at-least-once semantics, suitable for idempotent tasks. The second scheme extends the first one to alleviate the source host from the complexity of maintaining reliability, making it suitable for fault-prone limited capability devices. The third scheme guarantees exactly-once transfer semantics application with stringent requirements but can only be supported in hosts that can afford the complexity of transaction processing.

In the first scheme, the extension subscribes to the “migrate/replicate” event of SEMAS to be notified whenever a specified agent requests to be migrated/replicated to another host. Once notified, it first makes a persistent copy of the agent in the local memory of the device before calling ACP to migrate the agent to the destination host.

This copy serves as the checkpoint of the agent that can be used to recover it, in case the agent is lost due to network or destination host failure. The events offered by SEMAS are used for failure detection as described below.

The corresponding extension running at the destination host subscribes to the “migrated/replicated” event to be notified when a specified agent is received by the destination SEMAS. On the receipt of an agent, it sends an acknowledgement, encapsulated in an ACP frame, to the source extension to notify that the agent has reached safely. If an acknowledgement is not received by the source agent for some time T , it retransmits the agent. T is computed by the source extension by periodically sending “echo” messages to the destination extension, and taking a moving average of the round-trip time.

This simple scheme ensures reliable network transfer on top of ACP.

Further, the destination extension subscribes to the “migrate/replicate” event for the same agent to be notified when the agent requests to be migrated to another host. Hence, the “migrated/replicated” and “migrate/replicate” events for an agent implicitly serve to mark the time for which the mobile agent resides at a given host.

The destination extension uses this information to generate a heartbeat for the source host while the transferred agent resides with its host. If, after migration, the source extension does not receive the heartbeat for some time K , it infers that the destination host either disconnected or stopped working. If the destination host disconnects, the heartbeat would resume when it reconnects and nothing needs to be done. If the destination host fails then the transferred agent could be assumed to be lost with it, and the extension uses its checkpoint to replicate the agent again, using ACP, when the destination host comes back up. This approach ensures that the agent is executed at-least-once at the destination host but requires the source host to be running and accessible to provide this reliability.

This second approach, on the other hand, is suitable when the source host cannot be guaranteed to be accessible or running at all times e.g. battery operated mobile devices.

In this approach, the mobile agent that handles the “migrate/replicate” event makes a remote copy of the agent, in a backup server, before migrating/replicating the agent to the specified destination. A highly-available server needs to be selected for this purpose and deployed with an extension that can receive a mobile agent and a destination SEMAS to provide reliability as described above. This alleviates the source host from maintaining reliability and hands over the responsibility to the backup server. Moreover, this scheme can be extended to achieve a greater degree of resilience by replicating the migrating mobile agent to more than one backup server before migrating it to the specified destination. Where this scheme introduces additional agent replications and redundant heartbeat traffic, it does not require any single host to be running all the time to ensure reliable transfer of mobile agents, making it suitable for highly dynamic settings like an ad-hoc network .

The first two schemes, however, only ensure at-least-once reliability for agent transfer. The tasks that require exactly-once transfer semantics can be implemented using transactional queues proposed in [Rothermel98]. The extensions in this scheme subscribe to the “migrate/replicate” and “migrated/replicated” events generated by SEMAS and additionally implement a persistent log of agent transfers. Once notified to migrate/replicate an agent, the source extension makes a local copy, requests ACP to migrate the agent and atomically appends an entry in its log that an agent migration is in progress. The extension then blocks instead of returning a boolean value after transferring

the agent. On the receiver side, once notified, the corresponding extension makes a persistent copy of the agent and makes an atomic append to its log indicating that an agent migration is in progress. The extension, however, does not send a commit acknowledgment to the source extension. The transaction is only committed when the visiting agent requests a migration to the next hop. Based on the assumption proposed in [Rothermel98] that the agent performs its task at one host and then moves to the next one, or back to the originating host, “migrate/replicate” event on the receiver side is a sufficient condition for the receiver extension that the migrating agent has successfully performed its task. Hence, when the receiving extension is notified with the “migrate/replicate” event for the same agent, it sends a commit message to the sending extension, atomically removes the entry from its log and discards its local copy of the agent. Likewise, the sending agent on the reception of a commit message discards the local copy of the agent, removes the corresponding entry from its log and returns with a true value, indicating that the agent has been successfully executed at the specified host. If the transaction fails then a false value is returned to the requesting agent, and an appropriate exception is thrown to the requesting agent.

6.8 Summary

This chapter described how SEMAS, embedded at layer 1 in Ubiquitous OS, allows context-specific software to be deployed with an instance of Ubiquitous OS to enable a device to control, manage and program other resources in its context.

Keeping with the underlying theme of the thesis, distributed operation in SEMAS lends itself to context-driven adaptation using upcalls routed through Romvets. Extensibility in SEMAS is demonstrated by giving examples of protocols and policies to provide effective mobility, bootstrap load-balancing, disconnected-operation and reliability according to application requirements and the characteristics of the context of the device.

In addition to traditional proactive mobility, SEMAS provides reactive mobility, allowing agents to be notified and moved in response to the changes in the context of the device to address the dynamism of the system. Finally, this chapter proposes a new scheme for mobility, called effective mobility, provided by the use of explicit bindings and

extensibility in SEMAS, to leverage a self-organizing distributed system. Effective mobility allows mobility of components only if it can positively affect the system according to application requirements and the characteristics of the current context of the device.

“Why did you name your son John? Every Tom, Dick and Harry is called John”

Anonymous

Chapter 7

Extensibility, Dynamism and Context-awareness in UbiqDir: An XML-based Directory Service for Ubiquitous Systems

This chapter describes the architecture of the extensible registry, UbiqDir, embedded at layer 1 in UbiqtOS. UbiqDir is a yellow-pages distributed directory service. Components are installed with an instance of UbiqtOS by registering their XML descriptions with the corresponding instance of UbiqDir and it exports these components to other instances of UbiqDir in its context. Conversely, UbiqDir serves to capture and export the changes in the device context, to applications and services residing with it, to allow context-aware adaptation.

UbiqDir serves to extend and adapt the functionality of UbiqtOS as component references and their XML descriptions are registered, deleted or upgraded with it. UbiqDir uses the XML meta-data to allow introspection of the properties of the components registered with an instance of UbiqtOS, and hence, UbiqDir's interface serves as a meta-interface (c.f. MetaOS) to install, upgrade and delete functionality from UbiqtOS.

UbiqDir follows the design of UbiqtOS; a bare minimum extensible core adapted by context-specific extensions deployed as mobile agents. The core of the directory service comprises a simple soft-state based lookup service (integrated with the subscribe/notify Romvets architecture) which generates synchronous events to request distributed operations. The event handlers for these events implement the distributed operation of the directory service by exporting the descriptions registered with one instance of UbiqDir to other instances of UbiqDir in its context using protocols and policies most suitable for a particular context. This is achieved by the extensions forming a context-specific overlay to disseminate and lookup the descriptions of resources in the system.

As UbiqOS only supports one component type (mobet) and all mobets are registered, deleted and looked-up with an instance of UbiqOS using UbiqDir, it not only serves as a directory service to locate things in the system, it provides an orthogonal persistent store [Jordan98] for software components installed with an instance of UbiqOS. Hence, services (written as Java agents) in the system exist for as long as they are registered with an instance of UbiqDir, and not as long as they are being used (as would be case without UbiqDir).

The rest of the chapter focuses on the use of UbiqDir as an attribute-based directory service and its use of XML to encode meta-data to describe attributes of the components registered with it. It describes how the events generated by UbiqDir are used to support context-specific distributed operation and to leverage context-driven adaptation of applications and services in UbiqOS.

7.1 Motivation

To interoperate with their environment, resources first need to find other resources in their context. Users can move from one active space to another and expect the system to provide ubiquitous access to appropriate services. In particular, applications need to be able to find the “best” available services that can satisfy their requirements at any point of time in the system.

The problem, of course, is simple. “Find the best service that matches the given requirements”. Although this might appear to be a traditional service location issue in distributed systems, the heterogeneity and mobility of resources and dynamism introduced by resources moving, joining and leaving the system introduce new challenges in resource discovery [Banavar00][Czerwinski99].

First, service discovery needs to be central rather than external to the system, as no service can be assumed to be available at all times to allow its reference hardwired in applications. Second, the meaning of “best” is no longer static, like in traditional systems, instead it needs to address the dynamism and context-awareness inherent in the system. Delineation and interpretation of the attributes of a resource might vary as a resource is

moved from one active space to the other, or as new resources and services are introduced in a system. Third, the directory service needs to support a wide range of resources, with arbitrary numbers and types of attributes. Further, the resource description format needs to be human understandable yet amenable to machine processing to allow users and other devices to be able to discover other resources. Fourth, the design of the directory service should not enforce any rigid policies that might introduce inefficiencies or hinder interoperability as a resource is moved between different active spaces with disparate characteristics. Finally, the core of the directory service needs to be simple enough to be accommodated in limited capability devices, while allowing mechanisms to scale to more privileged devices and to adapt to changing contexts.

We address these challenges by designing an extensible XML-based directory service. The core of the directory service only implements a minimum local operation and allows distributed operation mechanisms to be deployed dynamically to suit the characteristics of the current context of the resource. The core generates a corresponding event whenever a resource description is registered, deleted or updated with it. Context-specific extensions, deployed as Java mobile agents, subscribe to these events offered by the core to implement mechanisms for discovery, replication, caching, consistency, partitioning, load-balancing and fault-tolerance. Mobile agents are also used to evaluate on-demand the attributes of a resource that change dynamically. Further, applications can subscribe to the events offered by the directory core to be instantly notified about the changes in the resource context to leverage application adaptation warranted by system dynamism. Finally, these events can be used by AutoHAN event scripts written in Cambridge Event Language [Bacon00] to automate the operation of an active space.

7.2 Contributions made by UbiqDir

While distributed directory services like INS [Winoto99], SSDS [Czerwinski99] and Chord [Stoica01] all support attribute-based resource discovery, they enforce fixed policies for the distributed operation of the directory service and, hence, fail to effectively address the dynamism and context-awareness inherent in a ubiquitous system. UbiqDir,

on the other hand, allows its distributed operation to be adapted according to characteristics of the current context of the device, using upcalls routed by Romvets. The unification of a lookup service and a subscribe/notify events interface also bridges the gap between service discovery [Winoto99][Czerwinski99] and context-aware application adaptation [Esler99][Grimm00][Banavar00]. Applications can subscribe to the events offered by the directory service to be notified about changes in the resource context to guide timely adaptation.

This chapter contains the following:

- It presents an analysis of a naming scheme befitting a ubiquitous system and highlight three unique characteristics of a ubiquitous name: intent, dynamism and context-awareness.
- It shows how these requirements warrant an adaptable system design and present the design of an extensible directory service, UbiqDir. It shows how this architecture addresses the challenges posed by a ubiquitous system and describes how mobile agents are used to deploy context-specific mechanisms for discovery, caching, replication, partitioning, load-balancing and fault-tolerance.
- It proposes a lazy evaluation scheme to compute dynamic attributes on the fly, using mobile agents, and presents an adaptive lease-based scheme to implement soft-state. These two schemes address the dynamism of the system while effectively reducing the overhead of periodic refreshes in a soft-state based system.
- Finally, it shows how this architecture and our naming scheme efficiently address the ubiquity of the system, still keeping the core service simple enough to be accommodated in impoverished devices.

7.3 Ubiquitous Names and Resolution

Experience with AutoHAN [Saif01] showed that typical queries in a ubiquitous system are as follows. “Show this video stream on the largest display with the least access latency”, “use the most reliable and least loaded server that is not in the children’s room to run intrusion detection software”, “find a surveillance camera that is either in the porch

or on the entrance door and has snap shot frequency greater than 10 shots /sec” etc. In general, a resource lookup query in a ubiquitous system is of the form “Find me the “best” resource that provides service X”.

Mobility, dynamism and heterogeneity in a ubiquitous system introduce new requirements to express and resolve the definition of “best” and the delineation of “service X”.

First, applications in a ubiquitous system often do not know the location [Winoto99] or the precise functionality of a service [Hodes97] that could best satisfy their requirements. Hence, resources can only give an “intent” about the resource that can best meet their requirements. e.g. “Find me the best display”, could be satisfied by “the nearest TV”.

Second, the description of a resource is not static as in traditional systems, but includes dynamic attributes as well. For example, the link latency or reliability of a resource accessible by a mobile client changes as the client moves in the environment. Likewise attributes like “least-loaded server” change with time and the definition of the “most reliable server” changes as resources fail, leave, enter or move in the system. This introduces “dynamism” in a ubiquitous name, which means that certain attributes of a resource are more useful when calculated on the fly.

Fourth, resource description in a ubiquitous system includes non-functional attributes in addition to the functional interface descriptions. These attributes are acquired by the resource due to its context in the system and could include information like location, quality of service supported by the underlying network, security information etc. These attributes are usually not known to the resource itself at development time, and are handled by the system to achieve better selection of appropriate resources. Hence, ubiquitous names are context-aware.

Due to the dynamism and mobility of resources, lookups in a ubiquitous system are more useful when performed relative to the requesting resource and not a directory service running in a remote server. Additionally, most of the lookups in a ubiquitous system tend to include context-aware superlative adjectives e.g. “nearest display”, “least-loaded server” etc. Consequently, superlative adjectives need to be supported and resolved in a given context, relative to the requesting resource.

Finally, queries involving more than one criterion need support for logical (AND, OR, NOT) and relational (Less than, Greater than) operators to allow composition of composite queries [Saif01].

7.4 Design requirements

A directory service should embody an information model, a functional model, a distributed-operation model and a security model.

7.4.1 Requirements for Information Model

The information model of the directory service specifies the structure and representation of how data is stored in the directory service, thus, defining the name space. This usually determines the syntactic structure of the queries as well e.g. relational, hierarchical etc.

The information model of the directory service for a ubiquitous system should aim to provide maximum flexibility to accommodate resources of arbitrary number and type of attributes without compromising efficiency for lookups. Second, as ubiquitous systems primarily interact with humans, the structure and representation of the data should be human readable. Finally, the structure of the stored information should support all three characteristics needed to meaningfully describe a resource in a ubiquitous system, including dynamic attributes and lookups guided by only an intent (partial description) of the service required.

UbiqDir uses XML to support a string-based, flexible and human-readable information model.

7.4.2 Requirements for Functional Model

The functional model specifies the interface offered by the directory service to access and modify its state. It is usually reflected in the directory access protocol as well e.g. LDAP [X.500].

The functional interface of the directory service allows entities to register their descriptions and to lookup other entities that best satisfy their requirements. Due to the heterogeneity of a ubiquitous system, the lookup interface should not enforce any policy of resource selection in case of multiple matches. Rather, this should be configurable according to system conditions and application requirements.

Further, the dynamism of the system warrants special consideration for the functional model to update and delete the description of a registered resource. Specifically, the directory service should be able to self-recover from changes in resource capability and accessibility, arising from mobility, disconnection and failure.

As mentioned above, the directory service defines an integral part of our universal substrate, and, hence, needs to be simple enough to be accommodated in impoverished devices, still allowing itself to be scaled to more privileged devices.

The design of UbiqDir meets these requirements by providing an active, extensible functional model that allows all but the primitive service to be dynamically configurable according to the system requirements.

7.4.3 Requirements for Distributed-operation Model

By embedding the directory service in every participating resource, our architecture ameliorates a peer-to-peer distributed operation model, avoiding performance bottlenecks and single node vulnerability.

The distributed operation of a directory service includes policies and protocols for discovery, replication, partitioning, caching, consistency, load-balancing and fault tolerance.

Clearly, all of these functions depend on the system requirements and application needs that vary from one active space to another in a ubiquitous system. For instance, it might be desirable to support strong consistency in environments that are stable and have low access latency whereas eventual consistency with periodic updates might suffice in less stable environments [Winoto99]. Similarly, it might be more efficient to replicate data in certain environments and partition in others [Winoto99].

Hence, the distributed operation model should not be part of the core directory service, rather the directory service should provide mechanisms to efficiently support different distribution operation policies, according to application and system requirements. Further, configurable distributed operation protocols could also lead to interoperability in the face of different standards. This reiteration of the end-to-end arguments [Saltzer84] for system design is most pertinent to address the heterogeneity, dynamism and context-awareness of a ubiquitous system.

In UbiqDir, the extensibility in the functional model allows a configurable distributed operation to support an efficient end-to-end system design.

7.4.4 Requirements for Security Model

The security model of a directory service ensures secure access and modification of its internal state. Encryption and/or integrity checking of the messages generated and received by the functional interface achieves secure interaction, while access control lists and/or capabilities are used to support authenticated access and modification of internal state.

A ubiquitous system would inevitably comprise a multitude of low-level networking protocols, some secure [Haartsen00] others insecure [HomePNA]. Similarly, some active spaces could be assumed to define a closed security domain while others might have to route packets on insecure links. Hence, encryption need not be embedded in the fixed core of the directory service, instead it can be deployed dynamically for environments prone to spoofing attacks or where the low-level protocols do not already have support for encryption.

Access control with authentication, however, is a necessity as resources participating in a ubiquitous system might come from different manufacturers and belong to different users. Hence, access and modification of descriptions exported by different resources could only be allowed to legitimate entities.

UbiqDir provides a fine-grained access control and authentication to meet these requirements.

7.5 Information Model

The information model of our directory service is based on XML. Experience with AutoHAN showed that XML is ideally suited for storing structured but irregular information that is required to capture the state of a ubiquitous system comprising resources of varying number and type of attributes. XML's string-based representation and loose matching of tag strings naturally lends to lookups when only a partial description of a resource is known i.e. "intent".

While systems like X.500 [X.500] and INS [Winoto99] define their own string-based information models that lend to efficient lookups, the *raison d'etre* for XML in UbiqDir is that it is a convenient format for tree-structured data that can be viewed and understood, at least to some extent, by a viewer, person or application that does not have full knowledge of the formal structure of the tree. Instead, the recipient may have partial knowledge, or knowledge of a previous release of the structure definition, or may infer the structure and meaning from the direct use of English and ASCII in the XML format.

Parts of an XML tree that are not recognized can be ignored while still correctly parsing other parts of the same XML document. An example is that XML can be viewed in a web browser with or without style-sheets to define the view. Therefore, XML is potentially a future-proof method of storing structured data, unlike say, early MS Word documents, that needed exactly the correct version of Word to read them, and unlike CORBA and Java RMI, where access to the same definition file is needed by both the sender and receiver of data in these formats.

An important, and debatable, issue is how to drive the description hierarchy in the directory service. Clearly, resource descriptions can be stored in several hierarchical ways based on geographic location, multicast scope, access privileges, price, resource functionality etc.

Depending on the distribution of the data stored in the directory, any of these can effectively organize the directory hierarchy that is efficient for lookups. UbiqDir can, of course, support any hierarchical distribution but we chose to drive the hierarchy is our directory service based on resource functionality for two reasons.

- Resource descriptions as functional hierarchies, as opposed to, per se, geographic location based hierarchies, allows pure names that are efficient to search [Birrell82] i.e. names that do not need the client to know the location of the resource as, for example, is required by DNS.
- The decisive factor however is the “general-category” interoperability leveraged by a functional hierarchy that is not natural to any other organization of data i.e. if an application does not know the exact functionality of a device, it could look it up by using an unspecific classification higher up in the hierarchy. For example, Output/Audio/HiFi can be looked-up and used just as an Audio or even as an Output device if the description HiFi is not known or required. Applications can discover the resources using their classification and interface description registered with UbiqDir and use the Java reflection API to invoke their interfaces.

Resource descriptions consist of attribute-value pairs, organized in a hierarchy. An entity in a ubiquitous system, such as a closed circuit camera, is stored as a fully qualified name, or point, which is similar to the distinguished names in X.500 [X.500] in its hierarchical structure. A point is a concatenation of all the XML tags, starting from the root of the directory, that both identifies the object according to its place in the resource hierarchy and lists attribute-value pairs that delineate the device according to its functionality and context. For example, an object belonging to the object hierarchy of Input/MultiMedia/Video/Camera/StillCamera/, having static attributes of a 10-sec snapshot frequency, a 5-requests capacity and output format of .jpeg, context-aware attribute of a location of corridor, and dynamic attributes of current load of requests would be represented in the directory service as shown in figure 7.1.

```

<Input>
  <Multimedia>
    <Video>
      <Camera>
        <StillCamera>
          <SnapFrequency type=static> 10 sec </SnapFrequency>
          <Capacity type=static> 5 </Capacity>
          <Format type=static> jpeg</Format>
          <Location type= static context> Corridor</Location>
          <CurrentLoad type=dynamic arguments=Camera> load.jar </CurrentLoad>
          <address>255.255.244.255<type>IP</type></address>
          <Hash> A34C7DBA</Hash>
        </StillCamera>
      </Camera>
    </Video>
  </Multimedia>
</Input>

```

Fig. 7.1 An Example Resource Description Registered with UbiqDir

Although, the whole point of using a string-based information model is to allow arbitrary number and form of data to be stored and looked-up with flexibility offered by string manipulation operations, UbiqDir requires that every mobet description should include description of its functional interface, and events offered by it, conforming to the format described in chapter 6. Additionally, UbiqDir requires that every mobet description includes an `<address>` tag, along with a nested `<address type>`, to allow other agents to get to it. The type of the address indicates the preferred protocol to be used by other agents in the system to send message to the looked-up agents e.g. IP, Ethernet. (Chapter 8 shows how this addressing scheme supports flexible protocol stacks in UbiqOS according to application requirements).

UbiqDir enforces the description rules by validating the agent descriptions before they are allowed to install themselves with an instance of UbiqDir. Mobets not conforming to this requirement are denied registration and hence cannot be installed with UbiqOS. Use of XML shows its merit in this respect as well, as the XML parsers can be configured with a DTD Schema [Ludascher99] to allow validation (read type-checking) of certain tags in an XML document without imposing rules on other tags. UbiqDir Schema allows

the agent description to include any information as long as it allows both local and remote agents in the system to discover and invoke its interface to allow dynamic composition of services.

7.6 Functional Model

The functional model of our core directory service is simple and extensible. The functional model allows entities of any sort to be registered, unregistered, updated, and looked-up both by qualified point names and “intent”.

The lookup() interface supports intent-based lookups by allowing non-qualified names, somewhat similar to relative distinguished names in X.500 [X.500] i.e. resources can be looked-up by any subset of XML tags and values comprising a point. Attributes that are required but whose value does not matter are specified by “ANY” for value in the look-up i.e. a wildcard.

Two relational operators, greater-than-equal-to and less-than-equal-to, are supported for range comparisons in lookups. Ranges are especially useful to express the varying properties of a wireless link. Similarly, conjunctions, disjunctions and negations are supported to formulate compound queries. Two “superlative operators”, maximum and least, are also supported which return the resource with the maximum and least value respectively. This expressiveness proved adequate to formulate queries in AutoHAN [Saif01].

Each entry in the directory service is allocated a large, secure ID which both serves to distinguish identical resources and as a secure hashed-index for faster lookups. These IDs include a 32 bit random portion, to minimize the chances of a collision between IDs generated by different instances of UbiqDir. The register() operation returns this ID on success. Clients can then use this ID to update() and unregister() a registered resource without quoting the full-point name every time.

The register() operation inserts the resource description at the appropriate location in the directory hierarchy according to its qualifying name. If a partial point name is used, then the register function inserts the object in the XML tree in the same sub-tree as any other

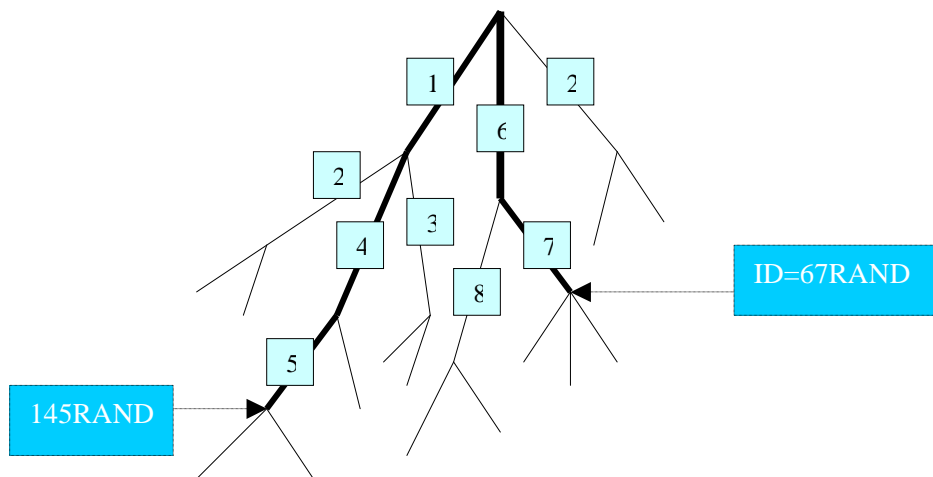


Fig. 7.2 This figure shows the algorithm to generate IDs in UbiqDir. An ID comprises the concatenation of the labels of arcs along the fully-qualified point name of the resource, augmented with a 32 bit random number. These IDs uniquely identify a resource in its context and serve as a secure index to allow fast updates and deletes.

previously registered element of the same category (having the same sub-hierarchy). A fully qualified name belonging to no previously registered hierarchy will, naturally, create a new section in the hierarchy. The write function includes an ID with the description of each resource registered with UbiqDir. This ID uniquely identifies the resource and serves as a secure index for faster future lookups and serves to reduce network traffic for updating the resource, especially when renewing its lease. The ID is generated by a simple algorithm as follows. The register method keeps a simple counter and labels every arc in the XML tree using a (monotonically increasing) number; every new arc inserted in the XML tree gets a number higher than the last arc inserted in the tree. The ID is simply the concatenation of integers along the point's fully-qualified name; a traversal from the root of UbiqDir namespace to the root of the subtree that stores the resource attributes. The ID therefore encodes the path to find the resource description in the XML tree and hence serves as unique identifier for the resource. Additionally, the ID includes a 32 bit random number to uniquely identify the resource in its context so that IDs generated by different instances of UbiqDir in a context do not collide with one another. The sparseness of the 32 bit address space also makes the ID

secure. This index is returned by UbiqDir when a resource is successfully registered with it. Clients can then use this unique ID, instead of quoting the whole XML description of the resource, to update and delete the resource. Figure 7.2 illustrates the use of indexes in UbiqDir.

As `update()` and `delete()`-ion of a registered resource can then use this ID alone, instead of the complete point, to identify the resource: it appreciably reduces network traffic and lookup overhead to update the resource.

The `lookup()` operation, when presented with a resource description, performs a breadth-first search of the XML tree and returns the complete fully-qualified point name(s) of the matching resource(s) i.e. resource's place in the hierarchy and all the attribute-value pairs, along with the lease of the registered resource. Lookups specifying fully-qualified point names yield only the resources that exactly match the query whereas an intent-based lookup returns all the resources matching the (partial) description of the looked-up resource. The resource selection policy is left to the applications or to the Dispatcher module. This is much more flexible than white pages lookup services like DNS that just resolve a name to an address and restricted yellow pages services like INS [Winoto99] that make the resolution and routing decision based on just a single metric. Our architecture allows the selection based on configurable policies that could make a choice depending on any combination of attributes that could be static, dynamic or context driven e.g. least latency, least load, maximum capacity, snap frequency greater than 10 snaps/sec.

UbiqDir is based on the principle of soft-state [Clark88]. Where soft-state enables fate-sharing and leads to self-recovery that helps address the dynamism and mobility in the system [Winoto99], it can burden the network with the overhead of refresh messages [Winoto99]. We address this issue using adaptive lease intervals, that are adjusted to suit the dynamism of an active space. The `register()` function takes a value-result parameter as a second argument that is used by the resource to suggest a lease within which it is happy to renew its registration by using `update()` with a new lease value. The directory service, in return, sets this value to an interval that it deems suitable for refresh. It could either be equal to the suggested value or a different value that is calculated to be suitable for that environment. The lease suitable for a system depends on a number of factors and, hence,

needs to be configurable for an efficient operation. In our architecture, “lease” is itself a special entity that is registered by the lease agent, part of the overlay, with a value suitable for an active space. This value is looked-up by the directory service to decide a lease. In our architecture, a special mobile agent called lease-agent, deployed as an extension to UbiqDir, takes into account the network bandwidth to node ratio [Veizadez 97], network reliability, an average of the suggested lease times, latency in the network, and the number of pre-mature failures of resources (lease expires). These and other factors have unequal importance depending on the system conditions e.g. high latency increases the lease time as it indicates a wide-area active space, pre-mature failures decrease the lease time as this indicates an unstable environment and so on. Configurable lease intervals combined with ID based updates reduce the refresh overhead that dominates performance in soft-state based systems [Winoto99].

This leads to the discussion of dynamic attributes in a resource description. Although no other directory service has explicitly addressed this issue, directory designs [Czerwinski99] [Winoto99], based on soft-state have the provision to get around the issue by enforcing short refresh intervals, to place an upper-bound on the staleness of view. Clearly, this introduces an undesirable tradeoff of exacerbating the refresh overhead against stale-views in the face of dynamism.

UbiqDir addresses the issue by allowing the dynamic attributes to be executable mobile agents written in Java. When a resource is lookup(ed)-up, the agents pointed to by its dynamic attributes are executed by the directory service (with the arguments registered with the agent) and the values computed by them are returned as part of the complete point name. As these agents are dynamically deployed by the system according to context requirements, they are free, however, to use any caching policy within themselves that suits the dynamism of the system. Secondly, this approach allows dynamic attributes to be context-driven, unknown to the device, by allowing the system to augment the device descriptions with dynamic attributes pointing to mobile agents tailored for that system. Finally, this scheme allows the mechanism used to calculate these attributes to be configurable e.g. ICMP ping or link layer echo to measure latency, consistent with the networking standards of an active space. Update() can be used to change the description of a registered resource, in addition to renewing its lease. Unregister() is provided to

force an immediate deletion of a resource. Though this operation might appear frivolous in a soft-state based design, we have found it to be especially useful in implementing distributed policies to ensure stronger consistency between directories, as described under distributed operation.

Finally, the functional interface of the directory service supports a method to `check()` the existence of a resource, given a hashed-index. This interface returns the lease of the resource if it exists and null otherwise. We have also found this interface to be especially useful to support distributed caching policies as described under distributed operation.

7.7 Implementation of Romvets

The search operations implemented by UbiqDir core are used to implement the subscribe/notify Romvets architecture described in chapter 5.

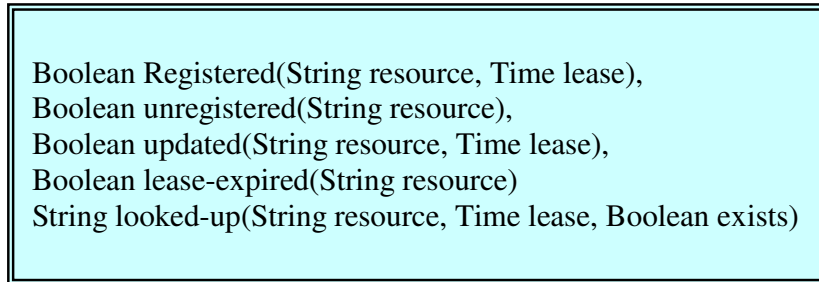
The lookup, register, delete and update operations of UbiqDir are used to implement the notify, subscribe, delete and update operations of the Romvets interface respectively.

Notify operation involves lookup of any matching event-handlers and their invocation using optimization implemented with Kaffe. Subscribe is just another interface for the register operation in UbiqDir, which allows subscription for events by registering the XML description of the desired arguments of the event. Likewise, delete and update interfaces for Romvets are implemented by delete and update operations of UbiqDir.

The soft-state of UbiqDir fits well with the Romvets model allowing automatic purging of obsolete subscriptions.

7.8 Extensibility in UbiqDir

The design goal of a simple core with configurable policies has already been emphasized to effectively address the challenges posed by the heterogeneity and dynamism of the system.



```
Boolean Registered(String resource, Time lease),
Boolean unregistered(String resource),
Boolean updated(String resource, Time lease),
Boolean lease-expired(String resource)
String looked-up(String resource, Time lease, Boolean exists)
```

Fig. 7.3 Events offered by UbiqDir to Request Extensible Distributed Operation

Extensibility in UbiqDir, indeed, follows the generic mechanism of extensibility in UbiqOS: UbiqDir generates a synchronous event, using Romvets, corresponding to its functional interface to request context-specific distributed operation.

The directory core generates five events corresponding to its functional interface to indicate changes in the description of resources registered with it, as shown in fig. 7.3.

The “Lease_expired” event is the only event that is generated spontaneously by the directory service, triggered by the expiry of a lease. All other events result from the corresponding operation on the functional interface of the directory service i.e. register, unregister, update and lookup.

The mobile agent deployed on an instance of UbiqDir use its subscribe(String event_name, XML_ResourceDescription) method to register interest in any of the five events offered by the directory service to extend its distributed operation. The subscribing agent is notified whenever the corresponding function is invoked or a lease expires for a resource matching the XML description specified (as the second argument) when subscribing. This allows extensions to support policies on the granularity of an individual resource.

The mobile agents implement the distributed operation of the directory service by subscribing and handling these events from one instance of UbiqDir and disseminating this information about changes in the local directory to their peers on the network using policies and protocols suitable for an active space.

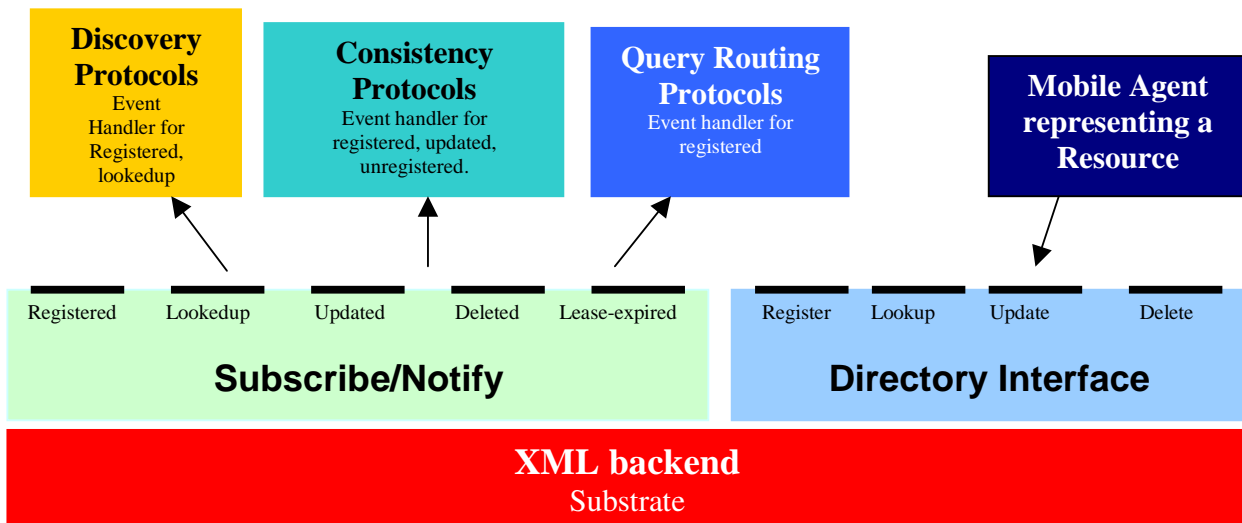


Fig. 7.4 Extensible operation of the directory service

These mobile agents implement an event-handling interface that is up-called by the directory core to deliver a corresponding notification. When notified, these mobile agent event-handling routines block and return a value once finished executing. The event handlers registered for an event are all executed after the corresponding local operation is performed and before the result for the function is returned. The functions corresponding to events return only after all of the event handlers for an event have returned a value. Romvets collects all the results in a vector and returns it to UbiqDir. This vector is used by SEMAS to determine the result of the extensible operation, as described in the next section.

For all event-handling except the lookedup one (which returns a string value), UbiqDir takes logical OR of the boolean values returned by the event handlers. “False” value forces an error condition to be returned for the corresponding function, while a true value causes normal operation to resume with the corresponding values returned. It is the responsibility of the event handler to “undo” the action of the corresponding function before returning a false value.

The handlers for looked-up(resource, lease, exists) event return a string value. If the value of “boolean exists” is set to false by UbiqDir when generating the event, that indicates that the looked up resource was not found in UbiqDir, then the extensions search for the agent with the matching description in the device context and, if found, return the XML

description as a string. UbiqDir can then return these values as the result of the corresponding lookup() operation.

This basic extensibility provides configurable distributed-operation and allows more sophisticated models for application adaptation [Esler99][Banavar00] and production rules to be supported on top of this basic model [Ceri96], as discussed in section 7.9.

7.9 Distributed-operation model

The peer-to-peer model provided by embedding a directory service in every resource, and the extensibility provided by the subscribe/notify events architecture allow sufficient flexibility to support configurable distributed operation to suite the requirements of the current active space of a resource. This section shows how we have used this model to implement a range of distributed-operation policies and protocols to allow context-aware adaptation of UbiqDir's distribution operation, refer to fig. 7.4.

7.10 Discovery and Caching

All of the local resources in a device are registered with its local instance of UbiqDir. The first consideration for distributed operation is how to make these resources accessible to other entities in the system. In other words, how do the resources discover other non-local resources in the system? In our system, this translates into how and when do the instances of UbiqDir embedded in every device exchange their information? The answer of course, as stated above, depends on the active space requirements and networking standards. Hence, making such a mechanism part of the core service, like the assumption of well-known multicast channels in SSDS [Czerwinski99] and SLP [Veizadez97], could lead to inefficiency and limit interoperability. Second, the scope of the context of the device also depends on the active space the device happens to be in; the context of the device could be all the other devices on the same IP subnet, on the same bluetooth ad-hoc network etc. In UbiqDir, the "looked-up" and "registered" events generated by the UbiqDir core are used by the mobile agents to implement different discovery policies. Mobile agents

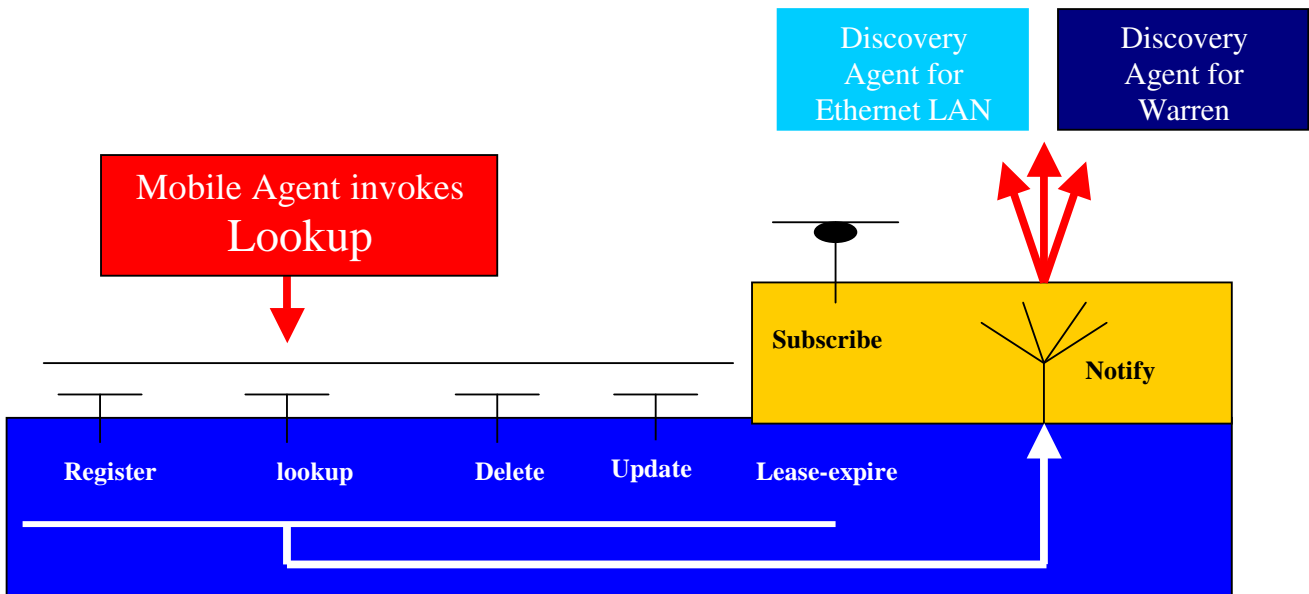


Fig. 7.5 UbiqDir can support multiple discovery protocols simultaneously, each for a different standard and network interface. These discovery protocols are deployed as extensions to UbiqDir and subscribe to the “looked-up” event, generated whenever a resource description is looked up in UbiqDir. These extension can then find the matching resources in their respective networks and return the descriptions to the mobile agent looking up the resource.

deployed to implement discovery policies and protocols subscribe to the “boolean looked-up(string resource, Date time-lease, boolean exists)” event to be notified whenever a particular resource is being looked-up. They check the “boolean exists” argument to determine whether the resource is registered with the local instance of UbiqDir. If the value of this argument is set to false by the directory then the event handlers for the “looked-up” event block and search for the resource on the network. The discovery mobile agents contact their peers on the network, which lookup in their local instance of UbiqDir and return the resource if one matches the desired description. Further, the requesting agents can selectively cache some of the entries hence discovered, by registering the description with the local UbiqDir, to avoid future lookups on the network. The core checks the value returned by the discovery mobile agent and if the value is not set to null by a discovery agent then the resource description returned by the

agent is returned for the corresponding lookup operation. Similarly, the event handlers for “registered” event can pre-fetch some of the entries from other remote directories.

These event handlers, of course, are free to use any low-level networking protocol suitable/standardized for that environment.

Multicast invocations [Bershad95] leveraged by modeling extensions as event handlers, allow more than one discovery protocol and policy to be used simultaneously, each, for instance, for a different scale in a wireless overlay network [Katz96]. We have also used this model to support “ubiquitous-discovery”, where a multi-homed device can discover resources on more than one network (IP and ATM in our case) corresponding to its network interfaces. Resources in our test machine can successfully discover resources both on the lab wide IP/Ethernet network and Warren [Greaves98] ATM network, simultaneously, by deploying two mobile agents, as event handlers for “lookedup” event, that transmit discovery packets on the corresponding network layer when notified, as shown in figure 7.5. The IP/Ethernet discovery mobile agent discovers the resources in the same IP subnet, while the Warren discovery mobile agent discovers resources on the Warren network.

Likewise, these event handlers can support any high-level discovery protocol. For example, we have prototyped a simple SLP [Veizadez97] style discovery protocol, as an extension to UbiqDir, which allows resources to be discovered in the same SLP domain. Similarly, other protocols like Salutations [Pascoe99], and SSDP can be supported.

7.11 Replication and Consistency

Once discovered, descriptions of the resources can be cached at the local instance of UbiqDir to avoid future lookups on the network. However this replication of information raises issues of consistency.

Several consistency algorithms have been proposed in the past, each suitable for a different set of system idiosyncrasies and application requirements.

Stable systems with stringent data integrity requirements warrant strong consistency mechanisms like two-phase locking and transaction-based updates like two-phase

commit, while eventual consistency sustained by soft-state refresh messages has been argued to be sufficient and desirable for dynamically changing systems like mobile networks [Brewer98].

Soft-state in our architecture provides default recovery, while events offered by the active core allow stronger consistency policies to be supported. Lease timeouts cause loss of remote state, which could lead to on-demand discovery and performance degradation, but whatever state is replicated can be managed by different consistency policies for correct operation. This architecture provides the desired robustness in the face of system dynamism while allowing mechanisms to efficiently meet requirements of stronger consistency under varying system conditions.

We have prototyped two-phase commit style transaction-based updates using this architecture. Mobile agents in devices with persistent storage can use registered, updated and unregistered events to ensure that changes in the local state are not committed until all of the replicated state is updated. The event handlers for these events only return true after all of the replicated state is updated, by contacting their peers and using distributed transactions to inform them of the changes in the state at the local instance of UbiqDir. It is the responsibility of the event-handling agent to rollback the transaction before returning an error condition to make the corresponding state modification operation fail.

We have found eventual consistency sufficient in our home network system for most of the services, though stronger consistency is used for security critical services and when stale information can adversely affect system performance. For instance, it is imperative to quickly disseminate the information that the burglar alarm has been shut down. Similarly, it is beneficial to make newly registered resources immediately visible in the system to avoid poor choices to be made when new resources can better satisfy the indicated requirements.

Likewise, a range of consistency policies between these two extremes can be supported depending on the system requirements. We have found strongly consistent register operations with eventually consistent unregisters to be an efficient model for mobile clients, as new resources made accessible to the client are usually the nearest ones available and often present the “best” choice. Resources that are made inaccessible by client mobility already present a poor choice in terms of accessibility, reflected in their

dynamic and location attributes, and a false indication of their presence does not lead to rebinding that could adversely effect system performance.

7.12 Load Balancing and Fault Tolerance: Overlay topologies

The replication of state provided by the events offered by the directory service can be used to distribute state for better load balancing and fault tolerance. In the extreme case, all of the entries corresponding to local resources can be distributed to every directory in the system so that the lookups can be performed locally to protect the source host from query overload. This model works well when eventual consistency is acceptable but incurs high overhead of synchronization when stronger consistency is required.

This architecture is also the key to scalability for wide-area active spaces. Overlay agents can be used to dynamically organize the directories in the system in any topology suitable for the requirements of an active space. State is replicated, updated and retrieved according to this topology to make lookup, register, unregister and update traffic scalable. We have prototyped three overlay topologies using our framework.

In the first topology, the discovery agents organize themselves in a hierarchy and use bloom-filters [Mullin83] to route queries, as proposed in [Czerwinski99]. Where bloom-filters provide a scaleable hierarchical structure, they are prone to generating spurious network traffic because of false positives. We have found this scheme to be most amenable to intent-based lookups in smaller active spaces, e.g. smaller rooms in our reference implementation of a home area network [Saif01], but wastes appreciable bandwidth for low-bandwidth ad-hoc wireless network like IrDA.

In the second approach, the discovery mobile agents organize themselves in a multicast spanning tree, as proposed in [Winoto99]. This approach efficiently utilizes network bandwidth but provides no fault tolerance, as link disconnections can lead to network partitioning. Given that most of the network links in our Home Area Network are reliable, we have found this topology to be most suitable.

Finally, in the most basic topology, the mobile agents deployed broadcast everything on the network. Where it leads to broadcast storms, this topology provides the maximum fault tolerance, and is well suited to active spaces with a high degree of dynamism.

Similarly, other overlay topologies and group communication protocols can be used by the mobile agents, suiting the requirements and characteristics of an active space.

7.13 Context-aware Adaptation and Production Rules

By unifying the events architecture with the lookup service, our architecture also bridges the gap between resource discovery and corresponding application adaptation. Applications can subscribe to the events offered by UbiqDir to be notified about changes in the resource context. For example, an application that streams video from a virtual VCR to the best display available in the active space can subscribe and be notified when a new display enters the active space, or the older one becomes inaccessible, to allow it to rebind itself to a better display.

Similarly, when notified about changes in the resources constituting the device context, mobile agents can be moved around among the accessible devices for load-balancing, fault-tolerance, high-availability. For instance, mobile agents can be moved to newly accessible device for load-balancing. Likewise, mobile agents can be replicated on more devices if a device with a replica becomes inaccessible (lease expire) to allow for fault-tolerance and high-availability.

Similarly, the events offered by the directory service can be used to support production rules as part of the overlay [Ceri96]. The production rules can be of varying granularities and are used for different purposes.

We have used this feature to augment the AutoHAN model to automate our reference implementation of a home area network. The events generated by the active core can be used by AutoHAN event scripts. These production rules written specify event-condition-action bindings to automatically control a home network [Saif01]. For instance, the lease-expired and unregistered events can be used to express a home automation

```
unregistered(<heatingcontrol>Central</heatingcontrol>) |  
leaseexpired(<heatingcontrol>Central</heatingcontrol>) → boiler.turnoff()
```

Fig. 7.6 An example production rule for events offered by UbiqDir

policy like “if the heating control stops working then turn off the boiler”, as shown in Fig. 7.6.

Similarly, production rules written in a well-specified algebra allow integrity constraints to be imposed on system operation and to detect and resolve conflicts in system operation. For instance, the system can impose policies to detect and report or permanently shut down a faulting resource that comes up and goes down regularly etc. Similarly, it could be forbidden to register a resource when some other similar resource is already registered with the same attributes by another incompatible manufacturer etc.

This work is the subject of next phase of research in the AutoHAN project.

7.14 Security Model

The security model of UbiqDir ensures legitimate access and modification of its internal state. Every entry in the directory service includes a list of entities that are allowed to access and modify the entry.

This access control list is part of the description exported by the registered resource (shown in figure 7.7).

Every tag comprising the resource description can specify its access control list. The access control list part of a tag higher up in the hierarchy is cascaded to the tags lower in the hierarchy if a successor tag does not specify its own access control. Hence, access control specified by a resource is also applied to all of its attributes, though attributes are free to override it. This allows a fine-grained access control where some of the sensitive attributes can have selective visibility and modification privileges while others have the

```
<StillCamera Access=Everyone Modify=StillCamera, Owner>
  <SnapFrequency type=static> 10 sec </SnapFrequency>
  <Capacity type=static Access=StillCamera> 5 </Capacity>
  <Format type=static> jpeg </Format>
  <Location type=static context Modify=LocationService StillCamera> Corridor </Location>
  <CurrentLoad type=dynamic Modify=None> load.jar </CurrentLoad>
  <address> 255.255.244.255 <type> IP </type></address>
  <Hash Access=none Modify=none> A34C7DBA</Hash>
</StillCamera>
```

Fig. 7.7 Access Control Specification in UbiqDir

same access control as the exporting resource. This implementation is also useful to accommodate attributes exported by system context e.g. location service. Context-driven attributes are exported by system services and, therefore, need to give modification privileges to services other than the resource.

The access control list comprises names of groups with appropriate privileges. These names refer to a group membership hierarchy within the directory service (fig 7.8).

“None” and “everyone” are well-known keywords known to the directory service.

Resources are authenticated using a capability-based system. As stated above, a secure ID is returned when a resource registers its description with the directory service. This ID serves both as a capability to authenticate the resource and as an index into the hierarchy for faster updates. The capability is internalized i.e. it is only meaningful to the issuing directory service.

Capabilities in our architecture are transferable and resources can assume the privilege levels of one another by exchanging capabilities. The exchange is done outside the directory service framework. To be registered at level n in the group membership hierarchy, the registering party needs to provide the capability of the resource at level $n-1$. The privilege level of an ancestor in the group membership hierarchy is inherited by its successors.

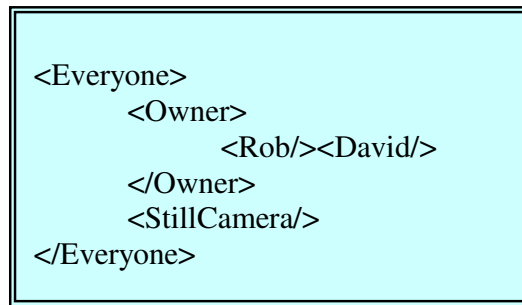


Fig. 7.8 An Example Group Membership Hierarchy of Principals in UbiqDir

This basic authentication and access control scheme is also used to ensure that only legitimate agents in the overlay are notified about the events generated due to a resource. Agents not in the access control list are not notified with the events corresponding to a resource. Extensibility introduces thorny issues of system security and integrity. The authentication and access control only provide a rudimentary solution based on the assumption that extensions coming from responsible resources would not misbehave. Though Java's type safety and sandbox guarantee that misbehaving extensions cannot access and modify arbitrary system components, our security model does not protect against extensions that block forever. Terminating these extensions after an arbitrary time can leave the system in an inconsistent state [Small96]. The runtime overhead introduced by instruction level rollback to undo the wrong doings of a misbehaving extension makes this solution infeasible for embedded devices. This is still an open research issue and a subject of our current work.

Encryption is optional in our architecture. Overlay agents can use encryption if the underlying network is insecure or the network is prone to active security attacks. This also allows different encryption algorithms to be supported suiting the system requirements.

7.15 Bootstrap

As mentioned earlier, every component in UbiqOS is modeled as a mobile agent and registered with UbiqDir to export it to other components in the ubiquitous system. In addition to the dynamically deployed extensions, this includes all the device drivers for hardware resources embedded in the device and the two components at layer 1 i.e. UbiqDir and SEMAS. As these components are fixed, they always return false to any mobility or life-cycle management requests (and hence those mobet methods are not shown as part of their external interface). However, these components register their network addresses and functional interfaces with UbiqDir like all mobets to enable other components to discover and make use of them. This allows the handles for layer 0 components as well as UbiqDir and SEMAS to be used as destinations for migration and replication requests. Indeed, that is how agents request to be migrated to another host; agents request to be migrated to a host by giving a handle of a fixed component at that host.

7.16 Comparison with Related Work

Naming and service discovery has been an issue since the inception of computer systems. A resource must have a name to let other things in the system reference it [Birrell82]. Classically, names have been divided into pure and impure names [Birrell82]. Where pure names provide location independence, their traditional representation as flat bit strings can lead to inefficient lookups. Therefore, most of the wide area naming systems like DNS and GNS support impure names where names are committed to fixed resolution contexts.

This makes impure names unsuitable for ubiquitous systems [Winoto99], where names should mean what and not where [Toole92]. Additionally, resources in a ubiquitous system have varying number and type of attributes and can usually only provide an indication of the service they are looking for instead of its precise functionality. This renders directory systems based on strict type schemas, like Grapevine[Birrell82] and X.500 [X.500], overly restrictive to capture and lookup resource descriptions in a

ubiquitous system. Our directory leverages XML's simple string based representation of data and loose-matching of tags to provide a flexible namespace.

The dynamism of a ubiquitous system cannot be captured by static descriptions alone. Further, the dynamic attributes of a resource are only meaningful when resolved from the client's perspective. This aspect has not been explicitly addressed in any of the previous distributed system designs where directory services are, often physically separated, external repositories of static information. Discovery services based on soft-state, like SSDS and INS, can use refresh messages to limit the staleness of information in a dynamically changing system but this leads to an undesirable tradeoff between network traffic and processing overhead caused by refresh messages and staleness of information. UbiqDir addresses these issues by embedding an extensible directory service in every participating resource and by allowing dynamic attributes to be attached to context-specific Java mobile agents that can calculate dynamic information according to system and application requirements. Use of active computations to resolve names has been explored in active names [Vahdat99] in the past, but the focus of that work has been to provide flexibility in wide area discovery and transport. Our architecture uses active computations to augment an indication-based lookup system to allow context-specific dynamism in resource descriptions.

XML presents a human understandable description of the system that is not natural to other systems that use proprietary encoding of objects [Vinoski97] [Waldo99] to represent the state of the components in the system. Additionally, these directory services enforce fixed distributed operation policies and, more often than not, language-specific APIs to access and modify the directory service. Examples include Sun's Jini [Waldo99] that solely relies on RMI and java-centric APIs, and IBM T spaces. XML is used as a description format in UpnP discovery protocol [UPnP], but it lacks the expressiveness offered by our directory service. SSDS [Czerwinski99], on the other hand, provides a rich XML searching facility but lacks support for context-specific adaptation.

Extensibility has been researched in other contexts, especially extensible operating system kernels [Bershad95] [Seltzer94], with the aim to provide application-specific adaptation of system components for better performance. The primary focus in UbiqDir, on the other hand, is context-driven adaptation to address the dynamism of the system

and to provide efficient interoperation of heterogeneous system components as a resource is moved from one active to another or new resources join an active space, changing the system characteristics.

Active database systems [Ceri96] have been used to support production rules to support applications like trigger alarms and to enforce integrity constraints on the data stored in the database. Whereas, the events offered by our directory service are used to provide extensibility and to automate an active space, as well.

Finally, UbiqDir allows different discovery protocols like SLP[Veizadez97] and salutations [Pascoe99] to be supported atop the extensible core to leverage interoperability with existing services.

With support for intent-based lookups, context-driven dynamic attributes, self-recovery, configurable distribution operation policies and support for production rules, while still managing to keep the core service simple enough to be accommodated in limited capability devices, UbiqDir addresses the dynamism, heterogeneity and context-awareness of a ubiquitous system better than any other lookup or discovery service available [Veizadez 97] [Winoto 99] [Czerwinski 99].

7.17 Summary

This chapter established the need for a naming system and a directory service to address the dynamism, heterogeneity and context-awareness unique to a ubiquitous system. It highlighted three characteristics of a ubiquitous name: intent-based lookups, dynamism and context-awareness. It then motivated the need for a configurable directory service and proposed that it should be part of the universal substrate to allow effective participation of resources in a ubiquitous system.

UbiqDir, like all other extensible components in UbiqOS, is divided in two parts, an extensible core and context-specific extensions. The core of the directory service provides the basic directory access operations and is based on soft-state to allow self-recovery in the face of disconnections and node failures. The functional interface of the core offers sufficient expressiveness to register resources with varying number and type

of attributes and to lookup resources using any combination of relational and logical operators. The events offered by the active core are used to support context-specific distributed operation, application adaptation and production rules to leverage self-organization of active spaces. The XML-based namespace in our directory service provides maximum flexibility to accommodate resources of varying number and type of attributes and presents human-understandable representation of the state of a ubiquitous system.

Security in our architecture is provided by fine-grained access control and authentication of resources. The overlay consists of Java mobile agents, running in the Java interpreter sandbox. This protects against misbehaving extensions from corrupting other system resources.

The core of the directory is small enough to be accommodated in even limited-capability devices, as apart of UbiqtOS, while extensibility allows context-specific scaling.

We have found this functionality provided by UbiqDir to be adequate in a practical home area network, as described in [Saif 01].

“Those who can, build. Others simulate”

Anonymous

Chapter 8

Implementation

Chapter 5 presented an overview of the architecture of UbiqtOS to enable dynamic, application-specific and context-aware adaptation. Chapters 6 and 7 respectively described the architecture of SEMAS and UbiqDir to illustrate how distributed operation is dynamically adapted in UbiqtOS to suit the characteristics of the current context of the device. This chapter describes a prototype implementation of UbiqtOS using Flux Oskit [Ford97] and Kaffe [Wilkinson00]. It presents a “standard distribution” of the operating system, describing the components of UbiqtOS implemented and evaluated as proof-of-concept.

This chapter describes an

- Implementation of a minimal micro-kernel to implement layer 0 using OSKit components
- Implementation of extensible scheduling and protocol stacks
- Implementation of Romvets, UbiqDir and SEMAS and
- Implementation of default extensions for SEMAS and UbiqDir to support a working prototype

This chapter also describes an optimization in Kaffe to efficiently execute the event handler for events routed by Romvets and shows how this architecture is used to efficiently support extensible scheduling policies and network protocol stacks at layer 2 in the UbiqtOS prototype. Further, this chapter presents an optimization in the Romvets implementation to avoid repeated XML tree lookups for operations with low-latency requirements, like processing of packets arriving on a network interface, and shows how

this allows UbiqtOS extensible protocol stacks to deliver comparable performance to traditional operating systems.

The prototype implementation of UbiqtOS demonstrates that a practical system can be built using the artifacts and principles proposed in chapter 5, 6, and 7. The prototype implementation shows that it is practical to support traditional operating system services atop an (optimized) interpreter to allow safe, dynamic, context-aware adaptation using mobile code to address the challenges posed by a ubiquitous system design.

8.1 Background

Any realistic OS, in order to be useful even for research, must include many largely uninteresting elements such as hardware dependent code for kernel startup, memory management, interrupt handling, context-switching, device drivers for embedded resources etc.

As the purpose of implementing UbiqtOS is to demonstrate the benefit of dynamic context-aware and application-specific adaptation provided by layer 1, UbiqtOS implementation focuses on issues above the platform dependent code at layer 0. Instead of implementing all these low-level details from scratch, UbiqtOS is implemented using the Flux OSKit framework from University of Utah [Ford97] to implement layer 0.

Although, the prototype implementation was done on x86 platform due to the availability of prototyping tools (like OSKit), it can clearly be done on any other platform supported by embedded devices. The implementation was done on a diskless machine with limited RAM (12MB) and multiple network interface cards to emulate an embedded device of the future.

8.2 Implementation of Layer 0

Layer 0 of UbiqtOS prototype is implemented using the Flux OSKit from University of Utah [Ford97].

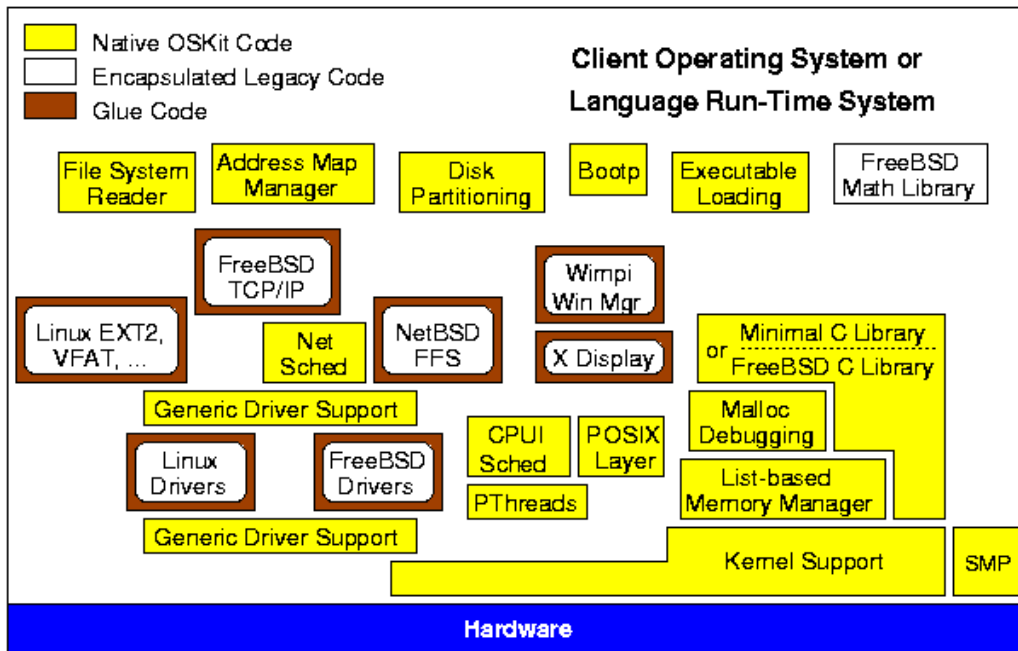


Fig. 8.1 OSKit: The COM-based framework and modularized libraries provided by OSKit to implement an operating system.

8.2.1 Flux OSKit

The Flux OSKit (shown in Fig. 8.1) provides a framework and a set of modularized libraries with well-specified interfaces to assist in the construction of an operating system on the x86 hardware. It provides functionality such as simple bootstrapping, memory management, debugging support for kernel development and even high-level subsystems such as protocol stacks and filesystems. The OSKit is designed to give developers an easy starting point to investigate more-involved issues in operating system research. OSKit's modular structure allows developers to replace the generic functionality provided by the framework with their own code to selectively specialize the operating system to suit their research goals.

8.2.2 Changes made to OSKit

Where OSKit framework encapsulates and exports hardware resources at an appropriate level of abstraction to allow high-level operating system development, it needed to be

changed in the following ways to implement the UbiqtOS architecture presented in chapter 5.

8.2.3 Support for Dynamic Extensibility

OSKit provides a build-time extensible system where components can be compiled together to assemble a custom operating system. Whereas, UbiqtOS is based on dynamic extensibility. Though OSKit is primarily used to implement the fixed, hardware-dependent substrate at layer 0, the following three modules of OSKit needed to be changed to allow dynamic extensibility proposed in chapter 5.

- The OSKit scheduler was changed to allow dynamic adaptation of scheduling policies to suit the requirements of the context-specific software deployed with a device. The modified scheduler generates scheduler activations[Anderson92], using Romvets, whenever required to make a scheduling decision. Extensible scheduling policies, deployed as Java mobile agents and executed by SEMAS, subscribe interest in these events and make scheduling decisions according to application requirements residing with the device.
- The OSKit network communication framework was changed to allow dynamic adaptation of protocol stacks to suit the requirements, characteristics and standards of an active space. The packet drivers generate events, using Romvets, to notify interested components that a packet has arrived on a network interface. Network protocols, deployed as Java mobile agents to suit the characteristics and standards of an active space, subscribe to these events to implement network protocol stacks at layer 1.
- Additionally, the OSKit scheduler and memory manager at layer 0 were changed to expose the memory and processor utilization to load-balancing agents at layer 2.

8.2.4 Changes Made to OSKit COM model

Components in OSKit use Component Object Model [Rogerson97] for interface definition, discovery and garbage collection. COM model provides implementation

independent interface definition; any object can implement an interface to export a specific “view”. Each interface has its own independent function table through which methods can be invoked on the object implementing the interface.

Interfaces are identified by algorithmically generated DCE Universally Unique Identifiers (UUIDs). Further, given a pointer to a COM interface the object can be dynamically queried to return pointers to the other interfaces implemented by them. Interfaces also keep a count of pointers to themselves to allow for automatic garbage collection. Though this model allows OSKit components to be modularized as COM objects, it suffers from the following shortcomings to implement UbiqtOS.

A reference to a COM object can be obtained only if its Universally Unique Identifier is known a priori. Given an interface UUID, the objects implementing the interface can be found using the COM Registry Object. OSKit itself is built around a bunch of COM registry objects that are queried to find other objects in the kernel. For instance, there is a registry object to find functions that implement the C library (libc) for POSIX environment e.g. malloc. As references to objects are obtained by searching the COM registry at runtime, this technique reduces static code dependencies even for low-level code, allowing existing objects to be replaced with new ones to allow specialization of the OSKit kernel. Though the design of UbiqtOS is itself essentially based on the same technique, OSKit’s use of dynamic lookups for this fundamental code introduces appreciable overhead in the critical path of operating system operation. Hence, two major modifications were made to the COM model employed by OSKit to make it suitable for implementing UbiqtOS.

- The low-level functionality like kernel code, memory allocation objects, C library etc. were re-coded without the indirection of COM registries. It is worth mentioning that the OSKit developers also realized this shortcoming and OSKit code is scattered with “Macro-hacks” where the macros used to search the COM registers have been replaced by macros that serve to “inline” the code being accessed. In the UbiqtOS prototype implementation, all the indirections from the low-level code have been removed to integrate the code fundamental for kernel operation.

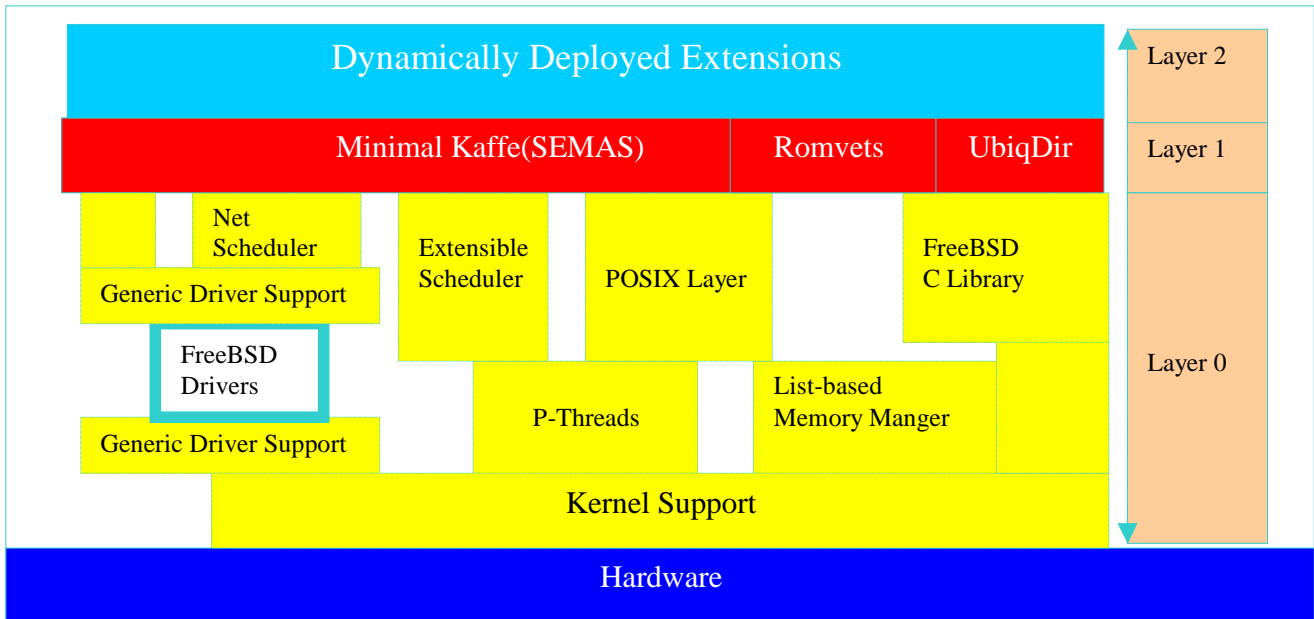


Fig. 8.2 Structure of OSKit components to implement layer 0 in UbiqOS prototype. Compare with Figure 8.1 to note that only a minimal set of Oskit components are used. The dotted lines between components signify that the COM indirection has been removed to improve performance. The list based memory manager, FreeBSD packet driver and scheduler were modified to support dynamic extensibility.

- Second, the interfaces for the objects used to control the resources embedded in the device, i.e. device drivers, are exported to layers 1, 2 in UbiqOS as follows:-
 - The upper half of the device drivers (c.f. FreeBSD), used to send commands to the device, were changed to export themselves to UbiqDir at layer 1. The device drivers export themselves by describing the capabilities of the device in XML along with the Java Native Interface (JNI) wrapper for its COM interfaces(c.f. agent descriptions in chapter 6) to UbiqDir at layer 1. This allows dynamic discovery of the resources embedded in a device without prior knowledge of the UUID of the interface implemented by its device driver.
 - The lower half of these device drivers (c.f. FreeBSD), used to handle interrupts from the device, were changed to generate corresponding events (upcalls) using the Romvets interface to allow extensions at layer 2 to subscribe to them to control the device.

8.2.5 Structure of Layer 0

OSKit, as the name suggests, provides a range of objects from low-level bootstrapping code to high-level subsystems like a filesystem. Whereas, the goal of UbiqtOS is to minimize the fixed part at layer 0 and to push high-level services atop layer 1 to allow for context-aware adaptation. Hence, only a fraction of the architecture shown in Fig. 30 is employed in the UbiqtOS implementation. The purpose of layer 0 in UbiqtOS is to provide enough capability to support the three artifacts at layer 1 i.e. SEMAS, UbiqDir, Romvets. SEMAS requires a Java Virtual Machine to be ported on layer 0, whereas UbiqDir and Romvets are themselves implemented in Java (and compiled to native code for efficiency), and therefore do not need any special support from layer 0.

Therefore, only that minimal set of components of OSKit is used that is just sufficient to support a minimal Java Virtual Machine at layer 1. All of the context-specific extensions that implement the operation of the operating system are executed within this Java Virtual Machine. This Java Virtual Machine supports extensible scheduling and extensible network protocol stacks i.e. scheduling and network communication are pushed atop layer 1, to allow adaptation by context-specific software deployed as mobile agents. Therefore, essentially, layer 0 in the UbiqtOS prototype only includes the OSKit modules that implement hardware-dependent code for network interface cards, physical memory and context switching to support extensible protocol stacks, memory management and thread scheduling as part of layer 2. Additionally, layer 0 includes a C library to support a POSIX interface to allow the Java Virtual Machine to be ported atop. The structure of OSKit components at layer 0 in the prototype implementation of UbiqtOS is shown in figure 8.2.

8.3 Implementation of Layer 1

Layer 1 uses the support at layer 0 to implement its three artifacts that provide safe, application-aware and context-specific adaptation of UbiqtOS. SEMAS requires a Java Virtual Machine to be ported atop layer 1, whereas UbiqDir and Romvets are themselves implemented in Java and compiled to native code for improved performance.

8.3.1 Kaffe Port for UbiqtOS

OSKit comes with a rudimentary port of Kaffe [Ford97], a publicly available implementation of Java Virtual Machine, atop OSKit. This port, referred to as Java/PC [Ford97] by OSKit, basically resolves all the external calls made by Kaffe, primarily using the libc component of OSKit, to allow compilation of Kaffe atop OSKit. Kaffe threads are ported on top of POSIX threads component of OSKit and the Kaffe memory manager is ported atop the list-based memory manager in OSKit to allow Kaffe to execute Java programs.

Although this Kaffe port for OSKit somewhat works, it does not fit the UbiqtOS design. UbiqtOS is designed to allow context-aware adaptation of all those system services that could effect interoperability, efficiency or availability of the system. UbiqtOS design pushes all but the hardware dependent code atop layer 1 to allow for adaptation of operating system services using mobile agents executed by SEMAS, whereas the Java/PC architecture follows the conventional model of running a full blown virtual-machine atop a complete, fixed micro-kernel. Therefore, the rudimentary port of Kaffe atop OSKit

- Does not lend to adaptation of the operating system services proposed by UbiqtOS and
- Results in a large memory footprint not suitable for embedded devices.

UbiqtOS implementation addressed these shortcomings as follows.

Kaffe implementation for UbiqtOS includes only a minimal fixed functionality. In addition to the support for JNI and Javah, UbiqtOS implementation has only the core Java classes (java.io.* , java.lang.* and java.util.*) statically linked with the Kaffe Interpreter. All other classes (like java.sql.*, java.beans.* etc), not needed in an embedded environment, are removed from the static image.

UbiqtOS implementation also removes network protocol stack from layer 0, implemented by OSKit, and, consequently, its corresponding support from Kaffe (java.net.*). Network communication defines the basis for distributed operation and hence needs to be adapted

as the context of the device changes. Further, as UbiqOS proposes explicit bindings that are looked-up and used like other mobile agents to allow communication between two agents, the classes that implement the sockets API are also not needed in Kaffe. Instead, as mentioned above, the packet drivers used to drive the NICs attached to the host are directly exposed, in JNI wrappers, to layer 2 to allow context-specific software to implement protocol stacks, as described in chapter 5.

Further, as prototype implementation is targeted for diskless clients, all the classes needed to support a filesystem from `java.io.*` are removed from Kaffe. Note, the only abstraction used in UbiqOS is that of a mobile agent, including “files”, which are mobile agents with additional methods to read and write byte streams.

Similarly, no virtual memory is supported in the prototype implementation of UbiqOS. Kaffe’s memory manager uses the simple list based memory manager of OSKit to allocate memory. However, to avoid ungraceful termination of programs due to abrupt exhaustion of physical memory, the list-based memory manager of OSKit is extended to support a “warning-scheme”. In this scheme, the list-based memory manager generates an event, using the Romvets architecture, whenever the free memory on its list falls below a certain threshold (10% in the prototype implementation). Context-specific extensions can subscribe to this event to be notified that the local memory utilization is high and can request other mobile agents to migrate to another host to balance load in the system. The list-based memory manager of OSKit also exports itself to layer 2 by registering itself with UbiqDir to provide a JNI-wrapped COM interface that returns the percentage of free blocks on its list. This method can be used by agents that implement load-balancing policies, at agent launch time, and the agents that implement effective mobility.

Finally, Kaffe’s scheduler is ported atop an extensible scheduler offered by OSKit to allow dynamic adaptation of scheduling policies to suit the requirements of the context-specific software deployed at a device. Extensible scheduling is described below.

decisions. The extensible scheduler at layer 0 presents the application-specific scheduling policy with its ready queue and the application-specific scheduler returns the index of the thread that should be scheduled next. A null return value indicates that the current thread should be blocked instead of a new thread being scheduled. A detailed description of the approach is presented in [Harris01].

UbiqtOS implementation extended this link-time extensibility approach to allow dynamic adaptation by employing the decoupling leveraged by the Romvets interface. In UbiqtOS implementation, the Kaffe scheduler forwards these activations to the Romvets interface, which routes them to the programs that had registered interest in them. Hence, scheduling policies can be dynamically adapted by replacing the event handlers for these upcalls using the Romvets interface. Further, the extensible scheduler of OSKit also exports itself to layer 2 by registering itself with UbiqDir and supports a JNI-wrapped COM interface that returns the load on the system as the number of processes in its ready queue. This method can be used by agents that implement load-balancing policies, at agent launch time, and the agents that implement effective mobility to balance load in the system. Extensible scheduling in UbiqtOS is shown in figure 8.3. The UbiqtOS distribution comes with two schedulers; a simple round-robin scheduler and a proportional stride scheduler proposed in [Waldspurger95], for soft-real-time scheduling.

8.3.1.2 Optimizations for Fast Event Handling

The implementation of safe application-specific scheduling described above led us to implement an optimization in Kaffe.

Extensions to UbiqtOS are deployed as Java mobile agents, executing inside the Java Interpreter at layer 1. The event-handlers implemented by these extensions, to handle the events generated by extensible services, are subroutines that are called by Romvets to request extensible operation. Where an ordinary subroutine invocation is satisfactory for extensions that implement context-specific distributed operation for UbiqDir and SEMAS, extensibility of low-level services like scheduling and network protocols require special handling.

The optimization resulted from the observation that the cost of a subroutine invocation inside the Kaffe interpreter itself incurs appreciable overhead in the critical path of a scheduling decision. The overhead is due to the time spent in allocating and pushing a stack frame on the interpreter stack, as noted by [Harris01]. The optimizations proposed below reduces this overhead by more than 60%. With this optimization in place, extensible scheduling in UbiqtOS incurs an overhead of less than 9% (for the round-robin scheduler), which is comparable to other systems like [Harris01].

8.3.1.2.1 Optimization in Kaffe

The Kaffe implementation for executing event handlers for the events generated by the Romvets architecture was changed to allow fast-handling of events in order to support extensibility of services like scheduling and network communication that require low-latency.

First, the Kaffe implementation was changed to accept an additional argument from the (native code) calls from Romvets to execute a subroutine inside the interpreter. This extra argument specifies the memory blocks to be used by Kaffe to execute the subroutine. When invoked like this, Kaffe uses these memory blocks to hold the data structures used by the subroutine, instead of requesting Kaffe memory manager to allocate memory.

The Romvets component is pre-allocated a pool of memory blocks (from the Kaffe memory manager) as part of bootstrap sequence. Romvets uses these blocks to allocate memory for the values of the parameters of the events it generates analogous to fast memory management in operating system kernels (e.g. skbuffs). Further, when asked to invoke a subroutine as an upcall to deliver an event, Romvets invokes the event handling routines by passing these memory blocks to Kaffe. Kaffe uses these blocks to execute the specified routine to manipulate the event, instead of incurring the runtime overhead of going through the Kaffe memory manager to dynamically allocate a stack frame.

This optimization is analogous to the interrupt handling scheme used in traditional operating systems, where interrupts are handled either on the running process's stack or on a pre-allocated kernel stack, instead of incurring the runtime overhead of creating a new stack to execute the interrupt handling routine. However, statically pre-allocating

memory blocks wastes precious memory in an embedded device. But, as this scheme is only used for extensible scheduling and protocol stacks, only a small fraction of the available memory is allocated for this purpose (10% in the prototype implementation).

This scheme is only slightly different from the deflated execution environment proposed in [Harris01]. Deflated execution environment proposes the use of a local variable table, instead of using the Kaffe global constant pool table, to speed up the execution. This prevents any local state to be preserved internally by event handling routines which is over restrictive for the UbiqOS model where event handling routines may be required to keep internal state e.g. TCP protocol. Deflated execution environment, therefore, requires inflation (reverting to normal Java stack) to execute more involved code. Whereas, nothing in UbiqOS's scheme limits the operation performed by the event handlers.

Using these optimizations Kaffe can invoke an event handling routine using 11 assembly language instructions on x86 instead of 34 required in the original Kaffe implementation, which is only slightly slower than the deflated execution environment which requires 7 assembly language instructions.

8.3.1.2.2 Optimization in Romvets

As mentioned in chapter 7, Romvets is implemented using the XML tree search operations of UbiqDir. Romvets subscriptions are stored as a separate hierarchy in UbiqDir namespace that is looked-up to deliver notifications to the extensions that had subscribed interest in an event type. This rich model of storing subscriptions in the same way as other resource descriptions allows subscriptions to be parameterized with any number and type of attributes, and supports subscriptions based only on partial knowledge of the event type (like intent-based lookups). However, this introduces the latency of XML tree search in the critical path of every event notification. Where this latency is acceptable for extensions that require partial subscriptions in order to interoperate with other components, extensions to layer 0 components neither require nor afford this latency. Scheduler activations and events from packet drivers are only parameterized with a single well-known argument (scheduler queue, packet type) and need to be handled with minimal latency. For these components, Romvets implements an

optimized subscription. This optimized subscription stores all the single argument subscriptions in a local hashtable, instead of using UbiqDir's store and search operations. This internalized optimization neatly covers subscriptions for scheduler activations and packet handlers and reduces the latency to only a hashtable lookup.

The two optimizations described above provide efficient extensible scheduling and allowed UbiqOS prototype implementation to achieve acceptable performance when supporting extensible protocol stacks as described below.

8.3.1.3 Extensible Protocol Stacks

In addition to scheduling policies, the UbiqOS prototype implements protocol stacks amenable to dynamic adaptation.

To allow adaptation by context-specific software, protocol stacks are removed from layer 0, implemented by OSKit, along with the corresponding support from Kaffe, and the packet drivers for the network interface cards attached to the device are exposed directly to context-specific software at layer 2.

As described in chapter 6, network communication between different instances of UbiqOS is provided by ACP. ACP defines a well-known frame format that every instance of UbiqOS needs to support to be able to transfer agents and exchange messages between different devices. However, as ACP is not a network protocol, it needs to be encapsulated in other network protocols to transport the ACP frames from one UbiqOS host to another. These ACP frames are transported to the destination SEMAS by context-specific network protocols at layer 2 that use the packet drivers, at layer 0, to transmit packets via the network interface cards (NICs) attached to the device. These network protocols are dynamically deployed to suit the characteristics and standards of the current network of the device.

The context-specific network protocols are installed by subscribing to the `transmit(ACP_Frame this, String Address_type, String destination_address)` event generated by ACP to request transmission of an ACP frame to the specified destination. Requests to ACP to transfer a frame specify an agent name as the destination. ACP resolves this address to the network specific address registered as part of the agent

description (enclosed in the mandatory <Address> tag). Protocols on the sending side register interest in the “transmit” event generated by ACP and specify the address type supported by them. On the receiving side, these context-specific protocols subscribe to the events offered by the packet drivers, indicating that a packet has arrived through the network card. These extensions at the receiving side also specify the protocol type that they can handle to allow de-multiplexing of packets arriving at the NIC. The packet drivers at layer 0 inspect the protocol type field (e.g. ethertype in Ethernet LANs) of the incoming packets and forwards the packets to Romvets as events parameterized with the corresponding type. Romvets then forwards these packets to the protocols that had subscribed interest in the event type corresponding to its protocol type e.g. IP would register interest in event parameterized with (string) “0600” to receive packets from an Ethernet packet driver at layer 0.

Therefore, UbiqOS protocols are structured slightly differently from protocols in a conventional operating system like FreeBSD. First, protocols are, indeed, mobile agents, and hence register their descriptions, both functional interfaces and non-functional attributes, in XML with UbiqDir. Second, these protocols register interest in the events generated by ACP and the packet drivers to send and receive packets. Third, protocols themselves generate two special events, using Romvets, “Pkt_trasmit(Byte [] packet, String address_type, String destination address)” to request transmission by lower layer protocols, and “Pkt_Arrived(Byte [] packet, String address_type)” to request processing at the next higher-layer. Protocols register interest in the packets of the corresponding type, e.g. IP, Ethernet etc. to indicate interest in processing the packets. A UbiqOS protocol processes the incoming packet, looks at its protocol type field and forwards it to the appropriate protocol by generating an event with the protocol type argument set to that of the next protocol. The final protocol in the receiving stack generates an event with the protocol address type set to ACP. ACP either launches the agent if the ACP carries an agent or generates a message_received event for the destination agent specified in the ACP frame to deliver the message. An important thing to note in this scheme is the lack of transport layer addressing; ACP frames are addressed to agents by name, and not to a Service Access Point at the transport layer e.g. a UDP

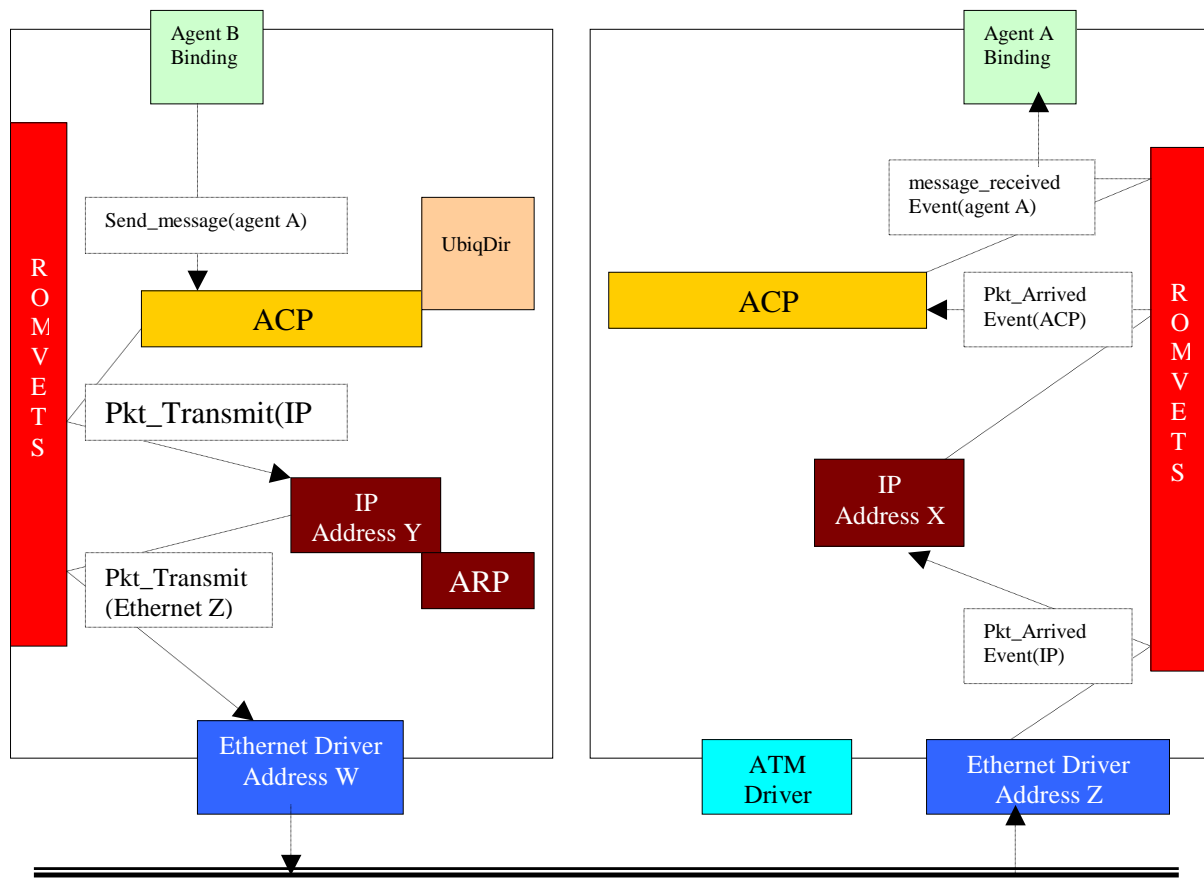


Fig. 8.4 An example protocol stack in UbiqOS.

Different protocols can be dynamically deployed to carry the ACP frames on the network according to the standards and characteristics of the device context. These dynamically deployed protocols run inside the Java Interpreter, and generate events parameterized by protocol type to pass the packets along the protocol stack, using the optimized event-handling support in Romvets.

port number. Therefore communication in UbiqOS is network independent i.e. ACP works independently of which protocol stack is dynamically deployed to transport the ACP frames over the network. This push-style scheme of delivering a message by invoking a message handling subroutine (as an event) of the receiver, as opposed to the pull-style message delivery in traditional operating systems where messages are queued in ports, is similar to the concept of active messages [Eicken92].

Finally, the messages are actually delivered to the explicit bindings, that the agent registers to be notified to deliver the message, and these binding can queue the incoming

messages to allow pull-style processing if need be e.g to enforce flow control. A simple protocol stack configuration in UbiqtOS is shown in fig. 8.4.

8.3.1.4 Connection-oriented Protocols

The lack of addressing at the transport layer and decoupling of network protocols from applications using (connectionless) ACP, however, raises a new problem for supporting connection-oriented network protocols e.g. TCP. Connection-oriented protocols require extra-messages to be exchanged to set-up the connection before any communication can proceed between the two applications.

This problem is handled by explicit bindings. In UbiqtOS, applications use an explicit binding to connect to other agents, by calling the connect method of the binding (mbox). This binding interposes the functionality between the agents to provide the desired quality of service in the face of changing system characteristics.

Connection-oriented protocols in UbiqtOS subscribe to two additional events, “boolean connect(mobex protocol, mbox source, source address, destination_address, connection_paramater_vector)”, and “boolean close(mobex protocol, mbox source)” generated by the binding responsible for the connection to request establishment and tear-down of a connection respectively.

If the service provided by a binding warrants a connection-oriented protocol e.g. ordered delivery, the binding searches for an appropriate protocol in UbiqDir (with an attribute of ordered delivery), and generates a connect event for that protocol, using the Romvets interface, when requested to “connect()” the two agents. Once notified, the protocol can establish the connection with the properties specified in the connection parameter vector and returns a boolean value indicating success or failure. The per-connection state saved by the protocol is indexed with the handle of the binding (mbox) generating the connection request to identify future packets from that binding. As mobets use a new mbox for every new connection, no additional transport level connection identifiers are needed to identify the connection. The connection can be torn down by the binding by generating the “close(mobex protocol, mbox source)” that requests the specified protocol to tear down the connection associated with the binding.

UbiqOS prototype includes both a UDP/IP protocol stack and a “port-less ACP/TCP/IP” suit. Applications use a special TCP_binding to use the protocol stack. This binding acts as a UNIX port (FIFO), allowing the application to read and write byte streams to it. The binding makes ACP frames from the bytestream and sends them to ACP that forwards them as events that are handled by TCP. TCP, that provides the traditional ordered and reliable delivery, encapsulates these frames in TCP segments and forwards them to IP which transmits them using a packet driver at layer 0 as described above.

The performance of the TCP/IP protocol suit is shown in figure 8.5 and figure 8.6. Figure 8.5 shows the achieved bandwidth, with measurements done using a modified version of TTCP benchmark to suit the APIs offered by UbiqOS, with a fixed packet size of 4096 bytes each (52MB total). Whereas, figure 8.6 shows the latency of 1 byte roundtrip time to compare the latency of extensible stacks in UbiqOS with traditional architectures. The performance comparison shows that UbiqOS protocol stack, implemented in Java, using Kaffe and Romvets optimizations, performs almost as good as the protocol stack in OSKit implemented in the native language. However, the performance is appreciably worse than a Linux protocol stack, and to a lesser extent to a FreeBSD protocol stack, due to the modular structure of UbiqOS and the cost of Romvets indirection, in addition to the cost of running protocols as Java code.

8.3.1.5 Adaptation of Protocol Stacks

The framework provided by UbiqOS allows the flexibility to dynamically add and remove protocol layers to suit the characteristics of the context of the device. Traditional systems with fixed protocol stacks (c.f. FreeBSD) are forced to use even “null-protocols” like UDP just to provide a uniform communication framework, whereas communication paths in UbiqOS can be tailored to the characteristic of a particular active space. For example, the protocol stack shown in fig. 8.4 avoids the overhead of a null transport layer like UDP and uses IP directly to exchange messages. This scheme, indeed, requires that

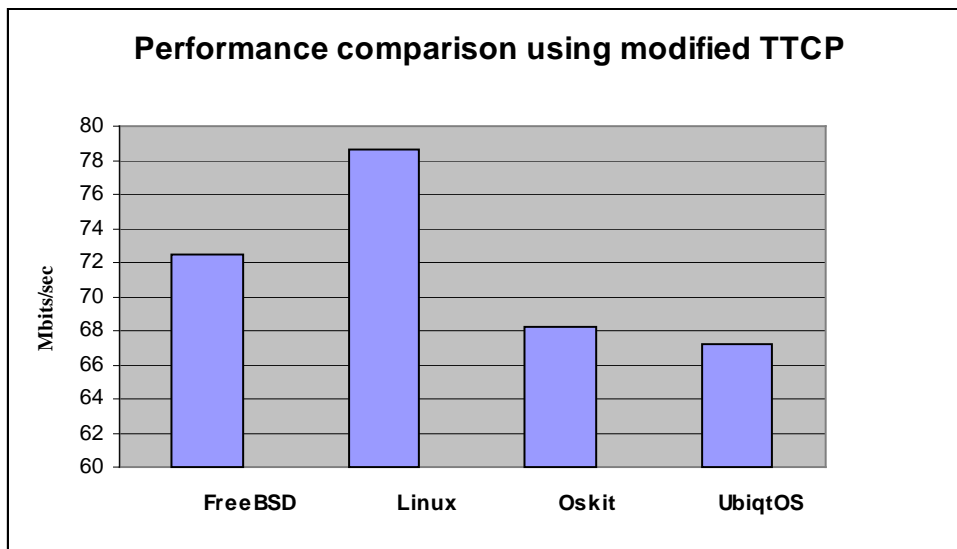


Fig. 8.5 Performance evaluation of the bandwidth of the UbiqtOS TCP/IP protocol stack with traditional architectures. Evaluation was done using two 200 MHz Pentium PCs, connected by a 100 Mbit/sec Ethernet

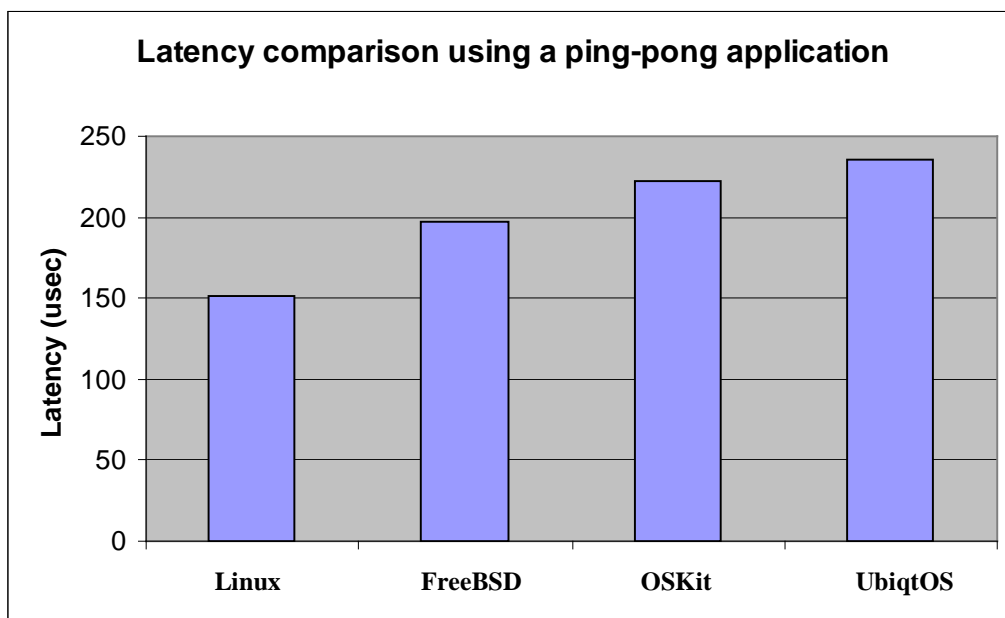


Fig. 8.6 TCP one-byte roundtrip time measured with *rtcp* to compare latency of UbitOS extensible protocol stacks with traditional architectures. Evaluation was done using two 200 MHz Pentium PCs, connected by a 100 Mbit/sec Ethernet

the receiving host be configured with the same protocol stack as the sending host before any communication can take place. This is done at the bootstrap stage when all hosts in a particular context are configured with the right set of protocols for a particular context. Chapter 9 shows how protocol stacks are adapted to provide flexible communication support for mobile hosts.

8.4 Implementation of UbiqDir

Our UbiqDir is implemented using an XML parser by SUN Project X to implement the register, update, delete and lookup operation. However to support a working prototype, UbiqDir implementation needs to support default extensions that implement its distributed operation.

8.4.1 Default UbiqDir Extensions

UbiqtOS distribution comes with three default extensions for UbiqDir to illustrate the flexibility of the architecture. These extensions implement two discovery protocols using different overlay topologies and employ different consistency protocols for different resource descriptions. The default UbiqDir distribution is shown in figure 8.7.

The distribution supports both an IETF Service Location Protocol to allow interoperability with IP-based networks and the in-house DHAN discovery protocol, presented in Chapter 2, to support interoperability with the AutoHAN system. SLP operates in an active mode, broadcasting all the queries on the local Ethernet using the SLP multicast address and supports eventual consistency with its periodic soft-state refresh messages every 30 sec. On the other hand, the DHAN discovery protocol uses HTTP messages to route all the queries to the central DHAN server running in the network to enable discovery of resources in the AutoHAN system. Both these extensions use the UDP/IP protocol suit in the UbiqtOS distribution.

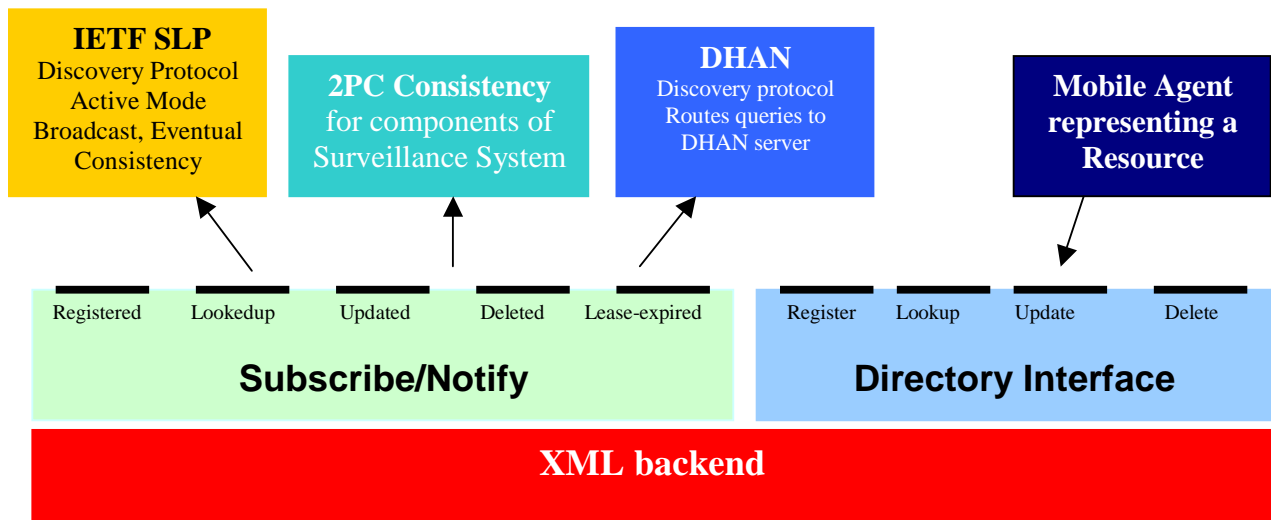


Fig. 8.7 UbiqDir default extensions ensure that UbiqOS devices can interoperate both with IP-based systems and AutoHAN and allow strong consistency for security critical information.

However, both these protocols only support eventual consistency, not guaranteeing that any changes in the system would be visible to all the devices in the system. Where this is acceptable for most applications in a dynamic network, the distribution includes another extension that uses two-phase-commit to guarantee strong consistency for selected resource descriptions. This extension ensures that any changes in the home surveillance system (resources with `<surveillance>` tag as part of their description) are committed to all the participating devices in the system by the DHAN server to ensure that security critical information is disseminated reliably in the system. This extension uses 2PC on top of HTTP to disseminate information from DHAN to UbiqOS devices.

8.5 Implementation of SEMAS

SEMAS executes as a thread in Kaffe, part of layer 1. In the default implementation, the default IP agent part of the UbiqOS distribution subscribes to the events generated by SEAMS Agent Communication Protocol to transmit and receive frames on the network. The structure of this protocol stack is shown in figure 8.4; notice the lack of a transport

layer. Agent bindings register interest in the events generated by ACP to receive messages (instead of using a transport layer identifier).

A working prototype of SEMAS requires extensions at layer 2 to interpose functionality between SEMAS and ACP to enable transmission of agents according to characteristics of the network and the agent requirements. UbiqtOS default distribution includes the following SEMAS extensions.

8.5.1 Default SEMAS Extensions

Default implementation of SEMAS includes three extensions to allow transfer of agents and to balance load in the system, as shown in figure 8.8.

The default distribution of UbiqtOS includes two extensions that serve to transfer agents with different reliability semantics. One of these extensions supports disconnected operation by allowing agents to be queued if they cannot be migrated at the time of the request due to an intermittent connection or temporary destination host failure. This queue is then served in a round-robin fashion, trying to migrate every agent at 30 sec intervals, for a maximum of 10 times, after which the agent is discarded and a false value is returned. Additionally, the queue is kept sorted according to agent destinations so that if a link becomes usable all the agents queued to use that link can be transferred consecutively. Finally, queue size is fixed to 10 in the prototype implementation based on practical experience; if a device cannot honor 10 migrations requests then it is disconnected from the network for a longer time, and this fault should be exposed to applications. Hence, when the queue fills up, further agent requests are denied migrations instantaneously instead of deferring it for later.

However, this extension provides only at-least-once reliability guarantees at the best. Agents that require exactly-once-reliable migrations are handled by the second extension in SEMAS distribution. This extension implements a transactional queue, proposed in [Rothermel98], as described in section 6.7.6. The extension returns a false value to the requesting agent if the transaction fails, leaving the recovery to the agent itself.

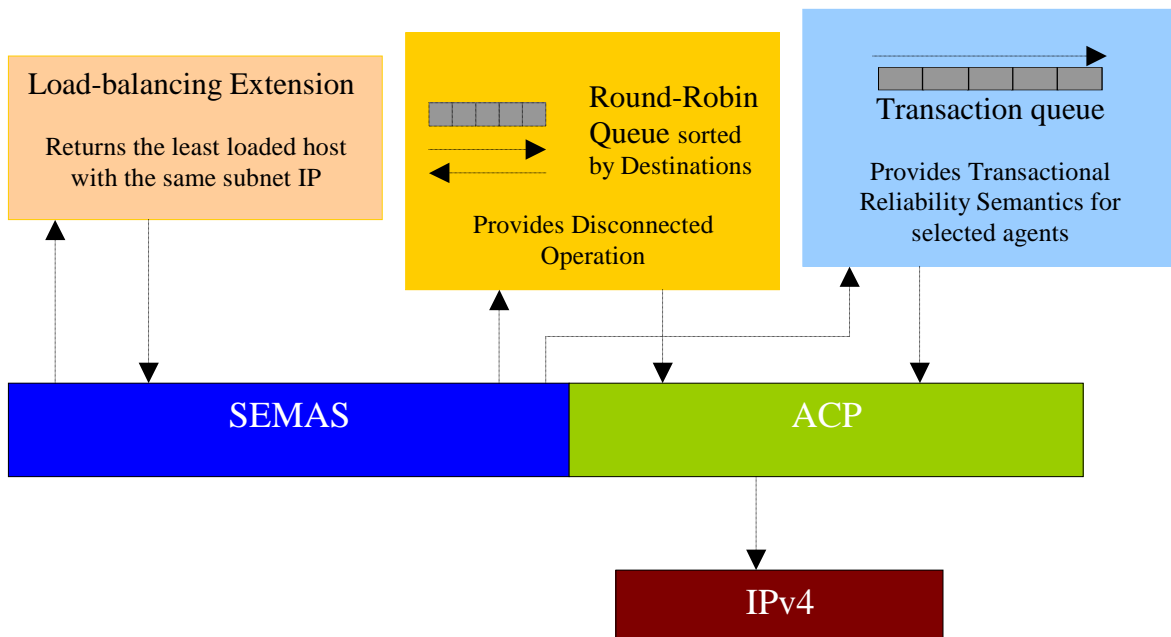


Fig. 8.8 Default extensions for SEMAS.

The default distribution of SEMAS uses IPv4 to transfer agents between different hosts and includes three extensions to provide load-balancing, disconnected operation and reliability.

Finally, the SEMAS distribution includes an extension that returns the least-loaded host on the same network. This extension subscribes to the events generated by SEMAS to request bootstrap load-balancing and effective mobility. When invoked, it finds the least loaded host by broadcasting the length of the ready queue of the local scheduler at layer 0 (by invoking its `Current_load()` method) with the IP destination address set to local network. Other extensions on the network listen to this message and reply with the length of their ready queue (and a handle of their SEMAS) only if the length of the ready queue on their host is less than that of the received message. The extension collects the result and returns the handle of the SEMAS with the shortest ready queue.

8.6 Default Distributed Services

In addition to the extensions to SEMAS and UbiqDir, UbiqOS's default distribution includes a dispatcher module, which when invoked makes default selection if more than one resource provides the requested service. Additionally, it includes four extensions for providing default support for load-balancing, fault-tolerance and high-availability, making UbiqOS a complete distributed operating system. The components included in the default distribution of UbiqOS are shown in figure 8.9. Applications discover these services and invoke their methods using their descriptions in UbiqDir.

These extensions take the context of the device as all the devices on the same IP subnetwork, like traditional distributed operating systems.

8.6.1 Default Dispatcher

The dispatcher module implements the context-specific view of the resources registered with UbiqDir. When invoked with (an intent-based) description of the required service, the dispatcher looks-up all the matching resources in UbiqDir, and selects one according to its (context-specific) policy. The default dispatcher provided with the UbiqOS standard distribution implements the simple policy that if there are more than one resources providing the requested service in a subnet, then select the one in the current room. It does that by augmenting the service invocation request with the location attribute set to the current room. This implementation assumes that every resource exports a location attribute along with its description registered with UbiqDir, acquired by a location service in the active space e.g. active badges [Want92] (used in our lab), Cricket Location System [Priyantha00] etc. The dispatcher contacts the location service to find out about the location of the device. The protocol to contact the location service is expected to be known to the Dispatcher as it is deployed in the device by the context itself. In the implementation, the Dispatcher contacts the DHAN service in Trojan room (the AutoHAN test-bed room, using 802.11 Wireless LAN) with an HTTP query to find "location" and the DHAN service responds with the string "Trojan Room".

8.6.2 Default Extensions for Load-Balancing

In addition to load-balancing at agent bootstrap, and as part of effective mobility, the UbiqtOS distribution includes two extensions to support reactive load-balancing. Reactive load-balancing is implemented by monitoring the load on the local host and requesting agents to move to another host when load exceeds a threshold.

The default extensions monitor two different metrics; processor utilization and memory utilization. One extension calls the `Current_load()` method of SEMAS every 30 sec to measure load on the local host, and normalizes the value by total length of the process queue length to measure process utilization. If this average exceeds 0.8 (80% load), the extension picks a random agent from the list of agents executing with SEMAS (by calling its `List_Running_Agents()` method) and generates the `Balance-load()` event (the same event SEMAS generates to elicit suggestions for load-balancing in effective mobility). In the default implementation, this event is handled by the default load-balancing extension of SEMAS described above, which returns the least-loaded host in the same network. The reactive load-balancing extension then requests SEMAS to migrate the randomly picked agent to the least-loaded host in the network. It is up to the agent being requested to honor or deny the request, which either saves its state, specifies the method to be called at the destination host to resume execution and returns a true value to allow SEMAS to perform the migration, or returns a false value to deny the request. If the agent denies migration, perhaps because it is in the middle of a computation and cannot move, the reactive load-balancing agent picks another agent and repeats the process until an agent agrees to migrate to another host or the local load falls below 80%, by some agent moving by itself or committing suicide after finishing its task.

The other default load-balancing extension part of the UbiqtOS distribution triggers migrations if the local memory utilization exceeds 80%. Traditional distributed operating systems do not implement such load-balancing as they are designed for hosts with large secondary memories to support large virtual memories. Embedded devices, however, have limited or no persistent storage to support virtual memory. Hence, UbiqtOS needs to balance memory utilization in the system.

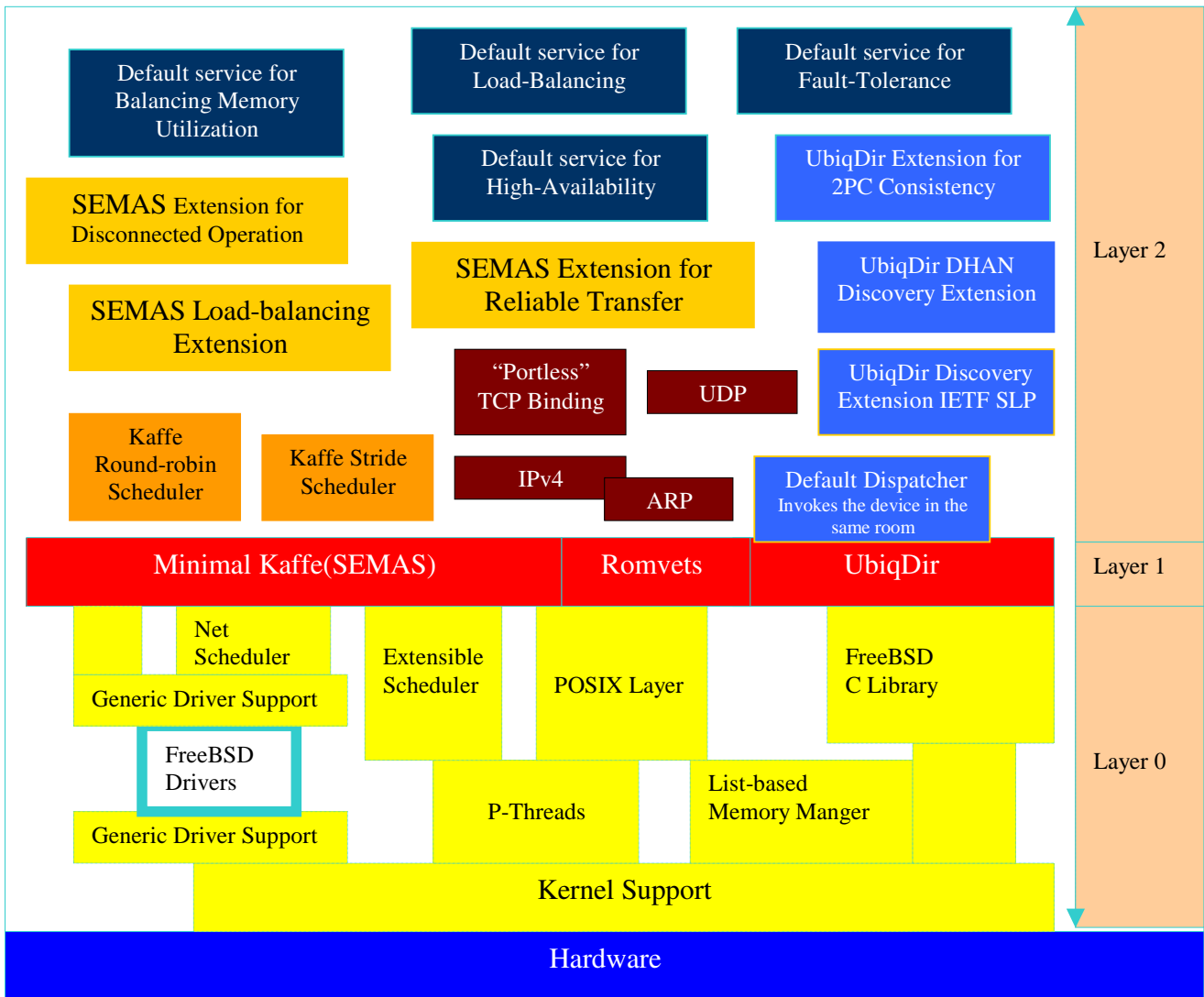


Fig. 8.9 Architecture of standard UbiqOS distribution.

In addition to the OSKit components to implement hardware dependent layer 0, and core SEMAS, Romvets and UbiqDir at layer 1, the standard distribution includes the following mobile agents at layer 2. 1) IP, ARP, UDP and “portless” TCP protocol suit 2) Two default schedulers; a round-robin scheduler and a proportional share stride scheduler 3) default extensions for SEMAS to provide disconnected operation, load-balancing, and reliability 4) default extensions for UbiqDir for discovery on SLP system, discovery on AutoHAN networks and 2PC consistency 5) Default services to support reactive load-balancing, fault-tolerance and high-availability.

The memory-balancing extensions monitors the memory utilization of the local host by subscribing to the “memory-warning()” event generated by the layer 0 list-based memory manager used in UbiqOS implementation. On receiving this event, this extension

contacts its peers on the same subnet and finds the host with the least memory utilization, just as the load-balancing extension finds the least-loaded processor. Having found the host, it chooses a random agent among the ones executed by SEMAS and requests it to be migrated to the host with maximum available memory, and repeats the process until one of the agents agrees to migrate or the memory utilization falls below 80 %.

8.6.3 Default Extension for Fault-Tolerance

The UbiqtOS distribution also includes a default extension to provide fault-tolerance for services that need to be highly available in the system. Agents can register with this service by calling its `Fault_tolerance(mobet source, int K)` method, where K is the degree of resilience they require. The extension talks to its peers on the same subnetwork, asks them the load on their hosts, and chooses $(K-1)$ least loaded nodes in the network. The extension then requests SEMAS to replicate the requesting agent on the selected $(K-1)$ nodes and informs its peers about the decision. Once the agent is replicated, the default fault-tolerance extensions on the selected hosts generate a heartbeat for the requesting extension, at 1-minute intervals in the default implementation. If the requesting extension, acting as a master in this case, does not receive a heartbeat message from a host for more than 1 minute, indicating that the host has become inaccessible, it selects another host in the subnetwork and asks SEMAS to make another copy of the agent. This simple scheme ensures that there are always K copies of the agent running in the system. However, the master device itself could fail or move from one network to another one. This is handled by the master extension sending a heartbeat to all its slaves at 3 minutes intervals. If the slave does not receive the heartbeat for more than 3 minutes, it asks its SEMAS to `kill()` the replicated agent. SEMAS in turn invokes the `Die()` method of the agent. It is up to the agent to honor or deny the request, depending on whether it would like to linger in the network or not in the absence of the master service. If the master extension misses three consecutive heartbeat messages from its slaves, it assumes that it has moved to another subnet and restarts the whole process of selecting $(K-1)$ least hosts and replicating the agent to provide fault-tolerance on the new subnet.

8.6.4 Default Extension for High-availability

Like the fault-tolerance extension, the standard distribution of UbiqtOS includes a default extension that leverages hi-availability of interested services. Agents register with this service by calling its `Hi_availability(mobet source, int number_of_requests_per_minute)` method. Once registered with it, the extension monitors the incoming requests to the agent, by subscribing interest in the messages it receives with `Romvets`, and takes a weighed moving average of the number of messages received by the agent over the last 5 minutes (with equal weights of 0.5). If this average increases the value specified by the agent, the extension selects the least-loaded host on the same subnet and requests SEMAS to replicate the agent on that host. Therefore, if the service provided by the agent gets overloaded, the system makes another copy of the service in the network to share some load. Given that services are discovered by intent and not by location in UbiqtOS, this scheme effectively leverages hi-availability of services that are not tethered to some embedded hardware resource. The use of an epoch-based scheme to measure the load on the service avoids reacting to temporary load spikes.

8.7 Conclusion

This chapter described the implementation of a UbiqtOS prototype using OSKit and Kaffe. It described how the implementation of UbiqtOS architecture led to optimizations within Kaffe and Romvets to support scheduling and network communication at layer 2 and described the implementation and evaluation of extensible scheduling policies and protocol stacks. Moreover, it described the implementation of default extensions for SEMAS and UbiqDir to produce a working prototype system. Finally, it presented the implementation of default extensions to provide load-balancing, fault-tolerance and high-availability in the system as an example of extensible distributed operation leveraged by UbiqtOS architecture.

The prototype implementation shows that it is practical to support traditional distributed operating system services atop an (optimized) interpreter to allow safe, dynamic,

application-aware and context-specific adaptation to address the challenges posed by a ubiquitous system design.

However, this chapter did not present an evaluation of the UbiqtOS design. The issue was deliberately avoided as evaluating a novel bottom-up architecture like UbiqtOS is a topic in itself and forms the basis for the next chapter.

“There are lies, damned lies, and statistics”

Mark Twain

Chapter 9

Evaluation

This chapter presents the performance evaluation of different components of the prototype implementation and gives examples of how UbiqtOS can be used to efficiently support novel applications in a ubiquitous system.

Two different applications are presented and evaluated to illustrate the flexibility and performance of context-aware adaptation in UbiqtOS.

9.1 Evaluation Methodology

Traditional operating systems are often evaluated purely in terms of quantitative performance e.g. round-trip RPC across protection domains, network communication throughput etc. Though such traditional benchmarks meaningfully indicate the efficiency of operating systems designed to provide traditional services, they, alone, are not sufficient to evaluate the different aspects of UbiqtOS. As UbiqtOS is a language-based, embedded, extensible distributed operating system, the benchmarks chosen to validate its design need to evaluate all these aspects. Additionally the evaluation methodology needs to evaluate the efficacy of dynamic context-aware adaptation in UbiqtOS. However, context-aware adaptation offers benefits other than performance; the flexibility and dynamism offered by UbiqtOS addresses new problems. Finally, there are no well-known traditional benchmarks to evaluate adaptive systems.

Hence, the performance evaluation of the prototype implementation of UbiqtOS is divided in two parts:-

- The first part evaluates the performance of the prototype implementation to enable comparison with traditional systems, using the following:-
 - Code size of the prototype implementation to evaluate suitability for embedded devices
 - Cost of installing, deleting, upgrading and looking up components in UbiqtOS (using UbiqDir)
 - Latency of notifications to indicate changes in the context using the Romvets Interface
 - Cost of invoking a system call using the Dispatcher module
 - Cost of inter-component communication
- In the second part, context-aware adaptation in UbiqtOS is evaluated using two new applications made possible by UbiqDir architecture.
 - Adaptation of context-aware bindings proposed in chapter 6 is evaluated using a soft real-time multimedia application to provide a “follow-me-video” service.
 - Adaptation of network protocol stacks implementation presented in chapter 8 is evaluated using flexible support for mobile IP proposed in [Zhao98].

9.2 Performance Evaluation of UbiqtOS Components

9.2.1 Code size

UbiqtOS is designed for medium to high-end embedded devices and hence needs to be small enough to be embedded in the ROM of such a device.

Most existing embedded operating systems are customized to the fixed functionality provided by proprietary device hardware architectures and, hence, do not provide a good

Layer 0	112.3 KB
Oskit	67.2 KB
Kaffe	45.1KB
Layer 1	23.2 KB
SEMAS + ACP	9.5K
UbiqDir + Romvets	13.7K
Layer 2	26.4 KB
Default System Dispatcher	2.3K
Default scheduler	6.7K
Protocol stacks	6.1K
UbiqDir Extensions	5.7 K
SEMAS extensions	6.4 K
Default Distributed Services	5.9 K
Total	168.6KB

Table 9.1 Code size of different components in UbiqtOS prototype

reference to compare the code size of UbiqtOS, which is a general purpose operating system designed to execute applications dynamically introduced in the device. Second, traditional embedded operating systems, designed for standalone, dedicated pieces of hardware, do not provide distributed services.

UbiqtOS's runtime extensible architecture allows it to be embedded in limited capability devices and scaled to more privileged devices to address the heterogeneity of the system. Additionally, layer 1 in UbiqtOS supports context-aware distributed operation to address the heterogeneity, mobility and dynamism of the system.

Table 9.1 shows the breakdown of the size of the fixed part of the UbiqtOS prototype that needs to be embedded in the ROM of a device. All of the code for layer 0 and layer 1 has been compiled to statically linked native code (.ar), instead of position independent shared libraries to reduce code size and to improve performance. Code for layer 1 was written in Java and compiled to native code using gcj.

Figure 9.1 compares the code size of UbiqtOS standard distribution with WinCE [Murray99], Embedded Linux [Linux] and QNX [Hildebrand94].

The size of embedded Linux is essentially a Linux kernel and glibc, without X11 libraries and command shells. However, not designed for embedded devices, its code size is significantly larger than WinCE. However, WinCE, like embedded linux, includes a

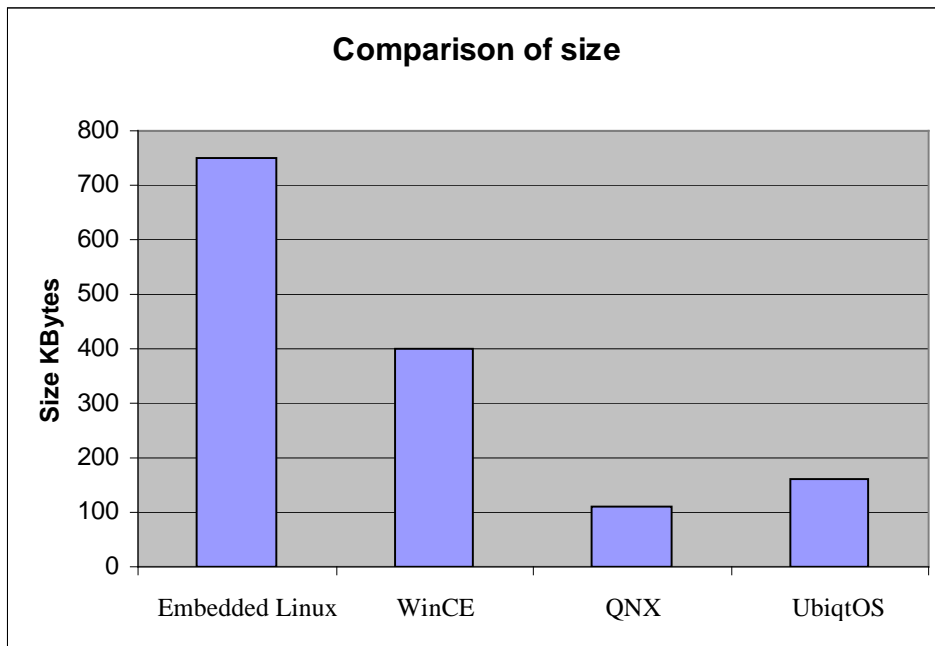


Fig. 9.1 Comparison of ROM Image of UbiqtOS with embedded linux, winCE and QNX operating systems on x86.

filesystem and its windows-based GUI support not needed in an embedded device. QNX, on the other hand, is a commercial operating system designed for embedded devices. Its build-time extensible microkernel does not include a filesystem, GUI or even network protocol stacks. These services are optional and can be included at build-time. The code size shown in figure 9.1 includes protocol stacks but not filesystem or GUI. UbiqtOS size, even with a Java Virtual Machine and distributed services is comparable to QNX. The small code size of UbiqtOS is primarily because of the better code density of Java bytecode. In UbiqtOS most of the components that are inside a traditional kernel are pushed atop layer 1 as Java byte-code which is more dense than x86 native code.

9.2.2 Cost of extension using UbiqDir

New components are installed with an instance of UbiqtOS by registering their functional interfaces and non-functional attributes in XML with UbiqDir. Components find one another by looking up these descriptions in XML and retrieving a reference to their interface to invoke operations on one another (using Java reflection API).

<i>Entities registered with UbiqDir</i>		
<i>Operation</i>	<i>10</i>	<i>100</i>
Update	1.3	2.1
Lookup	3	4.7
Register	2	3.2
Unregister	0.9	1.2

Table 9.2 Cost of UbiqDir Operations.

Table 9.2 shows the average cost, in milliseconds, of registering, deleting, updating and looking descriptions with UbiqDir. The costs, indeed, depend on the number of attributes being looked up, registered or updated, the costs shown are for average components in UbiqOS with 7-10 attributes. The costs were measured by running a trace for 100 simulated devices, and taking an average of the times for different operations.

UbiqDir uses a modified version of XML parser by Java Project X, which uses breadth-first search to search the XML tree. The parser was modified to optimize the update and delete operation with the ID-based optimization described in chapter 7. The cost for these operations shown in table 9.2 include the generation and checking of indexes to authenticate and speed up the update, and unregister operations. The measurements were made on Intel Pentium 200 MHz PC.

The cost of these operations compares favorably to other directory services like SSDS [Czerwinski99] and INS [Winoto99]. SSDS uses the XSET parser to search XML trees, and takes, on average, 8.9 msec to perform an XML search, and as much as 82 msec to respond to a secure-RMI query [Czerwinski99]. XSET uses treaps (probabilistic self-balancing trees) to store the XML trees to allow efficient searches, but that distorts the view of the natural hierarchical distribution of resources in the tree. In UbiqDir, the register function inserts a new resource at its “right place” in the hierarchy, e.g. a video/movie_camera would be placed as a sibling to video/still_camera under the root “video”, producing a very user-friendly view of the state of the system to allow introspection. Still, UbiqDir outperforms SSDS as it has been compiled to native code, as opposed to SSDS that runs as interpreted Java code.

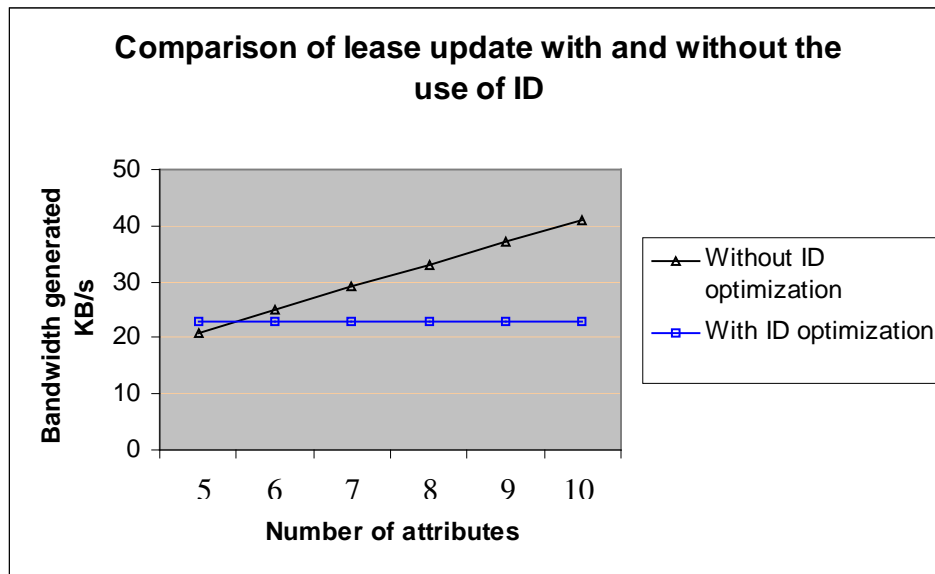


Fig. 9.2 Comparison of lease traffic for lease update with and without the ID optimization. The traffic generated by ID-based scheme remains fixed, while the traffic generated by transmitting the whole description of the resource to update its description increases with the number of attributes.

INS, instead of using XML, uses its own data format and tree organization algorithm, called graft, to allow highly optimized searches [Winoto99]. INS can perform, with 3 on average 800 lookups/sec with 3 attributes in the query, which is comparable to the performance of UbiqDir.

Further, SSDS and INS use the same search methods for updating the description of the resource as well. In a soft-state system, where updates are made periodically at short-intervals, the network traffic generated by update messages dominates the performance of the system [Winoto99]. The ID-based search algorithm also improves performance by more than 50%, as compared to an intent-based lookup, as shown in table 9.2.

Further, UbiqDir's use of ID-based updates, reduces the network traffic to an ID plus the attributes that need to be changed in the general case, or just an ID to refresh the soft-state entry. Figure 9.2 shows the network traffic comparisons of a 10-node simulation, with and without the ID-scheme, as the number of attributes, encoded as 16-bit Unicode characters, in the resource description are increased from 5 to 10 each assumed to be 20 bytes each. The lease-interval is fixed at 5 sec (instead of 30 sec to illustrate the benefit of the ID scheme), with the protocol used to transport messages is assumed to incur an

overhead of 27 bytes per refresh message. The nodes exchange descriptions of 10 resources at every 5 seconds to refresh their entries.

9.2.3 Cost of Adaptation using Romvets

Components installed with an instance of UbiqOS lend themselves to adaptation by generating events (upcalls) that are handled by extensions to implement their behavior according to the requirements of the applications and the characteristics of the context of the device.

Romvets is implemented using UbiqDir and its lookup, register, delete and update operations are used to implement the notify, subscribe, delete and update operations of the Romvets interface respectively.

The Notify operation involves lookup of any matching event-handlers and their invocation. Subscribe is just another interface for the register operation in UbiqDir, which allows subscription of events by registering the XML description of the desired arguments of the event. Likewise, delete and update interfaces for Romvets are implemented by delete and update operations of UbiqDir.

Table 9.3 shows the cost, in milliseconds, of subscription and notification of events using the Romvets interface. The costs, indeed, depend on the number of arguments to be subscribed, registered or updated, the costs shown are for average event-handlers in UbiqOS with 2-3 arguments.

However, the cost of notification is critical for high-performance components like scheduling and networking protocol stacks. However, it is slower by almost a factor of 2 from the SPIN dispatcher, which routes the upcalls in SPIN operating system [Bershad95]. This results from the high cost of an XML-tree search within the critical path of notify, as well as the time spent by Kaffe to allocate and push a stack frame to execute the event handler. These two problems were addressed by the optimizations in Romvets and Kaffe presented in chapter 8. These optimizations store single attribute-based subscriptions, like those for network protocols and scheduler, in a hash-table, much like a protocol switch table in FreeBSD, and avoid the cost of

<i>Subscriptions stored by Romvets</i>		
<i>Operation</i>	<i>10</i>	<i>50</i>
Subscribe	0.9	1.5
Notify	1.02	1.53

Table 9.3 Cost of Romvets operations (in msec) for the un-optimized case

<i>Subscriptions stored by Romvets</i>		
<i>Operation</i>	<i>5</i>	<i>10</i>
Subscribe	135	142
Notify	154	159

Table 9.4 Cost of subscription and notification (in μ sec) using the optimizations in Kaffe and Romvets.

allocating a new stack frame for every invocation. Table 9.4 shows the improvement in the performance using these two optimizations (cost is shown in microseconds).

This allows acceptable performance for extensible scheduling and protocol stacks, as shown in chapter 8.

9.2.4 Cost of System Call using Dispatcher

The system call interface is provided by the dispatcher module in UbiqOS. Dispatcher looks up the components matching the description of the service requested, in UbiqDir, makes a selection according to its policy, and invokes its pertinent methods using the Java reflection API and returns the results to the requesting application. The component can be local or remote; the dispatcher presents a single view of the distributed system.

A system call using a trivial dispatcher that invokes the first matching service and returns the result takes, on average, 2 milliseconds. However, this can be improved by internal caching by the dispatcher. An optimized version of dispatcher of UbiqOS caches the components recently looked-up using probabilistically balanced trees (treaps), allowing system calls to be dispatched as fast as 300 μ sec. Note, system calls in UbiqOS are

different from the system calls in traditional systems where services are well-known and fixed, and do not need to be dynamically looked-up at runtime.

9.2.5 Inter-component Communication

Security in UbiqtOS is based on dynamic type safety of the Java Language and runtime system. Further, components in UbiqtOS are modeled as passive objects i.e. components do not block on a thread to wait for messages. Instead they are either looked-up in UbiqDir and invoked by making a subroutine call on their functional interface or they are executed in response to an event generated by Romvets as event handlers. Hence, intercomponent communication, given a handle to an object, is merely a Java subroutine call. However, interaction between components at layers 0,1 and layer 2 incurs the overhead of the Java Native Interface. This cost is critical for event handlers for systems like virtual memory, scheduling and network communication. The optimization implemented in Kaffe, by allowing execution in pre-allocated blocks, reduces the cost from 34 assembly instructions, needed to allocate a frame and move operands between the Java and the native stack, to 11 instructions only.

Table 9.5 shows the cost of a null round-trip-RPC in UbiqtOS both within the same layer and between different layers. Performance measurements were done using PII 266 to compare with one of the fastest micro-kernels L4 [Hartig97]. Software protection provided by Java runtime environment, as opposed to hardware security domains, speeds up inter-component interaction by a factor of 10 for components at layer 2, and, with Kaffe optimization in place, by a factor of 2 between different layers.

However, if an agent makes an explicit connection with another agent, instead of making a default invocation using the Dispatcher artifact, the interposition of the binding adds to the cost of communication. The cost, indeed, depends on the functionality provided by the binding.

L4	242
UbiqtOS inter-layer	97
UbiqtOS layer 2	23

Table 9.5 Roundtrip RPC in the same address space. Cost measured in cycles on PII 266.

As an example, the cost of a simple FIFO binding, a binding that allows pull-style message delivery like a UNIX port, is two synchronized subroutine invocations; one subroutine call to enqueue the message in the FIFO and the other by the agent to dequeue the message.

9.3 Evaluation of Context-ware Adaptation in UbiqtOS

UbiqtOS enables dynamic, application-specific, and context-aware adaptation to address the challenges posed by a ubiquitous system design. The following two sections present two different applications to demonstrate and evaluate adaptation in UbiqtOS.

9.3.1 Follow-me-video Binding

The Follow-me-video application allows the user to carry his, UbiqtOS enabled, DVD player from one room to another and automatically display the video stream played by the player on the nearest screen available, instead of showing it on the tiny display built-in the device.

The use of explicit bindings in UbiqtOS allows the application to be structured, and deployed, using two components.

- An application (mobet) that fetches the video stream from the DVD player and passes it to the follow-me-video binding.
- A follow-me-video binding (mbox) that chooses the correct display to show the video stream according to the protocols and policy suitable for the characteristics of the surrounding system.

Therefore, the application agent just requests that it requires a follow-me-video connection with its context and the binding decouples it from details of, possibly changing, characteristics of the underlying system to provide context-aware adaptation.

Figure 9.3 and 9.4 show the relevant code snippet for the application agent and the follow-me-binding respectively.

The application code performs the following operations. It first looks up the two bindings; the Follow-me-Binding to connect to the display device driver (a mobile agent as well) of any nearby display device and a null binding, that just invokes the specified method and returns the results, to connect to the local DVDplayer device driver. It then looks-up the mobets representing the display device and the local DVDplayer and calls the SEMAS `do_Task` method to request the task. To this method, it passes its reference, the reference to the display mobet and the binding to be used. The mobile agent engine calls the `connect` method of the specified binding to request connection of two agents, allowing the binding to bootstrap itself. It then calls the `displayVideo` method specified in the `doTask` invocation. The `DisplayVideo` method reads the video frame by frame, from the DVD binding, and sends it to the display driver binding to be rendered on the display device.

So far, this just demonstrates how things are dynamically discovered and used in UbiqtOS.

The context-aware adaptation is performed by the binding. Its `connect` method, when invoked by SEMAS in response to a call to `doTask()`, first checks whether a counterpart binding exists on the destination host or not. If it does not then the binding requests itself to be replicated on the destination host and requests its `prepare-receive` method to be called at the destination host to indicate to its replica that it is the receiving side. This method, not shown in the code, registers itself with Romvets to receive ACP frames from the sending binding. Additionally, the `connect` method requests to be notified, by the Romvets interface, if any display device joins the device context. Consequently, the “Registered()” event handler implemented as part of the mbox interface is invoked by Romvets if any display device becomes accessible in the device context. Once notified,

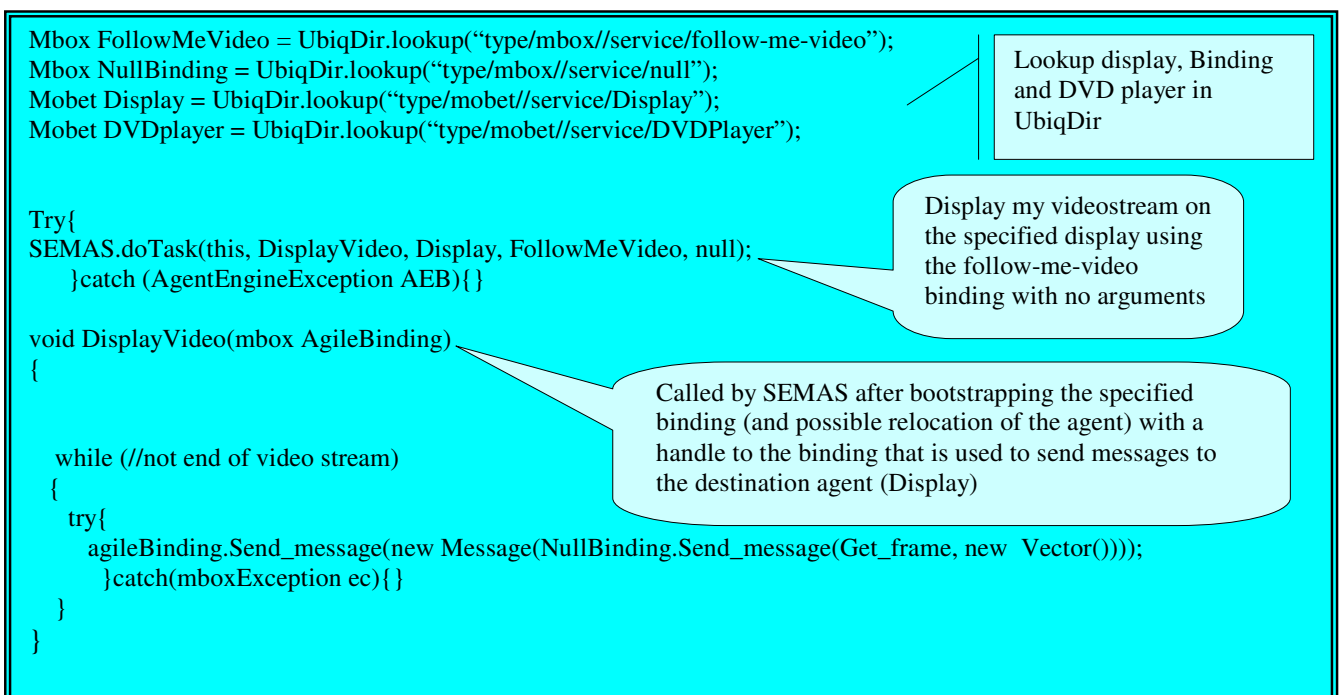


Fig. 9.3 Code snippet from Follow-me-Video Application

```

// class data

mobet source
mobet destination

Boolean Connect(mobet src, mobet dest, Vector arg, mobet suggestion )
{
    source = src;
    destination = dest;
    // if the binding does not exist on the destination host
    SEMAS.Replicate(this, this, destination, prepare_receive);
    UbiqDir.Subscribe("service/display", "Registered", this);
}

Boolean Send_message(Message msg)
{
    Message message = msg;
    ACP.message_transfer(destination, source , message);
}

void Registered (String resource)
{
    // if the resource is a display
    // look at its latency
    // if latency is less than current latency
    mobet newDisplay = UbiqDir.lookup (resource)
    this.Connect (source, newDisplay)
}

```

Called by SEMAS to allow bootstrap of the binding. The binding replicates its receiving side on the destination and subscribes interest to be notified if and when a new display becomes available

Called by UbiqDir (via Romvets) to notify when a new display becomes accessible. Upon notification, it checks the latency to access the new display and if it is less than that of the current display it rebinds to the new display

Fig. 9.4 Code snippet from Follow-me-Video Explicit Binding

Time to replicate (1.4KB bytes in size)	11.3
Context change notification	1.3
Time taken to change to a new display (without a binding) after it becomes visible	59.4

Table 9.6 Cost of Adaptation in Follow-me-Video Application

the binding checks the latency to access the new device, as exported by a dynamic attribute of the device description, and if the latency of the newly registered device is less than the latency of the currently connected display device, the binding connects the DVD player to the new device.

Table 9.6 evaluates the context-awareness of the binding by giving a breakdown of the time spent, in milliseconds, taken by a prototype implementation of the application. The performance measurements were done on Intel 200 MHz machine using its 1MB/sec IrDA connection.

9.3.2 Flexible Network Support for Mobile devices

Chapter 8 described how protocol stacks are supported at layer 2 in UbiqOS to allow context-aware adaptation of network communication. This section demonstrates how this flexibility in UbiqOS design can be used to provide efficient network support for mobile devices, proposed in [Zhao98].

Zhao et al argue in [Zhao98] that a single, fixed solution like the triangle routing in IETF mobile IP [Perkins97] does not efficiently solve the problem of packet delivery for mobile hosts due to following reasons:-

- Mobile IP is not needed for short-lived connectionless applications that are unlikely to move to another network midst an interaction e.g. web browsing. Hence, devices that do not have any alive connections to other hosts, or that do not provide a service on a well-known IP, do not need to incur the overhead of triangular routing used by IETF Mobile IP [Perkins97]. Instead, they are better off with normal IP layer configured with a local IP address.
- If the network is connected via secure routers that do source IP address filtering for incoming packets, then the mobile device cannot send packets with the source address

set to its home agent, and, hence, triangular routing used by IETF Mobile IP cannot be used. This problem requires support for bi-directional tunneling at the network layer [Montegnero96].

- The applications that need to join any multicast IP groups can either join them transiently using a local IP address or permanently by using the home IP address and paying the price of triangular routing, depending on their requirements.
- Finally, the networks that support route optimizations for Mobile IP can use route optimized mobile IP to avoid the overhead of triangular routing, which would not be possible with a fixed IETF mobile IP stack.

UbiqtOS solves this problem by allowing a suitable IP layer to be deployed according to the application requirements and the characteristics of the current network of the device.

As the device joins an active space, a configured device on the network replicates the appropriate network layer as part of the device bootstrap described in chapter 5. The network layer installs itself in the device by subscribing to the events offered by ACP and the device network drivers.

Table 9.7 shows the breakdown of the cost, in milliseconds, of replacing IETF Mobile IP installed with a device with a version that supports bi-directional tunneling as the device moves to a network connected with a secure gateway equipped with source IP address filtering. The performance measurements were done on Intel 200 MHz machine using its 1MB/sec IrDA connection.

Time to replicate mobile IP (2.3 KB)	14.2
Time to “update” UbiqDir	1.5 (30 entries)
Time to subscribe to Romvets	0.142 (10 entries)

Table 9.7 Cost of Mobile IP protocol adaptation

9.4 Conclusion

This chapter answered two questions: -

- Is UbiqtOS an efficient system for embedded devices?
- How efficient is adaptation in UbiqtOS?

The first question was answered by comparing the code size and the efficiency of different components in UbiqtOS with other related systems, while the second question is answered by evaluating two adaptive applications made possible by UbiqtOS.

The fixed part of UbiqtOS is comparable to commercial embedded operating systems. However the need to dynamically lookup components at runtime makes UbiqtOS slower than traditional systems that provide fixed services.

The need to lookup components and the use of explicit bindings, where make the system flexible, increase the complexity of applications as shown in the Follow-me-Video application.

Still, UbiqtOS meets its design goals by providing efficient context-aware adaptation, not provided by any previous system, to address the heterogeneity, longevity, mobility and dynamism of a ubiquitous system.

"We have not succeeded in answering all your questions. The answers we have found only serve to raise a whole set of new questions. In some ways we feel we are as confused as ever, but we believe we are confused on a higher level and about more important things."
Guido Van Rossum

Chapter 10

Conclusion and Future Work

Advances in digital electronics and rapid convergence between communication technology and consumer electronics have led to the development of sophisticated network appliances. This trend coupled with the emergence and widespread use of wireless, portable computers, presents with an opportunity to enable a ubiquitous system. Such a system would allow the computation resources to disappear in the infrastructure to define active spaces; buildings, shopping malls, theatres, rooms, instrumented with embedded devices that collaborate under user's directions to automatically carry out their everyday tasks.

The whole system, therefore, would consist of a multitude of, possibly disconnected, active spaces to provide ubiquitous access to system resources according to the current context of the user.

This dissertation has presented architectures to make such a system possible.

It investigated the challenges posed by heterogeneity, longevity, mobility and dynamism of the system, and showed how dynamic, application-specific and context-aware adaptation in UbiqtOS addresses these challenges. Additionally, it presented a simple event-based architecture to allow interoperability of low-end devices.

The reader is encouraged to go back and read the motivational scenario presented in chapter 1 to appreciate how close the architectures presented in this dissertation come to enabling such a ubiquitous system.

This chapter outlines the contributions made by the thesis and discusses the avenues opened by this research in a field that is still in its infancy; ubiquitous systems.

10.1 Contributions

This dissertation makes contributions in three broad areas. The first consists of conceptual ideas underpinning the work. The second consist of the architectures created in the course of the dissertation. The third set of contributions are the lessons learnt from the qualitative and quantitative evaluation of these architectures.

Each of these areas is discussed in the following sections.

10.1.1 Conceptual Contributions

Many researchers and industrial consortiums have started to realize the potential of enabling a ubiquitous system that could allow computation resources to disappear in the infrastructure to automate everyday tasks of the users. Research efforts like Portolano [Esler99], Gaia [Kon00], and most noticeably Oxygen [Oxygen] from MIT are all in their initial stages towards the vision to enable a ubiquitous system. Similarly, emerging industrial standards like UPnP from Microsoft™, Jini from Sun™, along with a whole barrage of other proprietary solutions, especially in the field of home and office automation, indicate the trend towards the vision of an infrastructure that could enable interoperability of embedded devices to automate user tasks.

However, the work presented in this thesis is the first to present a universal substrate that could be embedded in participating devices to enable efficient ubiquitous interaction. It identifies the challenges posed by such a system design and highlights the requirement for dynamic, application-specific and context-aware adaptation of the enabling system to address these challenges.

The second conceptual contribution made by this thesis is the taxonomy of devices in a ubiquitous system. The taxonomy is based on whether the devices can just export their embedded resources to be controlled by other devices or can support additional software to control, manage and program the system as well. This taxonomy led to two related, though different, control architectures for the system. An events approach to control, and monitor the limited capability devices and an embedded operating system to allow programmability of the system with novel applications.

The final conceptual contribution of the work was its evaluation strategy. As UbiqtOS is a language-based, embedded, extensible distributed operating system, the benchmarks chosen to validate its design evaluate all these aspects. However, UbiqtOS addresses new problems and makes novel applications possible and its context-aware adaptation cannot be evaluated by traditional operating system performance benchmarks. Therefore, the evaluation strategy proposes the use of novel applications to evaluate the design of UbiqtOS along with traditional measurements to enable comparison with related systems.

10.1.2 Architectures

This dissertation has presented two related but different systems, AutoHAN and UbiqtOS.

10.1.2.1 AutoHAN

AutoHAN proposes the use of events as a unified model to control, monitor and program devices in an active space. Event scripts not only allow a clean declarative style programmability of an active space, the home automation rules encoded by them are amenable to formal verification to allow to allow detection of pathological errors. Further, its design relies on the use of well-known open wire protocols instead of fixed APIs to allow future-proof interoperability with other systems. Devices can support any execution environment, but as long as they understand the open wire protocols of AutoHAN they can participate in the AutoHAN system. The use of XML to describe resources and HTTP as the transport protocol to carry events and access DHAN allowed

the AutoHAN architecture to be extended to allow users to control their home network using the Internet.

Finally, the use of events to monitor and control the device only requires the devices to support an event-loop, allowing even limited capability devices to participate in the system. AutoHAN proposes three simple additions to the GENA architecture that address the mobility and dynamism of the system without compromising simplicity.

The experience with AutoHAN led to the design of an embedded operating system for fat devices that allows their additional capacity to be used to control, manage and program an active space with novel applications.

10.1.2.2 UbiqtOS

The rest of the thesis described in detail the requirements, design, implementation and evaluation of UbiqtOS, an embedded adaptable distributed operating system. The design of UbiqtOS addresses the heterogeneity, longevity, mobility, and dynamism of the system by providing

- Dynamic
- Application-specific and
- Context-aware adaptation

This is enabled by four artifacts in its design.

- A Java-based extensible agent engine (SEMAS) to allow context-specific software to be injected in the system, executed securely in a device and moved around to accomplish its task.
- An extensible registry (UbiqDir) to allow components to be dynamically discovered in the system using intent-based lookups. UbiqDir serves to capture and export the changing context of the device to components residing with it to provide context aware adaptation.

- A synchronous-events routing system (Romvets) to allow dynamic, application-specific and context-specific extension of components installed with UbiqtOS.
- Finally, a dispatcher module that exports context-specific views of distributed resources in an active space.

These four artifacts provided the building blocks to enable dynamic, application-specific and context-aware adaptation in UbiqtOS.

Further, the use of explicit bindings and extensibility provided application-specific and context-aware “effective mobility” in UbiqtOS.

An implementation of UbiqtOS architecture was presented and evaluated to demonstrate the feasibility and efficacy of the artifacts proposed in UbiqtOS design. The implementation required optimizations to be introduced in the Romvets subscribe/notify architecture and the event handling mechanisms in Kaffe to allow fast handling of events to achieve acceptable performance. With this optimization in place, UbiqtOS prototyped delivered acceptable performance for application-specific scheduling and context-aware protocol stacks.

Finally, the merit of UbiqtOS design was demonstrated by evaluation of two novel applications; context-aware bindings and context-specific mobile IP support.

Where the flexibility offered by UbiqtOS addresses the challenges posed by a ubiquitous system, it exposes the dynamism of the system to applications, which results in increased code complexity.

10.2 Future Work

This dissertation has opened many avenues for future work in several areas.

10.2.1 Context-specific Protocols and Policies

UbiqtOS is infrastructure. It defines a system framework that lends itself to safe, dynamic, application-specific and context-driven adaptation. Where this approach is clearly a step in the right direction to solve the unique challenges posed by a ubiquitous

system design, only a limited number of policies and applications were described in the dissertation to use this infrastructure.

The dissertation leads to the whole new field of research to investigate the suitability of different system policies and protocols suitable for different system characteristics.

10.2.2 Power-driven Adaptation

Battery operated mobile devices need to economize on the power consumed by the system. The current implementation of UbiqtOS does not provide support for power-driven adaptation. Where power-driven adaptation for policies like scheduling and I/O access requires support from hardware, to allow system clock and disk-access policies to be adjusted, some policies for power-driven adaptation can be implemented in software alone. For example, routing protocols in ad-hoc networks can adjust their willingness to route depending on the power left in their batteries, load-balancing policies can take into account the power left in the device as one of the factors to balance load in the system, services to provide fault-tolerance in the system can judge the reliability of the device by the power left in its batteries etc.

Simple power-driven adaptation in UbiqtOS can be supported by the same scheme as the “warning-memory” system is implemented. The power monitoring services can generate low-battery warnings for dynamically deployed services using Romvets, and they can adapt their policies accordingly.

10.2.3 Security

Though Java’s dynamic safety and the access control implemented by UbiqDir protect system integrity against illegal access to system services, UbiqtOS relies on the assumption that agents coming from trusted entities would not hoard system resources to pose denial of service threats. Where this scheme makes the implementation of UbiqtOS simple to be accommodated in embedded devices, it limits the interoperability and future-proof-ness of the system by requiring every device to know a priori the security keys of every other trusted host.

A self-authenticating scheme like proof carrying code holds promise to address this shortcoming but PCC research is still in its early stages. Security in such an open system is still an open research issue.

10.2.4 Application Complexity and Backward Compatibility

As mentioned earlier, UbiqtOS exposes the dynamism and context-awareness of the system to applications, resulting in increased code complexity. Second, the new APIs offered by UbiqtOS are not compatible with traditional systems like POSIX based systems, requiring applications be re-written to make use of UbiqtOS. This shortcoming can be addressed by development of appropriate libraries that hide the unwanted details of the system and allow existing applications to be executed by UbiqtOS. Explicit bindings and the Dispatcher allow “glue code” to be interposed between the context-aware interface of UbiqDir, SEMAS, Romvets and the applications. For example, the the follow-me-video binding in the example presented in chapter 9 hides the context-awareness of the system from the application; the application only names the right binding to use and the application hides the details of UbiqDir and Romvets from the application.

10.2.5 Embedded Device Implementation

The prototype implementation was done on x86 due to the easy availability of prototyping tools like OSKit and Kaffe. Where a resource constraint device was used to emulate an embedded device, it is clearly desirable to implement the UbiqtOS architecture on an embedded device. The current implementation of UbiqtOS is used in x86-based devices like the Warren Controller in our testbed. AutoHAN project will port the implementation to new ARM-based custom-made networked devices as part of the AutoHAN project, over the next year.

10.2.6 Active Space Automation Rules

The use of event scripts, written in a formal algebra, not only allows a clean declarative style programmability of an active space, the automation rules encoded by them are amenable to formal verification to allow detection of pathological errors. However, the techniques for formal verification of event rules are still under research in the AutoHAN group.

10.3 Summary

This thesis proposed, presented and evaluated two architectures to enable a ubiquitous system. AutoHAN defines an event-based system to control, monitor and program limited capability devices and UbiqtOS allows medium to high-end devices to control, extend and program the system. UbiqtOS is a radically new architecture for embedded operating systems, which is viable given the processing power now affordable, and addresses the challenges posed by heterogeneity, longevity, mobility and dynamism to enable a ubiquitous system.

BIBLIOGRAPHY

- [Accetta86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tavanian, and Michael Young, *Mach: A New Kernel Foundation for Unix Development*, USENIX 1986 Summer Conference Proceedings. USENIX Association, June 1986.
- [Alistair96] Alistair C. Veich and Normal C. Hutchinson. *Kea -- a dynamically extensible and configurable operating system kernel*. In Proc. of the 3rd International Conference on Configurable Distributed Systems, pages 236--242, May 1996.
- [Anderson92] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*. ACM Transactions on Computer Systems, 10(1):53--79, February 1992.
- [Aniruddha96] Aniruddha Gokhale and Douglas C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in Proceedings of SIGCOMM '96, Stanford, CA, August 1996, ACM, pp. 306--317.
- [Back00] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. *Techniques for the Design of Java Operating Systems*. In Proc. of the USENIX 2000 Annual Technical Conf., pages 197--210, San Diego, CA, June 2000. USENIX Association.
- [Bacon00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, Mark Spiteri, "Generic Support for Distributed Applications" IEEE Computer 33(3), March 2000, pp 68-76
- [Balakrishnan95] Balakrishnan, H., Seshan, S., Amir, E., Katz, R., "Improving TCP/IP Performance over Wireless Networks," Proc. 1st ACM Conf. on Mobile Computing and Networking (Mobicom), Berkeley, CA, November 1995.
- [Banavar00] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. *An Application Model for Pervasive Computing*. In Proceedings of the 6th

- Annual International Conference on Mobile Computing and Networking, August 2000.
- [Beck96] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, "*Linux Kernel Internals*", Addison-Wesley, 1996.
- [Belkhatir94] N. Belkhatir, J. Estublier, and M. L. Walcelio. *ADELETEMPO: An environment to support process modeling and enactment*. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 187 -- 222. John Wiley & Sons Inc., 1994.
- [Bernstein96] P.A. Bernstein, "*Middleware: A Model for Distributed System Services*", *Communications of the ACM*, 39(2), February 1996.
- [Bershad94] Bershad B.N., Chambers C., Eggers S. et al., "*SPIN: An Extensible Microkernel for Application-specific Operating System Services*", in *Proceedings of the 6th ACM Sigops Workshop on Matching Operating Systems to Application's Needs*, Warden, Germany, September 1994.
- [Bershad95] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. *Extensibility, safety, and performance in the SPIN operating system*. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267--284, Copper Mountain, CO, 1995.
- [Beuche99] D. Beuche et al. *The PURE Family of Object-Oriented Operating Systems for Deeply Emb edded Systems*. In *Proc. of ISORC'99*, St Malo, France, May 1999.
- [Birrell82] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, "*Grapevine: An Exercise in Distributed Computing*", *Communications of the ACM* 25, 4 (April 1982), 260-274.
- [Birrell84] A.D. Birrell and B.J. Nelsen. *Implementing remote procedure call*. *ACM Transactions on Computer Systems*, 2(1), 1984.
- [Blair98] Blair, G.S., Coulson, G., Robin, P. and M. Papathomas, *An Architecture for Next Generation Middleware*, *Proc. Middleware '98*, The Lake District, England, November 1998.

- [Bray98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible markup language (XML) 1.0. Recommendation*, W3C, February 1998. <http://www.w3.org/TR/1998/REC-xml19980210>.
- [Brewer98] E.A. Brewer, R.H. Katz, Y. Chawathe, S.D. Gribble, T. Hodes, G. Nguyen, M. Stemm, T. Henderson, E. Amir, H. Balakrishnan, A. Fox, V.N. Padmanabhan, S. Seshan, "A network architecture for heterogeneous mobile computing", IEEE Personal Communications, vol 5, num 5, 1998 Oct, pp 8 -- 24.
- [Bricker91] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and M. Rozier, "Architectural Issues in Microkernel-based Operating Systems: the CHORUS Experience" Computer Communications, Vol 14, No 6, pp 347-357, July 1991.
- [Campbell93] Campbell, R.H., N. Islam, D. Raila and P. Madany "Designing and Implementing Choices : An Object-Oriented System in C++," Communications of the ACM, Vol. 36, No. 9, September 1993.
- [CEBus] Electronics Industries Alliance, "CEBus Standard EIA-600," EIA, Arlington, Va., Sept. 1996. <http://www.Cebus.com>
- [Ceri96] S. Ceri and J. Widom. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Mateo, 1996.
- [Cheriton94] D. R. Cheriton and K. J. Duda. *A Caching Model of Operating System Kernel Functionality*. In Proc. of the First Symp. on Operating Systems Design and Implementation, pages 179--193. USENIX Association, Nov. 1994.
- [Chess95] D. Chess, C. Harrison et A. Kershenbaum. *Mobile Agents: Are They a Good Idea ?*. IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, march 1995.
- [Clark88] Clark, D. D. *The Design Philosophy of the DARPA Internet Protocols*, in the Proceedings of ACM SIGCOMM '88, August, 1988.
- [Czerwinski99] Steven Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. *An Architecture for a Secure Service Discovery Service*. In Proceedings of MobiCom '99, Seattle, WA, August 1999. ACM.
- [DiGirolamo99] J. A. DiGirolamo, R. Humpleman, "The VESA Home Network. A White Paper," Version 3, <http://www.vesa.org/VHNwhthpprV3.PDF>, August 20, 1999.

- [Doorn95] L. van Doorn, P. Homburg, and A.S. Tanenbaum. "*Paramecium: An Extensible Objectbased Kernel*". In Proceedings Hot Topics on Operating Systems V, Orca's Island, Washington, May 1995. IEEE.
- [Douglas95] Douglas B. Orr. *Application of meta-protocols to improve OS services*. In HOTOS-V: Fifth Workshop on Hot Topics in Operating Systems, May 1995.
- [Douglass91] F. Douglass and J. Ousterhout. *Transparent process migration: Design alternatives and the Sprite implementation*. Software: Practice and Experience, 21(8):757--785, August 1991.
- [Eicken92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. *Active Messages: a Mechanism for Integrated Communication and Computation*. In The 19th Annual International Symposium on Computer Architecture, pages 256--266, Gold Coast, Australia, May 1992.
- [Eicken99] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel. *J-Kernel: A Capability-Based Operating System for Java*. Lecture Notes in Computer Science, 1603:369--394, 1999.
- [Engler95] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. *Exokernel: an operating system architecture for application-specific resource management*. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, December 1995.
- [Esler99] Esler, M., Hightower, J., Anderson, T., and Borriello, G. *Next Century Challenges: Data-Centric Networking for Invisible Computing: The Portolano Project at the University of Washington Mobicom 99*
- [Fielding99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP 1.1*. RFC 2616, June 1999.
- [Ford97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. *The Flux OSKit: A Substrate for Kernel and Language Research*. In Proceedings of the Sixteenth ACM Symposium on Operating System Principles, pages 38--51, Saint-Malo, France, 1997.
- [Ford99] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. *Interface and Execution Models in the Fluke Kernel*. In Proceedings of the 3rd USENIX

- Symposium on Operating Systems Design and Implementation, pages 101--116, Feb. 1999.
- [Fox97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. *Cluster-based scalable network service*. In Proceedings of SOSP'16, October 1997.
- [Francisco99] Francisco J. Ballesteros, Christopher K. Hess, Fabio Kon, Sergio Arévalo, and Roy H. Campbell. *Object-Oriented in Off++ - A Distributed Adaptable Microkernel, 2nd ECOOP Workshop on Object-Oriented and Operating Systems*. Lisbon, Portugal. June 14, 1999.
- [Ghormley97] D.P. Ghormley, D. Petrou, S. H. Rodrigues, A.M. Vahdat, T.E. Anderson. *GLUnix: a Global Layer Unix for a Network of Workstations*. Comp. Sci. Div., Univ. of California at Berkeley. Berkeley, CA 94720, USA, Aug. 14, 1997.
- [Ghormley98] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. *SLIC: An Extensibility System for Commodity Operating Systems*. In USENIX 1998 Annual Technical Conference, June 1998.
- [Greaves98] D. J. Greaves, R. J. Bradbury. "Warren: A low-cost Home Area Network", IEEE Network, 12(1):44-56, January 1998.
- [Grimm00] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. [A system architecture for pervasive computing](#) In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 177-182, Kolding, Denmark, September 2000.
- [Guillemont97] M. Guillemont. "CHORUS/ClassiX r3 Technical Overview." Chorus Systems Technical Report, May 1997.
- [Haartsen00] J Haartsen, "The Bluetooth Radio System", IEEE Personal Communications, Feb 2000, pp. 28-36
- [Harper98] *The Fox Project: Advanced Language Technology for Extensible Systems*, Robert Harper, Peter Lee, and Frank Pfenning, CMU-CS-98-107, 1998.
- [Harris01] Tim Harris, *Extensible Virtual Machines*, Ph.D. Thesis, University of Cambridge, April 2001.

- [Hartig97] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Walter. "*The Performance of micro-Kernel Based Systems*". In Proc. 16th SOSOP, Saint-Malo, France, Oct 1997.
- [Hartman94] J.H. Hartman, A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. *Scout: A communication-oriented operating system*. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994.
- [Helander98] J. Helander and A. Forin. *MMLite: A Highly Componentized System Architecture*. In Proc. of the Eighth ACM SIGOPS European Workshop, pages 96--103, Sintra, Portugal, Sept. 1998.
- [Hildebrand92] D. Hildebrand (1992) *An Architectural Overview of QNX*, Proc. USENIX Workshop on Micro-kernels and Other Kernel Architectures, USENIX Association, Berkeley, CA, pp. 113-126.
- [Hildebrand94] D. Hildebrand. *QNX : Microkernel Technology for Open Systems Handheld Computing*. In Pen and Portable Computing Conference and Exposition, May 1994.
- [Hodes97] Todd Hodes, Randy Katz, E. Servan-Schreiber, and Larry Rowe. *Composable Ad hoc Mobile Services for Universal Interaction*. Proceedings of the 3rd ACM International Conference on Mobile Computing and Networking, pages 1--12, 1997.
- [HomePNA] Home Phone Networking Alliance. <http://www.homepna.org>
- [Howes97] Timothy A. Howes and Mark C. Smith. *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*. Macmillan Technical Publishing, 1997.
- [Hsieh93] W.C. Hsieh, M.F. kaashoek, and W.E. Weihl. *The persistent relevance of ipc performance: New techniques for reducing the ipc penalty*. In 4th Workshop on Workstation Operating Systems, pages 186--190, October 1993.
- [Javaref98] *The Java Reflection API*. <http://www.javasoft.com>, 1998.
- [Jordan98] Mick Jordan and Malcolm Atkinson. *Orthogonal Persistence for Java - A Mid-term Report*. Proceedings of the Third International Workshop on Persistence and Java (PJW3), 1998.

- [Karnik98] Neeran Karnik and Anand Tripathi. *Agent Server Architecture for the Ajanta Mobile-Agent System*. In Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), pages 66--73, July 1998.
- [Katz96] R. Katz. *The Case for Wireless Overlay Networks*. Invited talk at the ACM Federated Computer Science Research Conferences, Philadelphia, 1996.
- [Kiniry98] J.R. Kiniry, "*The Specification of Dynamic Distributed Component Systems*", M.Sc. Thesis, Department of Computer Science, California Institute of Technology, also Technical Report CS-TR-98-08, July 1998
- [Kon00] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. *2K: A Distributed Operating System for Dynamic Heterogeneous Environments*. In Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9), Pittsburgh, August 2000.
- [Kouichi91] Kouichi Kimura and Ichiyoshi Nobuyuki. *Probabilistic analysis of the efficiency of the dynamic load distribution*. In The Sixth Distributed Memory Computing Conference Proceedings, 1991.
- [Kourai98a] Kourai, K., S. Chiba, and T. Masuda, "*Fail-Safe Mechanism for Extensible Operating Systems*," in SIG notes of Information Processing Society of Japan (98-OS-77), pp. 197--202, Feb. 1998.
- [Kourai98b] Kourai, K., S. Chiba, and T. Masuda, "*Multi-Level Protection: A New Fail-Safe Mechanism for Extensible Operating Systems*," Journal of Information Processing Society of Japan, vol. 39, pp. 3054--3064, Nov. 1998.
- [Kung95] A. Kung, B. Jean-Bart, O. Marbach, S. Sauvage. "The EHS European Home Systems Network". Trialog, 25 rue de General Foy, 75008 Paris, November 1995.
- [Lange98] Lange D. and Oshima M. *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley, 1998
- [Lea00] Lea, R., Gibbs, S., Dara-Abrams, A., and Eytchison, E., *Networking Home Entertainment Devices with HAVi*, IEEE Computer, 33(9), September, 2000, pp. 3543.

- [Leslie97] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications*. IEEE Journal on Selected Areas in Communications, 1997.
- [Liedtke95] J. Liedtke. *On microkernel construction*. In Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain Resort, Colorado, December 1995.
- [Lindholm96] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.
- [Linux] Linux embedded devices, <http://www.linuxdevices.com>
- [LONWORKS] Echelon, *The LonWorks Company. LonWorks Solutions*. <http://www.echelon.com/Solutions>
- [Ludascher99] B. Ludascher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. *View definition and DTD inference for XML*. In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, 1999.
- [Ma98] Chaoying Ma and Jean Bacon, "*COBEA: A Corba-Based Event Architecture*", In the Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems, Santa Fe, New Mexico, April 1998
- [Minar99] Minar, N., Gray, M., Roup, O., Krikorian, R., Maes, P., (1999), "*Hive: Distributed Agents for Networking Things*", Proceedings of. ASA/MA '99.
- [Montenegro98] G. Montenegro. *Reverse Tunneling for Mobile IP*. In RFC 2344, 1998.
- [Mullin83] James K. Mullin. *A second look at Bloom filters*. Communications of the ACM, 26(8):570--571, 1983.
- [Murray99] Murray, J. (1999) *Inside Windows CE*, Microsoft Press, Redmond.
- [Necula97] G. Necula. *Proof-carrying code*. In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), January 1997.
- [Neuman94] B. Clifford Neuman and Theodore Ts'o. *Kerberos: An authentication service for computer networks*. IEEE Communications, 32(9), September 1994.

- [ODP95] ITU-T | ISO/IEC Recommendation X.903 | International Standard 10746-3: "ODP Reference Model: Architecture", January 1995.
- [OSGi] Open Services Gateway Initiative, <http://www.osgi.org>
- [Ousterhout88] J. Ousterhout, A. Cherson, F. Douglass, M. Nelson, and B. Welch. *The Sprite network operating system*. IEEE Computer, 21(2):23--36, February 1988.
- [Pascoe99] Pascoe, Bob, "Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun," Salutation Consortium, White Paper June 6 1999.
- [Peine97] Peine H., Stolpmann T., *The Architecture of the Ara Platform for Mobile Agents*, In: Rothermel K., Popescu-Zeletin R. (Eds.), Mobile Agents, Proc. of MA'97, Springer Verlag, Berlin, April 7-8, LNCS 1219, pp 50-61
- [Perkins97] C. Perkins, *Mobile IP Design Principles and Practices*. Addison-Wesley, 1997.
- [Pike90] R. Pike, D. Presotto, K. Thompson & H. Trickey (1990) *Plan 9 from Bell Labs*, Proc. of the Summer 1990 UKUUG Conf., London, July 1990, pp. 1-9.
- [Pike92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. *The use of name spaces in Plan 9*. In Proceedings of the 5th ACM SIGOPS European Workshop, pages 72--76, Mont Saint-Michel, 1992. ACM.
- [Priyantha00] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. *The Cricket location-support system*. In Proceedings of the Sixth Annual ACM International Conference on Mobile Computing and Networking, Boston, MA, August 2000. ACM Press.
- [Pu95] Pu, C., T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang: *Optimistic Incremental Specialization: Streamlining a Commercial Operating System*. Proc. of SOSPP 1995
- [Rogerson97] D. Rogerson. *Inside COM -- Microsoft's Component Object Model*. Microsoft Press, 1997.
- [Roman00] Manuel Roman and Roy H. Campbell. "[GAIA: Enabling Active Spaces](#)" 9th ACM SIGOPS European Workshop. September 17th-20th, 2000. Kolding, Denmark

- [Rothermel98] K. Rothermel and M. Straer, *A fault-tolerant protocol for providing the exactly-once property of mobile agents*, in Proc. of the 17th IEEE Symp. on Reliable Distributed Systems, West Lafayette, IN, Oct. 1998, 100-108.
- [Rozier88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. *CHORUS distributed operating systems*. The Usenix Association Computing Systems Journal, 1(4):305--370, December 1988.
- [Saif01] Saif, U., Gordon, D., Greaves, D. J. "Internet Access to a Home Area Network", IEEE Internet Computing:54-63, Jan-Feb, 2001.
- [Saif01b] Umar Saif, David J. Greaves, *Communication Primitives for Ubiquitous Systems or RPC Considered Harmful*, Proceedings of ICDCS International Workshop on Smart Appliances and Wearable Computing, 2001.
- [Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. *End-to-end arguments in system design*. ACM Transactions on Computer Systems, pages 277288, 1984.
- [Schmidt99] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. *The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture*. In Proceedings 17th ACM Symposium on Operating Systems Principles, pages 32--47, Charleston, SC, December 1999. also appears in Operating System Review33: 5.
- [Sletzer94] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. *An Introduction to the Architecture of the VINO Kernel*. Technical Report TR--34--94, Harvard University Computer Science, 1994.
- [Small96] C. Small. *MiSFIT: A Minimal i386 Software Fault Isolation Tool*. Technical Report TR--07--96, Harvard University Computer Science, 1996.
- [Sokol90] L. M. Sokol and B. K. Stucky. *MTW: experimental results for a constrained optimistic scheduling paradigm*. Proceedings of the SCS Multiconference on Distributed Simulation, 22(1):169--173, January 1990.
- [Stoica01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for Internet applications*. Technical Report TR-819, MIT, March 2001.

- [Suri00] N. Suri et al., "*NOMADS: Toward a Strong and Safe Mobile Agent System*," Proceedings of the 4th International Conference on Autonomous Agents (Agents 2000) Barcelona, Catalonia, Spain, June 3-7, 2000.
- [Tabatabai98] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. *Fast and effective code generation in a just-in-time Java compiler*. ACM SIGPLAN Notices, 33(5):280--290, May 1998.
- [Talukdar97] A. Talukdar, B. Badrinath, and A. Acharya. *MRSVP: A Resource Reservation Protocol for an Integrated Services Packet Network with Mobile Hosts*. Technical report DCS-TR-337, Rutgers University, 1997.
- [Tanenbaum90] Tanenbaum, A., van Renesse, R., van Staveren, H., and Sharp, G. 1990. *Experiences with the Amoeba distributed operating system*. Communications of the ACM , 336--346.
- [Tannenbaum95] A. Tannenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [Toole92] J. O' Toole and D. Gifford. *Names should mean what, not where*. In 5th ACM European Workshop on Distributed Systems, September 1992. Paper No. 20.
- [Tullman98] P. Tullman and J. Lepreau. *Nested Java processes: OS structure for mobile code*. In Eighth ACM SIGOPS European Workshop, Sept. 1998.
- [UPnP] Universal Plug and Play Consortium, <http://www.upnp.org>
- [USB2.0] *Universal serial bus (USB) 2.0 specification*. <http://www.usb.org/>, April 2000
- [Vahdat99] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal, "*Active names: Programmable location and transport of wide-area resources*," in 2nd Symposium on Internet Technologies and Systems, Boulder, CO, October 1999, USENIX.
- [Veizades97] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol (SLP)*. Internet RFC 2165, June 1997
- [Waldo99] Jim Waldo. *The Jini Architecture for Network-centric Computing*. Communications of the ACM, pages 76--82, July 1999.
- [Waldspurger95] C. A. Waldspurger and W. E. Weihl. *Stride Scheduling: Deterministic Proportional-Share Resource Management*. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

- [Want92] R. Want, A. Hopper, V. Falcao, and J. Gibbons, "*The active badge location system*," ACM Transactions on Information Systems, vol. 10, pp. 91--102, Jan. 1992.
- [Wickelgren97] I.J. Wickelgren "*The facts about FireWire*", IEEE Spectrum 34, 4 (April 1997), 19--25.
- [Wilkinson00] Tim Wilkinson et al. *The Kaffe virtual machine*, 2000. <http://www.kaffe.org>.
- [Williams00] S Williams, "*IrDA: Past, Present and Future*", IEEE Personal Communications, Feb 2000, pp. 11-19
- [Winoto99] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. *The Design and Implementation of an Intentional Naming System*. In Proceedings of the ACM Symposium on Operating Systems Principles, pages 186{ 201, Charleston, SC, 1999. Page 14
- [Wojciechowski99] Pawel T. Wojciechowski and Peter Sewell. *Nomadic Pict: Language and infrastructure design for mobile agents*. In Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents), Palm Springs, CA, USA, October 1999.
- [Wong97] Wong D., Paciorek N., Walsh T, *Concordia: An Infrastructure for Collaborating Mobile Agents*, in Rothermel K., Popescu-Zeletin R. (eds), Mobile Agents (Proc. 1st Int. Workshop), Springer-Verlag, LNCS 1219, 1997, pp 86-97
- [Wong97] Wong D., Paciorek N., Walsh T, *Concordia: An Infrastructure for Collaborating Mobile Agents*, in Rothermel K., Popescu-Zeletin R. (eds), Mobile Agents (Proc. 1st Int. Workshop), Springer-Verlag, LNCS 1219, 1997, pp 86-97
- [X.10] X10 Devices <http://www.x10.com/>
- [X.500] ITU-T, Recommendation X.500, *Information technology - Open System Interconnection - The directory: Overview of concepts, models, and services*, November 1995
- [Yokote92] Y. Yokote (1992) *The Apertos Reflective Operating System: The Concept and its Implementation*, Proc. OOPSLA '92, ACM, pp. 414-434.

[Zhao98] Zhao, X., Castelluccia, C., Baker, M.: *Flexible Network Support for Mobility*
Proc. MOBICOM '98 Dallas, Texas, USA p145-156 (ACM, 1998).