ADAPTATION FOR MOBILE SYSTEMS

This dissertation is submitted for the degree of Doctor of Philosophy

Tim Edmonds
PETERHOUSE

September 26, 2001

LABORATORY FOR COMMUNICATIONS ENGINEERING
Department of Engineering
University of Cambridge

Abstract

Mobile devices operate in a environment with limited and volatile resource availability, suffering from poor CPUs, memory and battery limitations and low rate, unreliable communications when compared with their fixed counterparts. Under such conditions, applications for mobile computing require adaptation to make best use of the available resources without assuming the minimum set. This dissertation argues that such adaptation should be integrated, with centralised resource management for efficiency and fairness, but with application specific reconfiguration to ensure that adaptation remains meaningful.

Current integrated adaptation systems provide resource change events to applications but fail to say exactly how to react to them. This dissertation presents Pervasive Adaptation as a new approach to integrated adaptation. This approach separates the concerns of adaptation from application functionality by providing automatic dynamic adaptation as a service external to the application. This service instruments and models an application's operation and performs reconfiguration of the application, achieving adaptation through manipulation of the locality and fidelity of application processing and data. Separating the adaptation service provides two key benefits: firstly, it reduces the work required of an application developer and secondly, it allows the same adaptation service to adapt all of the applications running in the system, ensuring cooperation and fairness.

The culmination of this work is the design and implementation of the DPROJ framework which provides a complete infrastructure for the construction, deployment and dynamic adaptation of mobile applications. It not only provides the mechanisms for adaptation but also the "know-how". The adaptation service reconfigures the structure of the application based on cost-benefit analysis. Costs are estimated through a resource unification scheme under which many different resources such as CPU, communications and battery can be combined into a common metric. Performance is maintained as a dimensionless, application specific notion. Running in a simulated network environment, DPROJ is demonstrated to successfully manage the adaptation of several applications including media streaming and vision processing.

Keywords: adaptation, mobile computing, wearable computing

Declaration

Except where stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. No part of this work has been submitted or is currently being submitted for any degree or diploma at any other university. A paper summarising this work is published as [21] and parts of Chapter 4 have previously appeared in [23].

This dissertation is not greater than 60,000 words in length, including tables, footnotes and bibliography.

Tim Edmonds
September 26, 2001

Acknowledgments

I would like to thank my supervisor Andy Hopper who has offered excellent guidance yet promoted the freedom to explore. Many thanks also to Steve Hodges for his sterling performance in an advisory role and to Antony Rowstron for some enduring guidance from back in the early days. I would also like to offer my thanks to all members of the LCE past and present who make the lab such an easy place to spend time. Remaining kudos, not an insignificant amount, go to friends and family for all the usual stuff.

This work has been made possible through the generous support of The Cambridge Commonwealth Trust, Peterhouse, and AT&T Laboratories Cambridge.

Contents

List of Figures

User Applications and Mobility

1

Continuing advances in wireless communications technology and mobile low power devices bring opportunities for new and varied applications. Mobile devices, wearable computing and ubiquitous sentient computing are all burgeoning with the increased availability of such untethered computing opportunities.

Applications for these platforms require adaptation to overcome both the shortcomings and limitations of these platforms and also to capitalise on the novel features not found with stationary computing. Currently available software adaptation technologies, developed for operation within a predominantly static, fixed network, do not map well onto applications for mobile systems.

This dissertation presents a new approach to creating and running adaptive applications for mobile computing. In particular, it details the design and implementation of the DPROJ framework which provides a set of tools and services for the creation, deployment and operation of adaptive applications in a mobile system. Applications operating within this framework benefit from automatic integrated adaptation with little developer overhead.

1.1 A CASE STUDY

As is customary in theses of such futuristic topics, a scenario is provided to help envisage the context and utility of mobile applications in the near future to the average user[36, 81, 66]. For comparison, an examination of a current day scenario may prove helpful:

> **Scenario 1.1:** URBAN EXECUTIVE LIFESTYLE, 2000AD
> *Info-warrior Arthur embarks on a trans-global sales pitch*
> Arthur leaves his office and catches a train to the airport. On the train he works using his laptop which connects to his office system for periodic updates and email collection using his GSM phone. Arriving at the airport, Arthur stows the masses of equipment and cables back in his bag and wan-

ders aimlessly around the airport until he boards his flight. On the plane, he works in an offline mode on his cached documents.

Following current technological and social trends, such a scenario might be very different in just five or ten years time. More convenient access to smaller, faster technologies coupled with broader communications infrastructure and new user interfaces bring a new dimension to mobile computing:

Scenario 1.2: URBAN EXECUTIVE LIFESTYLE, 2010AD
Info-warrior Bill embarks on a trans-global sales pitch
Bill leaves his office, and catches a train to the airport. His computer interaction and his train of thought are unbroken as he works seamlessly using his head-up display and wearable computer which chooses amongst the wireless links available to communicate with the office servers. Arriving at the airport, Bill's computer guides him through check-in and then takes dictations and reads messages while he wanders around aimlessly. The computer on Bill's wrist automatically finds nearby bargains and a healthy lunch; personalised ads are superimposed on billboards specifically for him (how else does Bill afford all of the communications cost?). On the plane, Bill's computer recognises the reduced bandwidth conditions and reconfigures its operation to reduce its dependence on external resources.

Analysing what is required 'behind the scenes' in the above scenario gives some guidelines for what to expect from next generation mobile application services. This in turn helps form principles of operation for future mobile systems and application frameworks:

- Mobile computing operates in a very dynamic environment with continuously changing demands and resource availability,
- An abundance of overlaying communications networks[48] means that connectivity can be mostly continuous though varying levels of quality of service may be available at different times and under differing price structures,
- Mobile computing will become much more than simple remote data access (web browsing) and asynchronous messaging (email). Internet computing is slated to follow an increasing trend towards becoming not only a giant data repository but also a universal provider of services,
- Many of the services and much of the functionality used by a mobile user are provided by third parties, some of whom may charge for these services. An interesting note is that the billboard application not only consumes resources (bandwidth, CPU, battery, etc.) but also generates resources (dollars), and
- Much of the user interaction with the computing environment is passive: many computing tasks operate in the background, on the periphery of the user's conscious actions.

Additionally, some other observations about the general technological trend may be made:

- connectivity will be high and almost ubiquitous,
- mobile devices will tend to be resource-poor when compared to their static counterparts,
- there will be more mobile devices and more mobile services than we are familiar with today, and
- mobile applications will be mostly new applications so they will need to be constructed from scratch.

1.2 Adaptation

Adaptation is the ability of an application to perceive change its environment and then alter its own behaviour according to that change.

Developing applications with this capability is challenging and time consuming as the developer must manually code for every permutation of every resource situation. The large number of different services available, varying resource availability and a large number of concurrent tasks expand combinatorially the number of configuration decisions to be made. The user is not expected to micro-manage the operation of the mobile system — this should be performed automatically, at most the user may provide some high level hints.

This thesis introduces the concept of *Pervasive Adaptation*, a form of integrated adaptation in which adaptation is externalised as a service to applications. This service can instrument, model and reconfigure a suitably constructed application at multiple levels. In doing so, it ensures both horizontal and vertical integration of the adaptation mechanism through all layers of the system and amongst applications to maximise performance against resource cost.

The DProj framework employs pervasive adaptation to provide a complete application framework supporting automatic dynamic adaptation for mobile applications. It is hoped that this research will prove to be a basis not only for future mobile system development but also for other distributed thin client systems including wearable computing, ubiquitous embedded sensors/actuators and sentient computing.

1.3 Contribution of the Thesis

The thesis of this dissertation is that it is both feasible and beneficial to provide automatic dynamic adaptation of applications in mobile computing systems. In particular, it emphasises the following:

- mobile applications must adapt to be of best use in the changing conditions experienced by a mobile computing system,
- to support diverse, concurrent applications in an efficient manner, such adaptation needs to be pervasive,
- separating the concerns of adaptation from the functionality of the application is an important aspect of providing a generic adaptation system, and
- the DPROJ generic hierarchical framework is a plausible and useful approach to constructing such applications and such a system.

In the process of proving the above thesis, the work described in this dissertation has resulted in the following artifacts:

- the DPROJ adaptation framework comprising:
  - hierarchical Element Composition (EC) notation as a means for specifying application structure and function as a hierarchy of reusable functional building blocks,
  - the EC application skeleton generating toolset to process the above notation into executable code providing the structural skeleton of the application,
  - CORBA based DPROJ runtime support to link with the skeleton to provide application manipulation and monitoring functionality,
  - the DPROJ Application Monitor to hook into running applications and perform profiling of application resource requirements and performance and thence to perform reconfiguration of the application, and
  - primitive adaptation policy implementations which derive application reconfiguration decisions from collected profile information.
- application inspection interfaces and tools,
- an adaptation evaluation environment including modulated resource simulation, and
- some adaptive applications which operate using the DPROJ framework.

1.4 LIMITATIONS OF SCOPE

A complete adaptation framework requires a broad set of services. It is not the intent of this work to provide this complete set but instead to focus on providing the adaptation core. Hence, it attempts to use existing software or middleware services wherever possible and any peripheral services that need to be implemented specifically for this work are done so in a minimal fashion. Additionally, it assumes the following limitations on the scope of the adaptation core:

Mechanics not Policy — This thesis aims to produce a framework and mechanism providing automatic adaptation to mobile applications but without specifying or assuming any single adaptation policy. It should be possible to graft different policies onto the same framework for different automatic adaptation behaviour.

Human-Computer Interaction — Mobile and wearable computing applications belong to a class of application that by their nature involve tight interaction with the mobile users. While this thesis does not suppose to cover the relevant Human-Computer Interaction (HCI) issues, it is sensible to take user-friendliness and user interaction into account when designing these application frameworks and mobile systems. To this end, the DPROJ framework developed in this thesis makes several assertions about the nature of user interaction with mobile systems and the qualities of a such a system that a mobile user might expect. In particular, user interaction is required in several parts of the adaptation decision making process: in balancing resource priorities, managing resource budgets and sometimes even to evaluate application performance. While in theory such tasks could be well performed by heuristic based policies, user agents or learning algorithms, the simple 'user-based' approach is taken. This is in keeping with 'mechanics not policy.'

Host Mobility versus Code Mobility — The focus of this thesis is on providing adaptation services to mobile applications. Here, the term *mobile application* refers to an application intended for operation on a mobile device which is physically mobile and moves from location to location and from network to network. The application may also, but not necessarily, be capable of virtual mobility across hosts through code level migration. These are often distinguished as *mobile computing* and *mobile computation* respectively[14].

Mobile computation is a logical progression from mobile computing, presenting another facet to the problem of mobility and also, in some cases, aid in the solution. However, such aspects lie outside the scope of this thesis.

1.5 Structure of the Dissertation

The remainder of this dissertation proceeds as follows:

Chapter 2 discusses current and future trends in mobile computing systems. In particular, it describes devices and wireless communications, the prospect of continuous connectivity, and software support for mobiles in a distributed system. Furthermore, it examines the nature of applications that operate in mobile systems and outlines some general principles of operation.

Chapter 3 introduces the concept of adaptation as a counterweight to varying resource availability. It reviews some existing adaptation frameworks for both fixed and mobile systems and discusses the application areas and limitations of each.

Chapter 4 details initial experiments in bringing adaptation to applications for real-time control of mobile devices. It demonstrates a prototype adaptive application and discusses the merits and limitations of the techniques used.

Chapter 5 takes the lessons learned from chapter 4 to build a new, more general adaptation framework. It presents the concept and design of the DPROJ framework, a toolset and runtime for constructing adaptive applications suitable for mobile users.

Chapter 6 gives the details of the implementation of the DPROJ framework and of several peripheral services that are required for operation of the framework.

Chapter 7 details evaluation of DPROJ operation using simple test cases. This includes the design of a controlled evaluation environment and application instrumentation and visualisation extensions. It is followed by a description of the implementation of a vision processing application for augmented reality implemented using the framework.

Chapter 8 concludes with a summary of the contributions of this thesis and the implications it has for mobile computing. Finally, it suggests some scope for further work.

Mobile System Technologies

2

The advent of widespread coverage and the increasing capacity of wireless communications today allows an increase in the mobility of computing such that it becomes more than just portable computing. People will find themselves computing when on the move, not just carrying a computer from place to place. Theoretically, it allows us to use a device anywhere and at any time to access any type of data or service. The realm of mobile computing is expanded beyond the device in hand and the user is able to harness world-wide resources. Computers now find themselves obsolete if they can't interact with the rest of the world.

For the purposes of this dissertation, the term *mobile system* is used to refer to a complete infrastructure required to support such mobile device operation. Figure 2.1 shows such a system including the mobile device itself, a fixed server and network back-end and the communications infrastructure connecting the two. Not shown is the software structure: the applications, services and middleware that are also required to make the system useful.

This chapter aims to familiarise readers with the basic structure of a mobile system and the technologies which underpin today's mobile systems. To do this, it examines the components that comprise such systems and details the particular qualities that differentiate them from similar but wired, fixed systems. It concludes by providing some background on the use of mobile systems and details some of the characteristics of applications for these systems, explaining why, with the addition of mobility and the qualities of a mobile system, these applications may be expected to be new.

2.1 Devices and Wearables for Mobile Computing

Currently available mobile devices include such items as mobile phones, Personal Digital Assistants (PDAs) and laptop computers. Together, these items represent a broad cross-section of the classes of devices imaginable.

The mobile phone is primarily a communicating device with limited data and processing capacity. Therefore it is advantageous for any applications that might be run

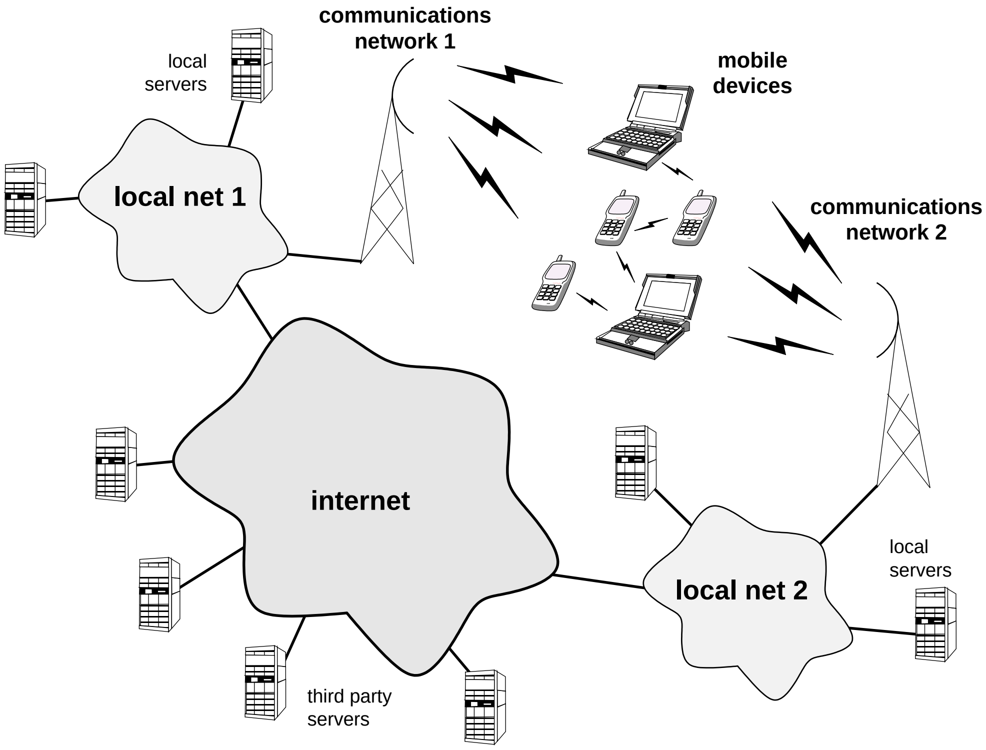


Figure 2.1: Mobile system infrastructure

on a regular mobile phone to be run as thin-client applications. Mobile phones are the primary focus of WAP (Wireless Application Protocol)[95], a thin-client approach to providing application services to mobile users. A key component of WAP is WML (Wireless Markup Language), an HTML like language that provides WAP enabled devices with web-like connectivity and applications.

Laptop or notebook computers represent the opposite end of the mobile computing spectrum. Basically a cut-down and portable desktop PC, laptop computers, when coupled with a means of wireless communications (GSM modem or wireless LAN), provide a computationally rich mobile computing device that is able to run applications developed for desktop computers constrained only by the limitations of the communications system. Laptop computers are of course much bulkier than mobile phones: whereas one might carry a mobile phone everywhere and use it on the move, a laptop computer is carried between locations in which it is used.

PDAs form an intermediate class between laptops and phones. They present a subset of each in a package smaller than a laptop yet typically with less communications capability than a phone. Light and highly portable so as to be taken everywhere, these cut-down devices offer an amount of on board processing but are unable to perform intensive work. Smaller PDAs such as the PalmPilot[90] or the Psion series[88] exist primarily as a kind of portable data store and light-weight application executor while more powerful PDAs approach the size and functionality of a laptop.

Although PDAs and mobile phones may have less processing capability than a lap-

Figure 2.2: Steve Mann in cyborg gear.

top, with the addition of communications, many users find that they are equally useful. While the number and quality of applications that may be run locally are small, communication brings access to a wider realm of applications. These small and very mobile devices comprise the beginnings of what may be termed *wearable computing*.

Wearable computing has long been an academic research topic but has until recently been limited to very few researchers who actually *wear* their computers for continuous all day, everyday computing. Steve Mann is regarded as one of the first *cyborgs*[56], wearing cameras, a head-up display, radio communications gear and a backpack full of electronics and batteries. While this may seem extreme (see Figure 2.2), such wearables do give us an indication of what may be expected in the future.

Miniaturisation of technology coupled with advances in mobile communications technology have made wearable computers a viable and useful option. Hands-free computing capability and ubiquitous data access provide greatly enhanced services to an individual. Applications range from Bush's *memex*, a hypothetical memory augmentation device[13], to *augmented reality* processing where artificial constructs are blurred with reality through visual overlays and other sensory stimuli[56, 77].

Wearable devices have different human interfaces than a desktop computer. It is cumbersome and ungainly to carry around a large screen and keyboard and mouse. Instead, wearables gain their inputs from small keypads, voice commands and contextual or location information. Feedback can be audio, visual (on smaller screens, head-up displays or retinal projection) or even tactile.

Furthermore, Weiser[97] envisioned a world of ubiquitous computing where computing devices are *integrated into the environment to such an extent that they become invisible to common awareness*[96]. These ubiquitous computing devices encompass not only wearable computers but also embedded and environmental devices. Wherever there is a sensor or an actuator, it can be networked for universal reading and control. For example, many small robots may wander around their workspace, relying on a remote server to control their actions[23]. Indeed many of these devices will match the mobile device profile: they will be small, cheap, battery operated and have limited processing; they will rely on remote communications, most likely wireless, to achieve full functionality and they will be context or location dependent. Such devices too should be counted under the class of *mobile devices*.

On account of their portability, mobile devices have less resources available than their wired counterparts. Mobile device designers must produce a device that is physically small, light-weight and of low power consumption in addition to the usual constraints of hardware design. A mobile device will always have less storage, memory, processing power and interfaces than a contemporary fixed machine. While mobile technology will improve, the mobile devices will always linger behind their wired counterparts from a performance viewpoint. The WAP Forum[1] summarises the challenges of mobile devices in their Architecture Overview[95]:

> *"Mass-market, hand-held wireless devices present a more constrained computing environment compared to desktop computers. Because of fundamental limitations of power and form-factor, mass-market hand-held devices tend to have:*
>
> - *Less powerful CPUs,*
> - *Less memory (ROM and RAM),*
> - *Restricted power consumption,*
> - *Smaller displays, and*
> - *Different input devices* (e.g. *a phone keypad)."*

2.2 Wireless Communications for Mobile Systems

The communications infrastructure for mobile devices will typically be wireless, providing connectivity between the roaming mobile device and the world at large. The communications services will be many and varied. This diversity stems from the overlay argument of [48].

Katz argues that while mobile devices will suffer some communications variation when connected to a single wireless network due to interference from multi-path, environmental hazards or other users, this is likely to be only a small part of the variance observed. Much greater variations will occur as a result of switching between networks of greatly different quality which cover different areas or in some cases even the same area.

[1] The WAP Forum is an industry body committed to the WAP protocol for mobile internet and application access.

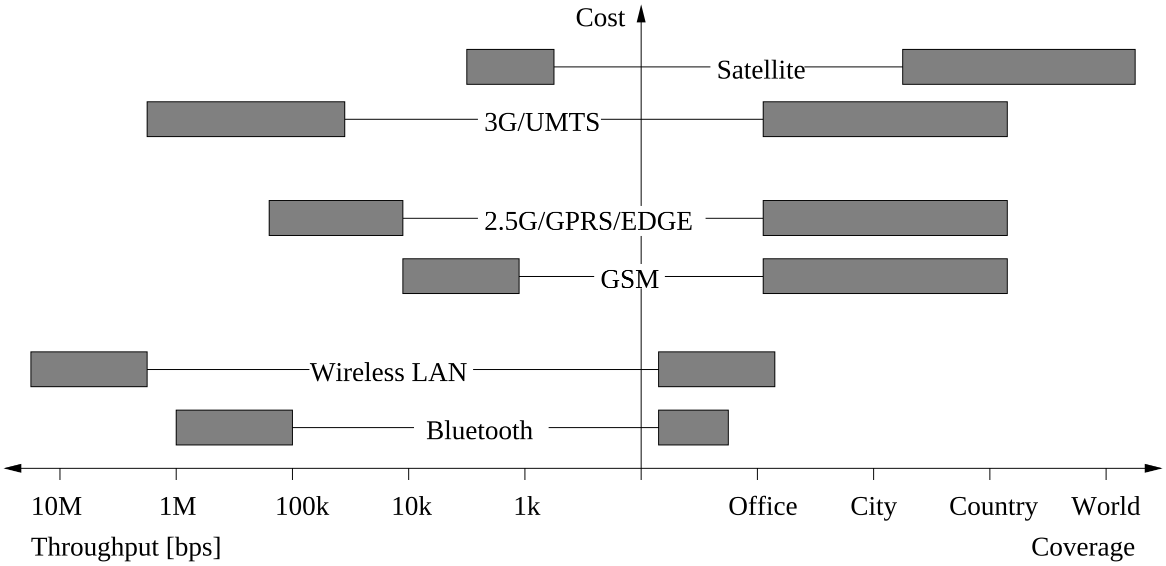


Figure 2.3: Throughput, coverage and relative cost of various mobile wireless communications technologies (2001).

Different wireless technologies operate at differing price/performance points making each amenable to a different environment. Figure 2.3 illustrates the throughput, coverage and relative cost of various wireless communications technologies currently or soon to be available. Here, relative cost is an approximation referring to the operational cost incurred by a users in using the communications facility. In general, greater coverage or greater throughput require more advanced technologies and are more expensive; different networks will focus on either high throughput or high coverage. Hence, overlaying networks results in expensive, low rate coverage for wide areas, reducing to cheaper, high throughput coverage for localised *hot spots*. Generally, the quality of wireless infrastructure deployed in any given area will be driven by the number of users in that area.

Table 2.1 lists some of the currently available communications technologies including some wired technologies (listed as having $desk$ range coverage). From a layered network point of view, a user may use LAN connectivity when at their desk and modem connectivity when at home. They might also have wireless LAN (such as waveLAN) available in their homes or places of work which offer untethered high rate connectivity. The nature and cost of such a network limits its deployment to such in-building use. Venturing outdoors, the user falls back onto systems such as GSM[2][32] or GPRS[3][12] which provide data transport capacity on the back of the cellular phone network. Venturing far from human habitation, the user drops back further to the globally available satellite based communications provided by companies such as Globalstar[28]. By this point, they have dropped to a relatively low throughput, high latency and very expensive service.

[2]Global System for Mobile communications
[3]General Packet Radio Service

Available	Technology	Rate	Coverage
fixed	LAN	100 Mbps	desk
	modem	56 kbps	desk
now	waveLAN	11 Mbps	office
	2.5G, GPRS	56 kbps	city
	GSM	9.6 kbps	country
	Globalstar	9.6 kbps	world
2002	2.5G, GPRS	115 kbps	city
2003	3G, UMTS	2 Mbps	city

Table 2.1: Currently available communications technologies (2001)

Like mobile devices, wireless communications are limited by the environment in which they must work. Again WAP Forum succinctly lists the shortcomings of wireless communication[95]:

> *"...wireless data networks present a more constrained communication environment compared to wired networks. Because of fundamental limitations of power, available spectrum, and mobility, wireless data networks tend to have:*
>
> - *Less bandwidth,*
> - *More latency,*
> - *Less connection stability, and*
> - *Less predictable availability."*

Additionally, the complexity of mobile communication is compounded by the presence of overlaying communications infrastructures. In such situations, a mobile application may experience:

- use of more than one wireless network concurrently[47],
- rapid switching between communications alternatives, and
- different pricing and access models: compare GSM's per minute dial-up with GPRS's always-on capability and per byte pricing.

Further, wireless communication is more power hungry than its wired equivalent. In any wireless communications link there is a trade off between transmission power, range and receiver error rate. The wireless receiver consumes additional power. With dial-up connection based communications such as GSM modems, the power consumption is approximately constant over time — the energy used remains the same independent of the amount of data transmitted or received; the reverse channel is always open for reception. However for always-on, packet based communications such as waveLAN or GPRS, power consumption is related to both the amount of data transmitted and time; having the communications interface operational for reception at all times can be the cause of significant power consumption.

2.3 TCP/IP Mobility Support

Connectivity for mobile devices will inevitably involve connection to and interaction with hosts on a legacy fixed network such as the internet. In this case, it is necessary to support the network protocols used on that fixed network. The most used protocol on the internet is TCP/IP and several efforts have focussed on extending TCP/IP to improve support for mobile hosts.

The *Transmission Control Protocol* (TCP) provides a reliable, connection-oriented data transmission service over the *Internet Protocol* version 4 (IPv4) packet bearer layer, commonly referred to as TCP/IP[61, 19]. TCP provides a virtual circuit transport layer over the IP network layer. Applications communicate using TCP via the common *port* scheme in which each virtual circuit operates through a separate *socket*.

TCP/IP is the fundamental networking protocol in use in the internet and is fast becoming the standard for nearly all applications. Unfortunately, TCP/IP has several shortcomings when when applied to the mobile computing field, a field for which it is not designed. Fundamental amongst these:

1. TCP/IP connections assume a static network and are fixed to the IP addresses of the client and server — a mobile host cannot move to another subnet without changing IP address and hence breaking existing TCP/IP connections,

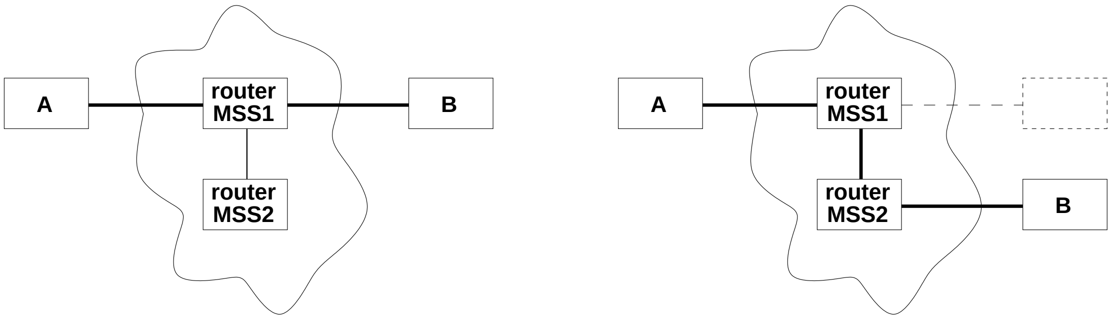
2. TCP/IP is connection-oriented and hence any communications disconnection or interruption for greater than the TCP timeout period will result in the TCP connection being dropped, and

3. TCP/IP uses exponential back-off as a means of congestion control — this functions well for wired networks where lost packets are a strong indicator of network congestion but fails for point to point links where there is only a single host at each end and for radio links where the bit error rate is much higher than wired connections (of the order of 10^{-4} compared to 10^{-10} for wired) and hence packet loss can usually be attributed to data errors rather than congestion.

Several projects have developed modifications of the TCP/IP stack or protocol to overcome these shortcomings in the mobile arena. These projects tackle the problems by interposing mobility support at different layers of the ISO network model, such as IP address (Network Layer), TCP connection (IP address plus a port number, Transport Layer) or socket descriptor (Session Layer).

2.3.1 Mobile IP

Mobile IP[62] provides infrastructure support to overcome the static network assumptions of IP which do not hold true for mobile hosts. This is achieved through the use of *Mobile Support Stations* (MSS) which are fixed hosts equipped to provide indirection support for IP by acting as *care-of* addresses.



(a) Mobile host B in its home campus, communicating to remote host A

(b) Mobile host B moves to a foreign campus: its communications are redirected via the foreign MSS.

Figure 2.4: Mobile IP supported inter-campus host movement.

Each MSS administers a logical internet subnet termed a *campus*. The MSS acts as a proxy for mobile hosts, receiving any connections to or from the mobile host and passing them through to the appropriate destination. Each mobile host has a home MSS. This MSS tracks the mobile host's current location. With reference to figure 2.4, when the mobile host moves to a new campus, it contacts the local MSS which reports the mobile host's new location to its home MSS. Now, when the home MSS receives incoming packets for the mobile host, it tunnels them through to the foreign MSS which then forwards the packets to the mobile host.

Mobile IP can be imagined as a generic TCP proxy performing address translation and packet forwarding for mobile hosts. Its major advantage is its transparent nature. Mobile IP was designed to implement mobility support while minimising changes in the current internet, by containing its mobile support functionality within the network layer, it can operate usefully without modifications to either the application software or the underlying network technology. The corresponding host need not have knowledge about Mobile IP to participate. However, Mobile IP violates the internet subnet addressing scheme in a very unclean manner. Further, the traffic routing is not optimised: all traffic must pass through the home MSS for rerouting. Thus, the home MSS also represents a single point of failure in the network, against the internet's original design goals. Further optimisations to bypass the home MSS may be possible in some cases without modifying the TCP/IP protocol. Modifications to the protocol to support complete bypassing of the home MSS and even the remote MSS are the subject of RFC2002[62]. These modifications have not been widely implemented outside of research systems but the concepts developed for MobileIP form the basis of mobility support in the upcoming IP revision, IPv6[86].

2.3.2 IPv6

Internet Protocol version 6 (IPv6)[20] is a new version of the Internet Protocol, designed as the successor to IPv4. In particular, the prime improvements of IPv6 over IPv4 are:

- expanded addressing capabilities,
- simplified header format,
- improved extensibility and options support,
- flow labelling to support QoS and differentiated services, and
- authentication and privacy capabilities.

Defined in the specification are several standard extensions and options which a fully compliant IPv6 protocol stack must support. These include *hop-by-hop* packet delivery, extended routing support, improved multi-homing and address resolution and renumbering through address rebinding. These may be used to implement improved traffic routing over the Mobile IP implementation, allowing direct mobile host to correspondent traffic and re-homing to new subnets. The system still requires a *Home Agent* (HA) for each campus (which corresponds to the MSS in a Mobile IP setup). The HAs continue to act as care-of addressees but include support to provide the *home* HA functionality in the case of non-availability of the home HA.

When a mobile node moves to a new location a hand-off is needed to guarantee a continuous connection between the mobile and a correspondent node. This is gained by informing the previous correspondent host by sending a binding update, which causes the previous HA to route packets arriving to the old care-of address to the new care-of address. A mobile should maintain a set of such care-of addresses to account for potential HA failure and for frequent hand-overs between subnets. The mobile node sends packets informing its correspondent nodes about its current address set and the lifetimes and the preferences of the addresses. A peer-destination cache is used by the mobile node to maintain a particular correspondent's address set view. Correspondent hosts that follow the rebinding order can then communicate directly with the mobile host. Correspondent hosts that do not follow the rebind or that receive the rebinding late continue to forward the traffic via the HA.

IPv6 hence solves the shortcomings of Mobile IP while maintaining compatibility at the IPv6 level between mobile and non-mobile aware hosts. IPv6 is not currently widely deployed but due to its backwards compatibility with IPv4, scope exists for it to integrate slowly with the existing internet. Already, some major internet backbones are built on IPv6 and some research networks exist based on IPv6. The advent of mobile internet devices with 3G mobile networks should propel the adoption of IPv6.

2.3.3 MobileSocket

Application level TCP/IP socket libraries such as MobileSocket[70] can provide similar functionality to Mobile IP by operating in the session layer without modifying the network layer. MobileSocket provides such functionality as a user-level, enhanced TCP/socket class for the Java Virtual Machine (JVM). It enables continuous communication including virtual circuit continuity for TCP socket based communications for applications written in Java.

MobileSocket uses two mechanisms to support continuity: Dynamic Socket Switching (DSS) and an Application Layer Window (ALW). DSS entails transparently switching the underlying native TCP channel used without disrupting the application's continuity of operation. The mobile host appears to open a socket connection to the correspondent host which remains connected at all times until the application closes it. Behind the scenes, continuity must be maintained when the remote host relocates or changes communications network (and hence also IP address) or if the TCP connection times out. The MobileSocket socket implementation silently closes the old socket and opens a new one at the new address and continues to process the data flows. If an application understands the extended MobileSocket interface, it may additionally perform this redirection imperatively or may register for call backs to be notified when such an event occurs (enabling adaptation).

The connection does not appear to be continuous at the correspondent host unless that host too uses MobileSocket which is able to perform these negotiations. If the connection is to a host or application that does not use MobileSocket, then the correspondent host must either accept discontinuity or a proxy *shim* must be installed on the fixed network between the MobileSocket and traditional socket connections. In this mode of operation, MobileSocket is similar to MSOCKS[55], a shared library offering transparent TCP session level continuity to all dynamically linked applications which requires a proxy server in all cases.

MobileSocket uses an Application Layer Window to provide further virtual circuit data consistency support. Data sent just previous to an implicit redirection to a new address may be lost in the buffers or in the network. It uses the window, a data buffer, to store sent data until a complete acknowledgement is received from the sender. This acknowledgement is a MobileSocket enhancement to the TCP protocol and requires that the correspondent host also use MobileSocket.

MobileSocket can be used transparently by existing Java applications and offers increased control to applications which understand the MobileSocket extensions. MobileSockets operate smoothly in both a mobile situation with hand-overs as well as in a fixed network while requiring no modifications to the network. Additionally, its Java implementation and the *serialisability* of the MobileSocket class can provide continuity even for applications that migrate amongst hosts. The primary limitations of MobileSocket are its dependence on Java (however MSOCKS[55] provides similar services at the operating system level) and its dependence on proxies or that the correspondent host (indeed server) also use MobileSocket (and hence Java).

2.4 Classical Distributed Systems

If a mobile device has wireless connectivity, it effectively becomes another host on the greater network. Software systems that span hosts connected in such a manner are termed *distributed systems*. The main motivations for implementing an application as a distributed system are resource sharing, improving resource availability and increasing reliability. A distributed application can use components concurrently, even on geographically disparate hosts. The downside derives directly from this distributed nature: the concurrency of components, lack of a global clock, latency and partial failure of components or communications[18].

2.4.1 Distributed Processing

Distributing the execution path of a software application across several, possibly heterogeneous computers over a local or wide area network is termed *distributed processing*. Distributed processing is the generic umbrella term for any software system where processing occurs on more than one host either concurrently, sequentially or arbitrarily. It is not, generally, concerned with remote data access such as world wide web transfers or media streaming. It encompasses aspects of client-server structures, resource sharing, load balancing and fault-tolerance. Some systems are inherently distributed due to their geographical nature (*e.g.* : environmental monitoring, group collaboration applications etc.)

The software infrastructure that enables distributed systems is termed *middleware* and provides an abstraction model and runtime support for distributed systems computing and applications. It is called middleware as it acts as an abstraction layer between the operating system (including network and communications services) and the applications themselves. It integrates tightly with both but is not logically a component of either. It provides *tools* to design, construct and deploy distributed applications and *services* to support the operation of these applications in a distributed environment.

Distributed systems and middleware may be employed for several reasons:

1. **Historical**: allowing new applications to connect with legacy stand-alone systems. Most middleware offers a clean 'wrapper' approach to encapsulating legacy systems and providing their services to newer applications.

2. **Functional**: allowing integrated access to specialised resources which may be available only on a separate host.

3. **Economical**: often it is more economical (both in development and maintenance terms) to build a system from a collection of smaller hosts than to have a single larger host.

4. **Operational**: a distributed system can increase reliability and fault-tolerance through component redundancy and simple maintainability. A distributed system also offers greater scope for reconfiguration and hence increased flexibility.

5. **Geographical:** many applications are naturally distributed over a physical area and hence any implementation of that application must itself be distributed.

The middleware platforms described in this chapter all aim to support applications while meeting these requirements to varying degrees. Of particular interest is item 5, a necessary condition for mobile devices working within a greater operating environment. Mobile systems bring special requirements through their geographical nature which impacts on all of the other considerations in manners not expected for static environments. It is the highly dynamic nature of mobile connectivity that requires a new approach not directly available through the systems described below which are representative of the current middleware platforms.

2.4.2 General Approaches

The key function of any middleware is to provide clean separation between application components running on different hosts in a manner transparent to the running application. Secondly, the middleware may also specify a set of standard services which operate with, and are built on top of, the middleware to provide commonly needed functionality.

Interface Based

One approach to building a distributed application involves augmenting a conventionally compiled programming language with pre-processors and runtime support to enable location-transparent component interaction. This approach generally offers support for a large range of practical languages and hence is more applicable to a wider set of problems and better suited to integration with legacy applications. In this approach, application components are defined by the interfaces that they export. These interfaces are specified in an Interface Definition Language (IDL) that provides a common cross-language interface specification. Middleware tools translate this IDL specification into stubs and implementations of the interface.

On the left of figure 2.5 is shown an intra-process client-server interaction between application components. These call directly as per any standard application. On the right is shown an inter-process interaction in which the application components are parts of separate processes which may be on different hosts. In this case, the client component code is unchanged but the component it interacts with is now a stub which marshals the call into a network format, passes this to the process corresponding to the server which unmarshals the call and performs the invocation directly upon the server component. In this manner, all notions of component location may be hidden from the programmer.

Java RMI

Another approach to building a distributed application, especially in an environment of heterogeneous hosts is to use a virtual machine approach to normalise all of the

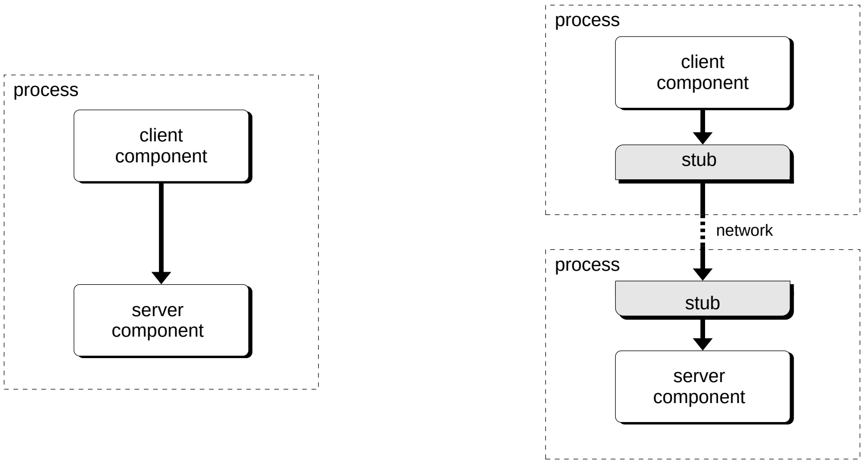


Figure 2.5: Intra-process calls pass directly while stubs are used as proxies to support inter-process calls.

hosts to a common baseline. This is often performed by creating the applications using a cross-platform byte-code language such as Java[29]. In particular, Java's Remote Method Invocation (RMI) allows applications to call object methods located remotely, sharing resources and processing load across systems. Unlike other systems for remote execution which require that only simple data types or defined structures be passed to and from methods, RMI allows any Java object type to be used — even if the client or server has never encountered it before. RMI allows both client and server to dynamically load new object types as required. This approach has three main benefits: firstly, the byte-code components can be executed on any of the hosts. Secondly, the virtual machine environment can often be packaged and moved to another host, hence allowing an application component to *migrate* without losing its state. Thirdly, these languages often provide the facility for introspection, making the components' internal structure and the data structures to the program itself. This allows the application runtime to inspect the structure of the application and dynamically create stubs or redirections to application components which have been moved or are located elsewhere. Manipulation of the program flow and data translation can be simplified and automated with this inherent meta-data. Such techniques may be incorporated as a specific implementation of some of the platforms described below.

Services

With this basic capability to interact with non-local components provided by the middleware, further, more sophisticated services can then be created using it. Typical services provide support to aid application component interaction with both users and other components. These may include:

1. **Naming**: provides mappings between a name hierarchy and components.

2. **Security**: provides user and application authentication and authorisation.

3. **Trading**: provides a method of finding a component that can provide a specified service or interface.

4. **LifeCycle**: provides mechanisms for creating, migrating and destroying components.

5. **Persistence**: provides a mechanism to save the state of an application component between executions.

2.4.3 Some Middleware Platforms

The following paragraphs give brief summaries of the key features of some representative middleware platforms.

ISO RM-ODP

The Reference Model for Open Distributed Processing (RM-ODP) is a joint effort by the ISO and ITU-T standards bodies defining a common set of terms, concepts and models to describe distributed processing systems[43, 42, 44]. RM-ODP aims to provide a "big picture" which organises the many pieces of an ODP system into a coherent whole but does not try to standardise the components of the system nor unnecessarily influence the choice of technology. In this, its function is much the same as the OSI Communications Network Reference Model[100] which provides a common model for communications systems.

RM-ODP defines a set of frameworks within which support for distribution, interworking, inter-operability and portability can be integrated. RM-ODP standardisation considers distributed systems spanning many organisations and technological boundaries. These typically lack any central point of control, and therefore show additional characteristics, such as heterogeneity, autonomy, evolution and mobility. In order to deal with these characteristics ODP standardisation aims to enable the building of systems with the following properties: openness, integration, flexibility, modularity, federation, manageability, provision of quality of service, security and transparency.

Transparency is ODP's mechanism of abstraction to ease the system development and maintenance cycle. Amongst others, ODP provides for abstraction over data representation, service and data location, relocation, replication and persistence. Much of the structure of RM-ODP is derived from ANSA.

ANSA

ANSAware is a middleware platform developed by the Advanced Network Systems Architecture (ANSA) project[57, 4]. Based on the ANSA architecture for Open Distributed Processing, it provided much of the groundwork for the ISO RM-ODP specification.

ANSAware comprises a software suite providing basic platform and software development support in the form of program generators and system management applications. ANSAware allows programs to be written in an object-based style in which each

ANSAware object is an encapsulation of an application and its data. These objects interact in a client-server manner through exposed *interfaces*. An object can provide and use more than one interface simultaneously. The interfaces may be used either locally or remotely by client objects. All objects are considered to be logically separated — when co-located, the location transparency service provided by ANSAware bypasses the network overhead by allowing direct communication.

This object-based approach allows effective management of distributed program components by providing distinct units of management and by containing failures.

OSF-DCE

The Open Software Foundation (OSF) has defined the Distributed Computing Environment (DCE) as a standard for distributed applications in an open environment[71]. It aims to provide a standard technology to make possible tight integration of system components supplied by different vendors.

DCE is composed of standardised components including threads, client-server interaction, naming and directory services, time synchronisation and authentication services all operating through Remote Procedure Call (RPC) across a heterogenous network.

2.5 OMG CORBA

RM-ODP, ANSAware and OSF-DCE have provided pioneering work in commoditising distributed systems but have made way to second generation middleware platforms. The most popular amongst these today are Java RMI, CORBA and DCOM, all of which build on object based technologies.

This section describes the Object Management Group's Common Object Request Broker Architecture (OMG CORBA)[68, 92] in detail, as it is used as the primary middleware platform for the work in this dissertation. However much of the discussion applies equally well to both Java RMI and DCOM which are founded on similar principles. Java RMI is introduced above in section 2.4.2; DCOM is the distributed version of Microsoft's Component Object Model (COM)[17]. All are object-based systems with only Java RMI meeting the criteria to be classified as being object-*oriented*. Java RMI works only with the Java language and byte code and hence can achieve the cross platform and first class object benefits as outlined in 2.4.2 above. DCOM and CORBA are not strictly object-oriented as they lack such features as functional polymorphism, overloading and functional inheritance, but represent an interface based approach to distributed components.

2.5.1 The CORBA Object Model

CORBA is a growing standard for building applications from components. The CORBA specification is both architecture and language agnostic allowing integration of diverse components. It aims to be truly open and flexible, allowing it to be used to integrate

```
interface Hello {
  string hello(in string who);
}
```

Figure 2.6: IDL description for a Hello interface.

systems from various vendors and it includes mappings to a large number of common programming languages as well as to other middleware platforms.

CORBA is centred around identifiable, encapsulated, reusable pieces of software referred to as *objects*. Its object-centric model is purely peer to peer without the distinction between client and server made in other distributed systems such as DCE. However, these objects may interact in a client-server manner by client objects making requests on a server object.[4]

The possible requests that an object can serve are defined by its *interface*. An interface is a strongly typed reference to a collection of related functions. In CORBA, like other middleware platforms, interfaces are specified in a format known as the Interface Definition Language (IDL). IDL is not specific to any one language or machine architecture and hence allows integration of objects operating on heterogeneous hosts written using different languages. In this way, it allows developers to easily build applications using objects even if they do not have the source code for those objects. IDL can be mapped into many popular programming languages and hence allows simple cross-language development. Figure 2.6 shows an example IDL description of a simple "Hello" interface. This interface defines a single method, `hello`, which takes a string and returns another string. The interface definition does not define the implementation or semantics in any manner but merely provides a strict syntactic definition.

An interface provides the external view of an object while the object provides the *interface implementation*. The interface implementation is bound to an explicit object instance and may be accessed through that object's *object reference*. Figure 2.7 shows a sample C++ invocation on a `hello` object reference. The implementation, location, host architecture and language of the actual `hello` object is hidden. Interactions between objects thus reduce to requests on an object reference. The mechanics of the requests are handled by the Object Request Broker (ORB).

2.5.2 The CORBA Object Request Broker

The ORB is the core infrastructure component of CORBA and acts as a software bus, logically relaying requests from the requesting client object to the serving object as in figure 2.8. If the server object is local, the routing reduces to a direct invocation on the object, while for non-local requests (for client and server in separate address-spaces or on different hosts) the ORB relays the request across the required communications path to reach the server object. This encompasses several additional stages compared to the

[4] The terms client and server in this case are purely relative and refer only to the objects dispatching and serving a request respectively.

```
#include <Hello.hh>

void use_hello() {
  Hello_var helloserver;

  //.. obtain reference to
  //    hello server somehow ..

  // a method call via the object reference
  cout << helloserver->hello("World");
}
```

Figure 2.7: Client calls to a Hello object are direct invocations on a Hello object reference, independent of the object's actual location. The reference may be obtained through any one of many possible means, for example via a Naming Service.

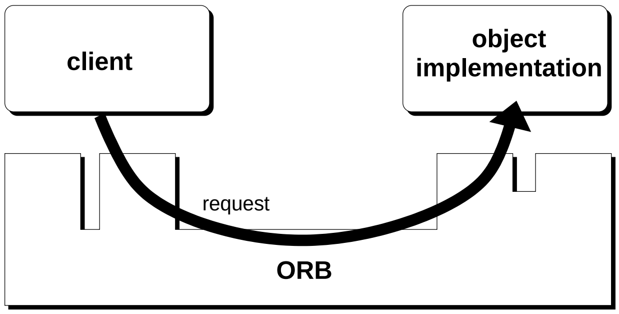


Figure 2.8: CORBA ORB software bus model.

direct case. Figures 2.9 and 2.10 shows the indirections performed by the ORB to relay the request through to the receiving object. This involves:

1. **IDL Stub**: The IDL object stub presents the server object's interface to the client, acting as a proxy for the actual server object. It is compiled from the interface's IDL description and contains the required type data for the parameters and return value (if any).[5]

2. **Marshalling**: Using the type information from the stub, the call and call parameters are then marshalled into a format suitable for network transmission.

3. **Transmission**: The call and parameters are transmitted to the server object's ORB over the network to a different host, or via local communication mechanisms should the object reside on the same host (but in a separate address space). There are

[5]it is also possible to perform *dynamic requests* in which the stub is not precompiled from the IDL definition but instead the equivalent functionality is dynamically built from the IDL description at runtime.

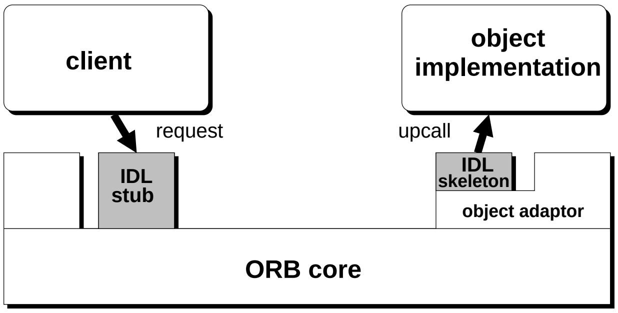


Figure 2.9: CORBA ORB stub and skeleton interfaces.

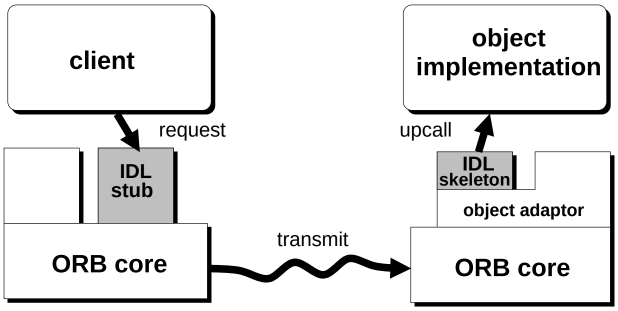


Figure 2.10: CORBA Inter-ORB request path.

various on the wire protocols, each suited for different ORBs or networks, but the most common is IIOP, the Internet Inter-ORB Protocol (which uses the General Inter-ORB Protocol (GIOP) over TCP/IP). These protocols specify exactly the transaction sequence and data formatting requirements.

4. **Unmarshalling**: Once received, the request is unmarshalled by the server's ORB into its native data format.

5. **Object Adaptor/IDL Skeleton**: The object adaptor performs the request upcall to the server object through the IDL skeleton interface. The IDL skeleton is precompiled from the interface's IDL description and provides an empty *skeleton implementation* of the interface. Typically, the actual object will be implemented by subclassing the skeleton and hence receiving the upcalls directly.[6]

6. **Return Path**: The request result returns via a process similar to the request. CORBA includes the ability to perform *one-way* requests in which no return value is expected — this enables elimination of the blocking time for the requesting object.

[6] as per the stub, the skeleton functionality may also be provided dynamically at runtime from an IDL description.

2.5.3 Standard CORBA Services

The CORBA standard[68, 69] defines several services to provide common functionality that is not specific to any single application domain. It provides IDL and semantic definitions of the services but does not mandate implementation or existence of the services. Indeed, many ORBs ship with only a selection of the defined services, while other services can be obtained from third party providers. The services compose an application support system and only the services that are actually required need be used. Below is a brief description of some of the more popular or generally applicable services:

Naming — The CORBA Naming Service provides a mapping from object names to object references in the same way in which the DNS maps internet host names to IP addresses. The Naming Service provides a hierarchical naming structure using *naming contexts*. Clients may use the service to bind an object to a name in a context and to resolve a name in a context to an object reference.

Trading — The CORBA Trading Object Service facilitates the offering and the discovery of instances of services of particular types. It can be viewed as a service through which other objects can advertise their capabilities and match their needs against advertised capabilities. The Trading Service allows dynamic discovery of, and late binding to, services by client objects.

LifeCycle — The CORBA Life Cycle Service defines services and conventions for creating, deleting, cloning and moving objects. This control of objects' life-time allows dynamic relocation of objects whilst maintaining references and relationships, using *forwarding stubs* if necessary to catch requests to the old location and forward them to the new location. The extension Compound Life Cycle Service can use object relationships held by a Relationship Service to extend the Life Cycle service to apply the Life Cycle functions to graphs of related objects at once.

Events — The CORBA Event Service provides a decoupled communication mechanism between CORBA objects via an *Event Channel*. An Event Channel connects one or more *event suppliers* with one or more *event consumers* in a manner that allows asynchronous one-way communication using both *push* and *pull* models (even concurrently). The CORBA Notification Service provides an extended Event Service with channel filtering and some quality of service enhancements.

2.5.4 Developing with CORBA

In summary, it appears that object-oriented middleware platforms such as CORBA are gaining the greater developer mindshare for distributed systems development. In particular, CORBA, DCOM and Java RMI are major platforms enjoying widespread use today. The levels of abstraction, transparency and efficiency enable simple integration of diverse platforms and application operation across heterogeneous platforms. Of the three, Java RMI has perhaps the cleanest object model, made possible by Java's own well crafted object model and its single language implementation. CORBA and DCOM are both cross-language platforms, differing primarily in their application/host domains: DCOM for Microsoft-only environments and CORBA for more diverse environments.

2.6 Applications for Mobile Systems

Section 1.1 examined the context of mobile computing and postulated that coming mobile applications will constitute a new application domain that may be expected to be substantially different and have different requirements when compared to current applications designed for static environments.

Mobile devices can perform word processing and office work, even email and scheduling without major consideration but users do not integrate such activities seamlessly with everyday life. These remain separate activities. Scenario 1.2 illustrated what might be expected from future devices from a user viewpoint: to have true wearable devices that can be accessed and used ubiquitously, with new user interfaces, new input and output mechanisms, and ever-present applications with continuous access and continuous usage, bringing passive 'helper' applications or agent-like assistants to the user. New applications with voice recognition, video handling, location systems, or intelligent agents, with large processing requirements and large data sets require new application infrastructure.

2.6.1 Models for Distributed Mobile Applications

The previously described middleware platforms are all designed to deploy distributed applications in a primarily static environment. Mobile applications bring new qualities and demands to the situation and while the fundamental concepts of distributed systems remain, the middleware platforms will need to evolve to provide further support.

Traditionally, mobile applications and mobile devices have been autonomous systems that operate independently while on the move but have the ability to *synchronise* with fixed or connected systems from time to time. However, the increased connectivity options detailed in section 2.2 and in particular always-on packet based communications promise a mostly-connected mobile device environment in which access to the wider network is more often there than not. In this case, applications will not need to synchronise data periodically but operate seamlessly within the network instead, perhaps caching data for the brief periods of disconnection that they might suffer.

Client-Server Architecture

Mobile devices and applications naturally lend themselves to operation within a client-server architecture where the application components are distributed between a client host and a server host. The client is usually characterised as being a terminal host or user interface device while a server host provides more fundamental services, often to more than one client. The client/server model can be implemented in several general variants

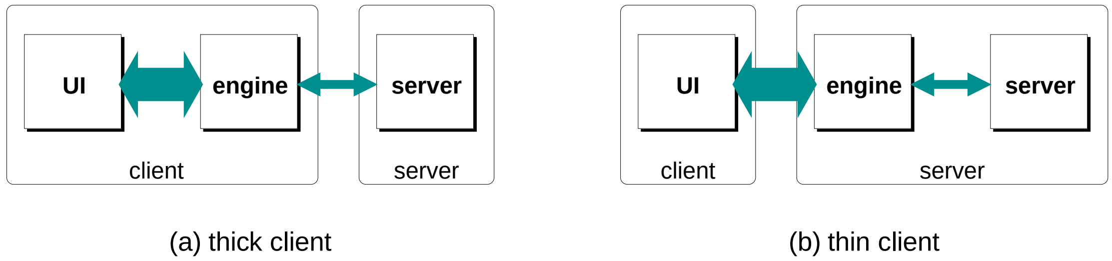


Figure 2.11: Client-server models.

depending on the exact implementation environment and implementation goals. These vary in the allocation of the application components to the client and server hosts.

A typical client-server application may be broken into three connected components: user interface (UI), processing engine and data/service server. The distribution of these components between client and server leads to the two major variants of the client-server model: *thick client* and *thin client*. Figure 2.11 shows these client-server models and the required communications bandwidth between the functional components.

A thick client approach (figure 2.11(a)) has a processing bias towards the client and can minimise bandwidth usage by having the client-server partition at the back end. An example of this might be a web-browser which connects to a web server to fetch a page of data but performs the data interpretation, layout and rendering in the client. In this case, the transfer of the page involves significantly less bandwidth than the transfer of a fully rendered page while the client is responsible for the bulk of the processing involved.

A thin client shifts the bandwidth/processing balance in the opposite direction, consuming greater bandwidth but having a lighter processing load at the client (figure 2.11(b)). The client can be lightweight but the server must instead be able to perform all the computation for all of its clients. An extreme example of this is the Virtual Network Computer (VNC)[78] thin client display using the Remote Framebuffer Protocol. VNC takes thin client to its extreme, making the client a zero-knowledge *stateless* entity. In this way, the displayed application (and workspace) state exists only on the server and the client is merely a display. This has the benefit of allowing the application to be accessed through various different clients transparently and without losing or changing any state. The thin client approach here consumes much greater bandwidth than a semantically richer display protocol (such as X11[84]) but can operate with a very resource poor client such as a PDA or a web-browser.

The thin client approach is an attractive model for mobile devices due to their inherent lack of resources. Thin client approaches can make clients cheap, lightweight (in all aspects) and replaceable and hence increase client practicality and system maintainability. However, as noted above, thin client systems have negatives and these are amplified in the case of mobile devices. In particular, thin clients require higher bandwidth connectivity while mobile devices will often be subject to fluctuating levels of connectivity with changing availability of bandwidth, latency and costs to the extreme of disconnected operation.

Limitations of the Mobile Computing Model

Dealing with objects in a distributed system is intrinsically different from objects that interact within a single process. Focusing on a client-server model can lead to improved distributed performance for an application while simplifying development; however, such object interactions require special attention to latency, memory access limitations, security and issues of concurrency, failure and partial failure [94, 16]. This is especially true for mobile systems which violate many assumptions made about the nature of distributed systems[15].

The *latency* of interaction via a wired network is typically four or five orders of magnitude greater than local intra-machine interaction. Ignoring this difference when designing a distributed application can lead to severe performance degradation if the design requires a large amount of communication between separated components. The exact partitioning of an application amongst several hosts for optimum performance becomes very dependent on the nature of the application and even on the particular use of an application.

Memory access in a distributed system must be completely controlled to preserve consistency between processes on the same or different hosts. Object based distributed systems (such as CORBA) perform this by requiring all object data access to be via method calls, using strict ownership and pass-by-value semantics for method data parameters. Shared memory based distributed systems perform this by providing locks on memory segments and/or traps to detect memory accesses — in either case, care must be taken to prevent deadlocking.

Distributed systems that operate in a public or open environment must be more *security* conscious than a system operating as a single process. In a distributed system, data and communications patterns may be observed by a third party which may cause concern for some applications, even if using strong encryption.

Concurrency, failure and partial failure are particularly salient points for distributed systems as, unlike purely local applications, objects in such systems are generally unaware of failures in remote parts of the system. With no global monitoring and resource management infrastructure (such as an Operating System), distributed components may be in an indeterminate state: an unanswered remote method invocation may not have returned because of either unexpected system latency, communications errors or failure of the remote host during the method invocation — all of these possible causes exhibit the same symptoms and are indistinguishable by the calling object.

While latency, memory access and security can conceivably be abstracted away given sufficient technological advances, partial failure issues cannot be adequately resolved in a true distributed system without a central point of control and monitoring. Fortunately, a mobile client-server architecture does not require that it be truly distributed. There is a point in the system that remains constant during operation: the client is an essential part of the system and may be assumed to be available for any notion of a useful system.

2.6.2 Principles of Mobile Application Development

Constructing individual, isolated, stationary applications is relatively simple and well understood; constructing applications that operate in a mostly connected distributed environment in which resources must be shared and may change is much more difficult[85]. The question then is "How should applications for mobile devices and systems be constructed?" There are several key design principles that must be considered when constructing mobile applications and systems:

> **Principle 2.1 (Diversity):** *Mobile devices will be called upon to execute a large range of applications concurrently over a large range of conditions.*

Current devices assume that the user only requires to perform one operation at any one time and so mobile phones and PDAs often run simple operating systems (such as PalmOS[90]) that only run and present a single application at a time. Future applications such as those predicted in section 2.6 may be more passive, pervasive and hence will require that multiple applications be run concurrently on a single device or single resource set.

> **Principle 2.2 (Adaptation):** *To perform such operation efficiently, adaptation is required[47, 66].*

The range of conditions a mobile application is subject to is broad and widely variable, forcing some level of adaptation upon any application that wishes to operate efficiently under more than one resource configuration. Adaptation is the subject of chapter 3.2.

Connected mobile devices have access to resources beyond the device itself in the form of remote data and remote services. An application that uses such remote services is a *distributed* mobile application. Using remote resources can greatly enhance the capabilities of a mobile application but at the same time, the limitations of the mobile infrastructure mean that the application must be able to cope with poor connectivity (up to and including disconnected operation), flagging resources (battery or CPU), and insecurity (mobile devices are much more susceptible to theft or security violations than their stationary counterparts)[81].

In particular, locality of data, locality of processing and locality of control must be resolved consistently:

> **Principle 2.3 (Data Locality):** *Where should data reside? The home of data may be either on the mobile device itself or in the network property. The choice depends on the application itself and the nature of the data.*

Intermittent connectivity means that if data is to be accessible at all times, it must reside on the device itself. On the other hand, the mobile device is small, limited and vulnerable — only small data sets can reside on the device. The device may only manipulate restricted data sets and employ external facilities for more data. Additionally, maintaining data access and consistency (in the case of shared data) through disconnection become major issues.

> **Principle 2.4 (Process Locality):** *Where should processing occur? As for data, provision must be made for the isolated case without restricting operation to exclude the advantages of connectivity to external facilities.*

Whereas maintaining consistent data through disconnection requires there being only a single data home or master, if process state is treated as data, then the processing itself may occur in any location. For a mobile device that may suffer disconnection, any application that is required to operate even without connectivity must be able to operate acceptably whilst performing all processing on the client.

> **Principle 2.5 (Control Locality):** *Where should system control and monitoring be performed?*

Mobile systems represent a distributed system in which the distribution is not even, but is biased towards the client as the centre of operations. This makes sense for continuous operation even under disconnection, however, existing telecommunications systems are very network-centric in their service approach.

> **Principle 2.6 (Clean Slate):** *New applications and new application classes present an opportunity to construct application systems and infrastructure without legacy or backwards compatibility impediments.*

As we might expect applications for mobile devices to be new and novel, a new and novel application framework tailored to mobile systems may be installed at the lowest level. Ideally, any new applications and application infrastructures for mobile devices should also allow a degree of inter-operation and coexistence with existing legacy applications as a secondary objective.

2.7 Summary

This chapter has provided a brief summary of the technologies involved in mobile applications and the infrastructure support for such applications. It has followed this with a summary of software support available and an outline of the principles that differentiate mobile applications from traditional fixed applications.

Mobile devices continue to shrink in size and increase in power, assuming tasks and functionality once reserved for fixed computers. While progress in mobile device technology is impressive, it is clear that in terms of performance, mobile devices will always lag behind their fixed counterparts.

Observation of the technological trends show that communications systems are part of the fastest growing technology sector today. GPRS and third generation services are designed to provide multi-level, always-on, everywhere connectivity. This level of connectivity leads to a *wearable* concept of computing where mobile applications are as accessible as the time from your watch. Applications which operate in an always-on manner become more like ancillary, helper agents rather than the monolithic explicit purpose applications we use now.

While device and communications technology pushes forward, the software systems in place afford little acknowledgement of the potential for integrated connected mobile operation. Current distributed systems technologies provide simple and obvious abstractions over issues of location and access that can provide a sound basis for developing distributed applications. Further abstractions can be applied over these layers to provide security, fault tolerance and mobility, with regards to both mobility of code and mobility of hosts.

Mobile computing occurs in a volatile environment, one where resources are variable and intermittent. Mobile devices and applications cannot rely on continuous connectivity, yet at the same time a device designed primarily for standalone activity is functionally and data limited when the potential of external resources are considered. When designing an application for operation in such an environment, care must be taken to address these connectivity issues with regards to locality of data, processing and system monitoring and control to account for degraded connectivity to the point of disconnection. Additionally, applications themselves should be aware of the changes in their environment to be able to perform optimally without assuming the minimum set of resources: to be able to take advantage of sporadic resources when they become available. This necessitates adaptation on behalf of the applications and the system.

The expected wave of completely new applications for mobile devices grants an opportunity to develop a fresh application infrastructure and development framework to overcome the limitations of mobile computing and provide enhanced services and adaptation mechanisms that allow such applications to make best use of available resources.

3 Adaptive Systems

This chapter presents the concept of adaptation as a powerful counterweight to the varying resource availability experienced by applications in a mobile context. For a mobile application to work effectively, it must work efficiently, while operating within the resource limits available to it. To do this, it might be designed to work with the minimum resource set that might be expected to occur over the execution life of the application. However, given that resource availability may change dramatically and unpredictably, the minimum set will often be a very small proportion of the average or maximum set (or may even be nil for some resources). To make *best* use of available resources in such *turbulent* conditions, a mobile application, and indeed the entire mobile system, must adapt[66].

Generally, adaptation may be achieved through two key controls: fidelity and locality, applied to both data and processing. This leads to two approaches to adaptation: *algorithmic adaptation*, in which the quality of data or processing algorithms is degraded to trade CPU or communications resources for performance, and *distributive adaptation* in which locality patterns are manipulated or exploited through the use of proxies, caching, replication or migration of data and/or processing in order to reduce communications usage or to reduce or spread the processing workload.

This chapter investigates both approaches, examining various forms of algorithmic adaptation and distributive adaptation in a fixed environment. This leads into an in-depth examination of several mobile adaptive systems which combine elements of both algorithmic and distributive adaptation. Whilst all of the systems presented achieve adaptation through control of fidelity and locality, each operates through a different mechanism. The choice of mechanism affects the effectiveness of an adaptation system under various criteria, such as efficiency, correctness, fairness and legacy application support. The mechanism used to implement an adaptive system may be used to classify it in a manner orthogonal to the adaptation control approach. The systems surveyed suggest a broad classification into three models: *application-transparent*, *application-specific* and *integrated* adaptation. A comparison of these *adaptation models* leads to the conclusion that effective, correct adaptation supporting multiple concurrent applications requires integrated adaptation but at the cost of legacy application support.

3.1 Considerations

When constructing an adaptive application, the developer must consider various points as to the exact nature of operation of the application and the nature of the adaptation required.

The key considerations must be *what* and *how*. The developer must foresee and decide under *what* conditions the system should react through adaptation. Typically, applications will require adaptation to cope with changes in resource availability or to changes in the performance goal of the application. *How* an application adapts is the crux of the adaptation process and must be addressed by any adaptation framework. The nature of how an application is to adapt should address:

- Should adaptation be open or closed? With open adaptation, new behaviours and strategies may be introduced during runtime such as through evolutionary programming or the dynamic addition or subtraction of services during operation; with close adaptation, the scope of adaptation and reconfiguration is known or specified beforehand.
- How should the adaptation process be integrated with the application? Which model of adaptation should be used and how should this integrate with the application functionality. Ideally, any adaptation mechanism should aim for separation of concerns of application function from adaptation as this facilitates independent operation, analysis and evolution of the adaptation mechanism.
- How autonomous should the adaptation process be? Adaptation may be fully automatic adaptation, service assisted adaptation or even user-guided adaptation.
- How frequently should adaptation occur? The rate of adaptation of an application derives from the balance between the desired *reactiveness* of the application to changes in the environment and the cost overhead of undertaking the adaptation process.
- How cost-effective is the adaptation process? All adaptation will incur some cost overhead which may even be greater than the costs saved through the adaptation process. Hence, the frequency and priority of adaptation may be adjusted to account for the cost-effectiveness.

3.2 Algorithmic Adaptation

Algorithmic adaptation involves altering the operational behaviour of an application depending on certain environmental variables. Algorithmic adaptation is typically very application specific, however some generalisations may be made about the possible forms that this approach may take. The following describes a broad spectrum of adaptive processes and a set of characteristics which may be used to describe adaptive per-

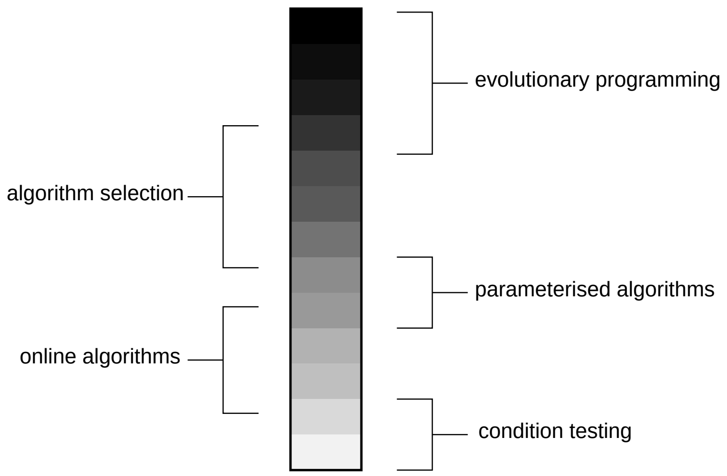


Figure 3.1: Spectrum of adaptation from basic isolated option selection at the lowest level through to unanticipated changes and self-programming at the highest level.

formance. This leads to a set of points to consider when designing an application that performs algorithmic adaptation.

3.2.1 A Spectrum of Algorithmic Adaptation

Algorithmic adaptation can occur at various levels of the system and to varying degrees. Figure 3.1 shows the spectrum of adaptation capabilities. The description below is a summary of this adaptation spectrum as described by Oreizy[72].

At the lowest level is *condition testing*, a primitive form of adaptation in which the program evaluates an expression at build time or start time and performs one or another task depending on the value of that expression. This simple method is easily implemented to provide basic adaptive behaviour, such as selecting different data sources or display formats depending on the current host device.

Online algorithms perform runtime adaptation of a process by observing current conditions and application use and optimising operation for such usage in the future. This can be seen in Java Just in Time (JIT) compilers which optimise byte code conversion in heavily used code or in caching algorithms that exploit temporal or spacial proximity to optimise data access.

Parameterised algorithms are algorithms that can perform within a range of performance-cost points depending on input parameters. These inputs can be keyed to environmental conditions, and hence provide a continuous adaptive behaviour. For example, a video compression algorithm might be parameterised by a quality factor and frame size parameters to control its use of processing, memory and bandwidth resources.

This concept can be extended and combined with condition testing as *algorithm selection* in which the most suitable algorithm to perform a certain task is chosen dynamically at run time from a selection of available, possibly parameterised, algorithms. The video compression example above might be implemented in this manner, selecting the most

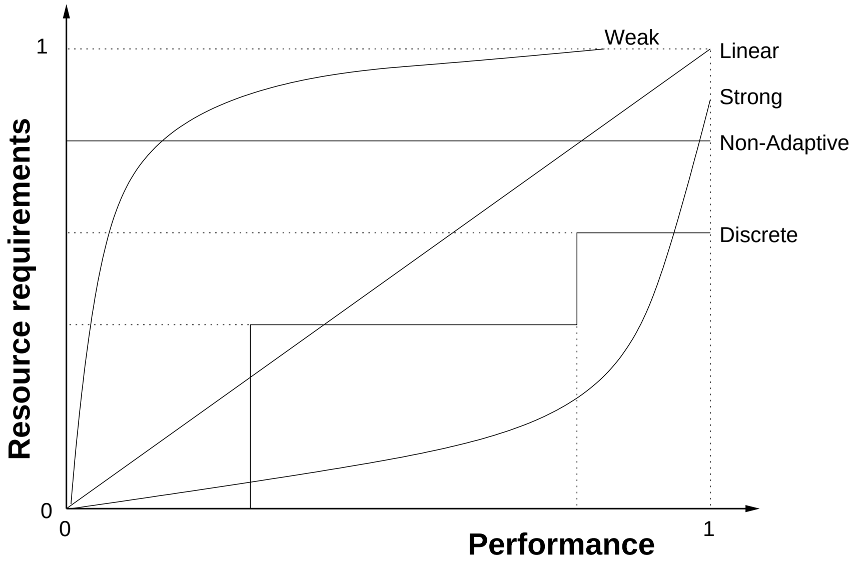


Figure 3.2: Adaptation characterisations used to classify applications or components according to how their resource needs vary according to desired performance.

suitable video codec dynamically according to resource availability and performance demands.

Finally, at the top of the spectrum, *evolutionary programming*, including genetic algorithms and machine learning, provides a means by which an application can self-adapt through self-modifying code, making it possible for an application to reach new configurations and new algorithms not originally conceived via analysis of the operating environment, conditions and the effectiveness of previously conceived algorithms.

3.2.2 Characterisation

Across this spectrum of techniques, adaptation can be characterised in terms of the cost-performance relationship a component exhibits. Figure 3.2 shows the relationship of resource requirements versus performance index for some generic adaptation characterisations. These characterisations may be considered fundamental — further characterisations may be built from some combinations of these:

No Adaptation — the component requires a set amount of resources to perform. With less than this it fails completely; with more resources, performance is not improved.

Discrete — the component has a discrete number of *modes* of operation, each of which operates at a different resource-performance point.

Weak — the component can adapt to operate under different resource conditions but cannot perform well until it has almost a full set of resources.

Linear — the component performance and resource requirements exhibit a direct linear relationship.

Strong — the component can adapt to operate under different resource conditions and can achieve high performance even under poor resource availability.

Condition testing typically results in discrete characteristics. Continuous adaptation in the form of linear, weak, or strong adaptation may be expected from or online or parameterised algorithms. Algorithm selection may be characterised as a discrete piecewise combination of continuous adaptation characteristics; the discrete pieces come each from a different algorithm selection, while each algorithm may present a continuous function within its piece. Any adaptation characterisation could theoretically be approximated by a discrete process with a sufficient number of modes, tending to infinity to approximate continuous functions.

3.3 Distributive Adaptation in Fixed Systems

Distributed applications provide access to distributed resources and can offer a more flexible system that can make efficient use of resources. However, designing a distributed system is non-trivial. In particular, the high overhead of accessing remote resources means that the locality of distributed resources must be a prime consideration in determining the distribution of an applications functionality. Thus, this distribution should derive directly from the application's data and code access patterns.

This section examines several *Automatic Distributed Partitioning Systems* (ADPS) which use program meta-data (*e.g.* symbol tables) along with in-place instrumentation and data flow and call flow monitoring to perform off-line partitioning of an application for operation in a distributed environment. These systems aim to distribute an application based upon profiling the operation of the application running sample scenarios in the execution environment it will eventually be run in. This allows the application to be adapted to run in a given environment (hosts, networks, etc) which may differ from the environment the application was originally designed for. Nevertheless, the operating environment is expected to remain static during the course of execution of the application. Application components are assigned to locations at compile time or at instantiation time and remain at these locations for the lifetime of the application (or component).

3.3.1 ICOPS and CAGES

Early work in distributed graphical systems postulated that distributed applications are best distributed using an automated partitioning system. Andries van Dam's Interconnected Processor System (ICOPS)[91] and Foley's Configurable Applications for Graphics Employing Satellites (CAGES) [33] were early automatic partitioning and distribution systems. Both automatically break an ALGOL-W or PL/1 application into relocatable chunks at a procedural module granularity using automatically generated stubs to perform remote procedure calls. A typical application might be composed of around 10 such relocatable modules.

ICOPS used scenario-based profiling to determine the optimal layout of the application. In this technique, the application is instrumented to monitor data flow and call paths whilst running a sample scenario. The information gathered is used to partition further runs of the application based on processor load minimisation or a minimum cut, maximum flow algorithm (Min-Cut Max-Flow[26]). The completed application would run in a virtual uniprocessor environment consisting of the main host and a collection of satellite hosts. The user could interrupt the program at any point and force a redistribution of the application.

CAGES was very similar to ICOPS in that it performed static analysis of a program to produce distribution information but was less automated than ICOPS: the programmer was still required to decide on the final distribution. CAGES advanced over ICOPS in providing the application a virtual dual-processor unihost execution environment distributed over two physical hosts. However, it did not allow redistribution at runtime, instead requiring recompilation for each distribution.

3.3.2 IDAP

The Intelligent Dynamic Application Partitioning (IDAP)[49, 50] system provides static application distribution as part of IBM's VisualAge Generator[41] using profiling information gathered from applications run in an Interactive Test Facility.

IDAP is a tool for distributing object or component based applications across client-server or *n*-tier topologies. It observes an actual run of the application, observing all interactions between objects, and then makes recommendations to the programmer concerning how the application should be partitioned into groups of objects and to which host machine each group of objects should be assigned to achieve the best overall performance of the application. These recommendations are then reviewed by a programmer who may refine the partitioning scheme or even refine the design of the application to further improve performance. Tests show that this automated static partitioning can give an order of magnitude sized performance increase over manual partitioning in some applications. IDAP has been shown to operate with applications composed of several hundred static objects. The partitioning remains fixed once decided upon by the programmer.

3.3.3 Coign

The Coign ADPS[39, 40] is similar to the systems described above but provides several advances: firstly, it acts upon binary executables and is hence language and development environment independent; secondly, it can act upon a much greater number of components — the examples given are composed of between 200 and 500 components; thirdly, Coign partitions both statically defined components and dynamically instantiated components.

Coign operates on Microsoft Windows win32 binary executables, performing partitioning of application COM components. This defines the granularity of partitioning and restricts operation to applications which are composed of many COM objects. It is indicated that further work could extend the system to operate on DCOM objects as well.

Coign aims to distribute both statically defined as well as dynamically instantiated components. It does not aim to relocate or migrate components and so operates by predetermining where a component should be instantiated and hence where it should remain for the full duration of its life. To partition an application during execution the Coign ADPS must compare each component (or component instantiating request) with profiles gathered from previous scenarios, searching for a match in order to determine its expected communications pattern. To achieve good performance, each instance must be matched with the most similar profile as accurately as possible, before the component is used.

Coign uses component context to build component classifications both while performing scenario profiling and during normal application execution. Component classifications are used to classify a single component instance within a single profiling scenario, across multiple different profiling scenarios, or a group of identical or similar instances in a single scenario or across multiple scenarios. Components need to be classified by not only their type but more importantly also by their behaviour or what their roles are within the application. Essentially, each component needs to be classified by its context, which primarily relates to its communication patterns. Classifying components during or after profiling scenarios is relatively simple as the communications patterns may be observed intimately during profiling; however, components instantiated during normal runtime must be classified before they are used, hence the classification must be performed using the application state. Coign performs dynamic classification by examining the Component Call Chain (CCC) which gives the exact hierarchy of component events that leads to the current instantiation.

Hunt compares the effectiveness of CCC based classification against various alternative classifiers including: incremental classifiers (based on instance counting), static type classifiers (based on component type), procedure call chain classifiers (based on function call stack), static-type component call chain (based on the types of the CCC), and other variations on the CCC. These classifiers may be evaluated in two dimensions: granularity of classification and accuracy of classification. CCC and its variants provide reasonable middle ground for the granularity: incremental classification provides the finest granularity by assigning each unique instance to a new class of its own — however this fails once the program execution departs from the profiled scenario; the static type classifier provides the worst granularity, assigning all like types to one category. The CCC based classifiers also use the most context information and hence can provide the greatest accuracy in classification. The CCC variants that use a longer call chain provide increased accuracy.

Once components have been classified and the application scenarios have been profiled, sufficient information exists to perform distribution analysis of the application. Coign performs this analysis by calculating the cost of communication for a given message, m, based on the latency of the communication and bandwidth required:

$$\text{Cost}(m) = \text{Latency} + \frac{\text{Size}(m)}{\text{Bandwidth}} \tag{3.1}$$

Minimisation of this cost is the set goal of the Coign partitioner. The potential to introduce compression on the communications (which would require compressibility analysis on the data as well as processor load considerations) is not considered.

Hunt considers several distribution algorithms all based on graph cutting techniques, in which the components and communications are modelled as a directed graph[91]. The component distribution problem maps well onto graph cutting problems for which algorithms exist. As accurate graph cutting for more than two subgraphs is provably NP hard, Coign only attempts a client-server distribution over two hosts.

The application is mapped as a directed graph in which each component is represented as a node and communication between components is represented as an edge, weighted by the cost of that communication (the difference between communicating remotely and communicating locally). Two additional nodes are added, representing the client and the server. These are tied into the graph depending on component-host dependencies as described below.

Partitioning of the application thus reduces to cutting the graph to separate the client node from the server node while at the same time minimising the total weight of the cut edges, using an algorithm such as Min-Cut Max-Flow. Additional constraints are easily added to the partitioning process as extra edges. Coign explictly has the facility to append extra edges to the graph to account for affinity and CPU usage constraints as follows:

- Components that are necessarily tied to operation on either the client or the server host are tied to the corresponding node by an edge of infinite weight. This may occur for components that rely on a resource only available on one of the hosts such as a GUI on the client or a file on the server.

- CPU usage or processor loading for the component running on client and server can be mapped onto the graph as two additional cost weighted edges connecting each component node with the client and server terminals. The graph cutting process must cut one or the other during partition. The relative cost of CPU usage to communications costs is represented by the relative weight of these additional edges to those representing the communications.

Figure 3.3 shows an example graph for an application consisting of six components, two of which are tied to the server and one to the client.

Coign was tested on three applications consisting of between 200 and 500 COM components. Coign was shown to be of little benefit to applications such as the image manipulation program test in which transactions between components are few but consist of very large data sets and include many non-remotable interfaces (that rely on shared memory structures for efficient passing of large data sets). In this case, Coign chose to place only the file access component and some related components on the server. However, for applications which are much more dependent on frequent, but small, communications between components, Coign can often realise a more even distribution of components which in some cases can reduce communications costs by up to 99%.

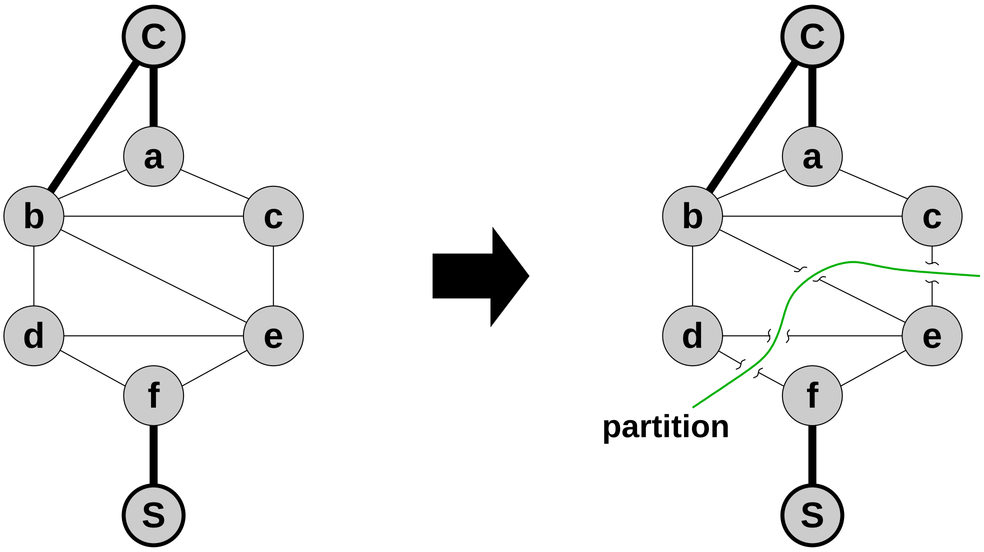


Figure 3.3: Partitioning an application through graph cutting. Heavy lines show the artificial bonds of infinite weight used to tie components to an individual host. To improve clarity, edges representing CPU usage are not shown.

In particular, Coign exhibits the following advantages over a partitioning scheme devised by the application developer:

- An application developer will often partition an application along logical grounds or language grounds whereas Coign partitions purely by cost. Partitioning by logical purpose (such as removing application logic from application interface functionality) or by language (the application logic may be in C++ while the interface is in Visual Basic) are obvious and easily maintained partitioning boundaries but can place artificial limits on the partitioning scheme in order to increase application maintainability. The Coign partitioning scheme however is applied to the binary and hence need not be explicitly coded or stated in the project source code or build process.

- As Coign operates on binaries, it needs no knowledge of the actual application source code and hence may be operated by users rather than developers. In this way, users can reconfigure the partitioning of their applications to suit their individual operating environments. Partitioning schemes devised by the developer can only aim to perform the best on average for all possible environments. This, perhaps, is the key advantage of Coign.

3.4 Algorithmic and Distributive Adaptation in Mobile Systems

Applications for mobile systems require more advanced adaptation services than their fixed counterparts in order to handle the increased volatility of resource availability including the possibility of disconnected operation.

This section describes several adaptation services aimed at the mobile environment. Each of these focuses on the communications bottleneck experienced by mobile applications but approaches the problem from a different angles, making different assumptions about the nature of mobile applications.

3.4.1 Mobile Application Framework

Hild's Mobile Application Framework (MAF)[36, 37] developed at the University of Cambridge Computer Laboratory provides a set of system background services which support wired applications running on mobile hosts. It is used to *mobilise* conventional, wired applications by transparently augmenting them with functionality similar to that found in dedicated mobile applications.

MAF is a *mobile enabler*, a set of small utilities and system services that address the difference between the high speed, low cost networks that most applications are designed for and the low speed, high cost networks available to mobile devices and applications. The MAF services allow 'wired' applications to function across such data links with little or no modification to the applications themselves.

MAF offers a form of application-transparent adaptation, primarily in the form of location-transparency and through bandwidth optimisation and cost control. Its adaptation is performed through condition testing and online algorithms and may be regarded as adaptation at the more primitive end of the spectrum. The adaptation techniques used are aimed primarily at stand-alone applications operating on a mobile host that may require sporadic data services from remote hosts. The effectiveness of the adaptation performed for a distributed type application may be characterised as weak-discrete.

The three major services provided by MAF are the *Call Manager*, the *File Logger* and the *Communications Scheduler*.

Call Manager

The Call Manager's contribution is twofold: first, given a communications task it identifies the most appropriate communications channel from those currently available; second, it analyses and optimises the outgoing communications traffic.

The first of these functions is provided via a dynamic routing service called *NetWatch* which operates under the mobile device's TCP/IP stack through modification of the system's routing table. NetWatch periodically monitors the available communications channels for availability, rate, cost and setup requirements and compares this with the system's present and future usage requirements (as determined by the Communication Scheduler). It uses this information to select an optimal channel and then dynamically reroute the traffic accordingly. The selection policy which balances the relative values of time (call latency or update period), communications bit-rate and financial costs is provided by the mobile user interactively.

The Call Manager's communications traffic optimiser is in the form of a collection of protocol specific *call handlers*. A *call redirector* intercepts TCP/IP communications and infers their function and protocol by inspecting the destination port number (*e.g.* 80 is typically HTTP, 21 is typically FTP). If a call handler is available for the particular protocol or service requested and is willing to accept the call, the communication is forwarded to that call handler, otherwise the communication is passed as normal traffic. Intercept/pass decisions may also be overruled by destination IP address based criteria.

The call handlers are protocol specific and can use their explicit knowledge of the protocols to perform some of the tasks more efficiently than through the usual TCP/IP communications over the data link. Example applications include a disconnected name service (*i.e.* a caching DNS server with a notion of temporary IP addresses) and an FTP handler (performs directory caching and, with the aid of a remote FTP gateway, ZModem transfers of actual files).

File Logger

The File Logger enables data sharing via a filesystem which caches a network file share and provides data consistency during disconnection and primitive file reconciliation upon reconnection. This differs from other mobile network filesystems such as Coda[82] which are derived from network file systems and allow for temporary disconnection in that the File Logger is designed for operation in a predominantly disconnected state.

The File Logger is a much more basic service than the complete filesystems provided by Coda, providing a simple user level file replication, reconciliation and update system. It collects changes to a set of nominated files and reconciles these with remote filesystems periodically in coordination with the Call Manager and Communications Scheduler. It provides feedback to the user indicating the divergence of the nominated files from their last update and requires user interaction to aid in conflicts that may occur during reconciliation in the case of multiple divergence or in reconciliation of non-trivial data.

Communications Scheduler

The Communications Scheduler makes network activation decisions, using user supplied costs and perceived utility of network availability to coordinate network connectivity from a range of connectivity options. It achieves cost minimisation through batching data transmissions to amortise setup costs and improve data compression schemes.

To perform this, the Communications Scheduler manages communications cost *budgets*, estimates expected future traffic profiles and applies priorities to various traffic schemes according to a set schema.

Applications

Hild demonstrates the utility of MAF through some sample applications: ftp, web browsing and email. The ftp transfer uses the FTP call handler as described above; the web browser's HTTP fetches are intercepted by a similar call handler which uses page caching, predictive pre-fetching of linked pages and image transcoding (compression). The email application adds priorities to the email headers and then uses in and out *email buffer* files and the File Logging service to reconcile these buffers with mail servers when connection is resumed.

These examples show that the user can continue to use standard applications with familiar interfaces in a wireless mobile environment with minimal loss of functionality. When connected to a high rate network, the system resumes operation as normal.

3.4.2 The Rover Toolkit

The Rover[46] toolkit developed at the MIT Laboratory for Computer Science provides developers of mobile applications with a set of tools to abstract over the limitations of mobile devices and systems, in particular the communications bottleneck. The toolkit allows developers to construct applications using a distributed object system based on a client-server architecture. Clients are Rover applications running on mobile (or fixed) devices; servers run on fixed hosts, may be replicated, and hold the *master* state of the system.

Rover offers a form of integrated adaptation, primarily in the form of location-transparency and secondarily through communications bandwidth, cost and availability control. Its adaptation is limited to simple disconnection support and prioritisation of some applications and services over others according to user preferences. The adaptation techniques use code and data mobility to support device mobility, exploiting sporadic connectivity to maintain duplicated state on both client and server. Rover provides facility for application developers to include type-specific reconciliation algorithms to counter concurrent object modification and discrepancies. The effectiveness of the adaptation performed for a distributed type application may be characterised as weak and/or discrete.

Rover supports mobility and communications disruption through two mechanisms: *Relocatable Dynamic Objects* (RDOs) and *Queued Remote Procedure Call* (QRPC).

Relocatable Dynamic Objects

A Relocatable Dynamic Object is an object with a clearly defined interface that may be loaded into a client host from a server in order to reduce communications requirements. Rover RDOs are implemented as Tcl/Tk scripts, providing cross-platform execution and a limited amount of security through *sandboxing*. Applications using Rover RDOs are hence best written in Tcl/Tk although it is possible to write them in a different language such as C or C++ but using a thin veneer of Tcl/Tk for Rover interfacing.

Each RDO may be a data object with some type-specific methods or a greater functional part of the application and has a *home* server which maintains its master state. They may be viewed as simple intelligent mobile agents[79, 30], encapsulating state and functionality into a mobile package. RDOs execute within a client host's object *cache*: a secure, controlled executive environment.

Rover applications operate on these objects using check-in, check-out semantics: first they must `import` the RDO into the local cache, then `invoke` one or more methods of the RDO, and finally `export` the RDO back to its home. RDOs in a host's cache are secondary copies, shared amongst applications on that host. Importing a RDO does not lock an object. Hence, a disconnected application that has imported an object may continue to make invocations on that object. Rover does not provide automatic object consistency if more than one client accesses an RDO concurrently. Object consistency must be provided by application-level locking or by application-specific coordination of disparate, conflicting invocations. When the system reconnects at a later time, the RDOs are exported back to their home servers. If update conflicts are detected at this time, Rover attempts to reconcile them, using type-specific actions if supplied.

Queued Remote Procedure Call

Queued Remote Procedure Call provides non-blocking asychronicity extensions to Remote Procedure Call (RPC)[9]. QRPC will queue any RPCs issued during periods of communications unavailability, permitting applications to continue to make non-blocking RPCs even when the mobile host is disconnected: the RPCs will be executed and responses collected when connection resumes. RDO fetches and commits are performed over QRPC.

QRPC provides for asynchronous data fetches as well as for *lazy* RPC execution. This allows applications to be decoupled from the underlying communications infrastructure: disconnection appears to an application as merely a very slow network.

QRPC may use a prioritised queue for out-of-order calling, prioritising between applications and also within an application. Callbacks provide upcalls to applications to notify the completion of a QRPC. QRPC includes a limited capacity to handle partial failure: queue entries are only cleared when a response is received. Partially executed calls will remain in the queue and will be re-executed.

Developer Support

Rover provides a toolkit for developing mobile applications; it does not provide adaptation or disconnection services to legacy applications. A developer uses the Rover functionality by encapsulating each of the application's data items as a Rover RDO. Each such object must be provided with appropriate access and modifier methods as well as conflict detection and resolution for each of the modifier methods. Often, the applications will be forced to resort to user intervention if it is unable to resolve inconsistent divergence of an object's state (as might occur during disconnection).

Porting an existing UNIX application requires replacing the file model with Rover's RDO model and dividing the application into client and server components. Generally, QRPC calls are not made directly but provide the underlying support for the RDO implementation.

RDOs are used in Rover applications to both represent basic application data and also application components. The Rover data access and sharing described above is relatively simple and natural. Joseph[46] continues to describe the use of RDOs as application components, giving several examples: an application can dynamically download an RDO representing the application GUI, using a different GUI depending on the mobile device characteristics; a server could send a decompression algorithm along with compressed data — the decompression may be encapsulated within the RDO; exporting client computation to the server — useful when performing filtering or searches over large server based data sets.

Applications

Joseph[46] describes some sample applications implemented using the Rover toolkit. Like those implemented for MAF, the application example set consists of ports of common user productivity applications: an email client, a calendaring tool and a web browser proxy.

Rover Exmh is a port of *Exmh*, a Tcl/Tk based email browser. It was modified to run in client-server mode, using RDOs for both list and message objects. Each list object represents a mail folder and performs pre-fetching of the RDO message objects it contains. The user interface is modified to show via colouring the caching status of each message. The resulting application provides a balanced compromise between the message delivery and temporary storage facilities of a POP mail server and the message access and permanent storage of an IMAP mail server. POP does not provide a permanent storage mechanism but is merely a mail delivery service and hence cannot provide a permanent mail store accessible from several clients. IMAP provides a permanent mail store with search and indexing facilities but is not designed to be used in a potentially disconnected environment. Using RDOs as the message bearer provides shared message access while the caching and resynchronisation facilities allow continued operation during disconnection.

Rover Webcal is a port of *ical*, a Tcl/Tk based calendaring and scheduling program. It is modified to use RDOs to represent calendar items and events, providing a finer granularity over the default file-locking basis of the original. This allows much wider modifications of the calendar even during disconnection, while minimising conflicts.

Rover Web Browser Proxy provides a caching and pre-fetching HTTP proxy to conventional web browsers running on the client. The cache status and control is exposed to the user through a list interface, allowing direct user intervention in cache operation. The proxy is not in itself very novel but opens the way to RDO base documents which are dynamically generated on the host, for example to ship dynamic documents or fragments of databases to remote hosts.

Summary

In summary, Rover provides basic integrated adaptation to applications through object migration, focusing on persistent data sharing and concurrent access, but with the addition of facilities for true function migration. The example applications demonstrate that

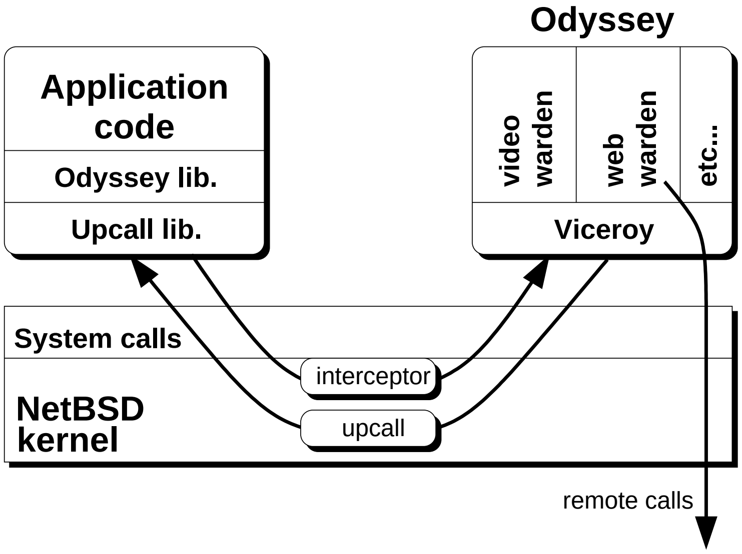


Figure 3.4: Odyssey client model.

existing applications can be easily modified to provide utility even in a disconnected environment. While Rover provides much of the mechanics for object migration and support, it relies heavily on application developer support for object locking and consistency as well as for client-server distributions.

3.4.3 Odyssey

Odyssey is a platform for *Mobile Data Access* from Carnegie Mellon University that provides adaptation services promoting *application-aware* integrated adaptation for mobile applications[64, 66].

Odyssey inherits much of its design from the Coda Distributed Filesystem, also from CMU[80, 81]. In particular, it continues Coda's strict client-server partitioning, Virtual File System (VFS) based call and data in-kernel interceptors and user-level implementation. It builds on Coda's *volume* based file management, using type-specific *tomes* as its data transaction devices.

Odyssey provides facilities for applications and the operating system to collaborate on the adaptation process: the system is responsible for providing the mechanisms for adaptation while applications set adaptation policy.

Figure 3.4 shows the basic Odyssey client platform. It can be seen that the bulk of Odyssey runs in user-space, providing flexibility whilst sacrificing only a small amount of latency. Kernel modifications consist only of system calls and signals to perform upcalls to the applications and VFS interceptors to handle type-specific data. These constitute the two key mechanisms employed by Odyssey to provide integrated adaptation: upcalls are used to notify applications of changes in system resource availability while VFS interceptors allow the inclusion of type-specific data *wardens* to manage data resources.

Upcalls and Resource Management

The bulk of the Odyssey core comprises the mechanisms provided by the system, the structural framework or architecture containing those mechanisms, and the application programming interfaces to those mechanisms that allow applications to construct and implement adaptation policies.

Odyssey centres around providing information to applications concerned about resource availability. Core to this is the concept of an upcall: an event notification service which serves notices of resource changes to applications[67].

Odyssey uses a generic resource model for system resources, including battery power, disk space, bandwidth etc. System resources may be either *universal* resources with no frame of reference, or *item-specific* resources which refer to the resources associated with a particular context (such as bandwidth which is available only in the context of a client-server pair). Odyssey supports four universal system resources (disk space, CPU cycles, battery power, money) and two item-specific resources (network bandwidth and latency). The current Odyssey implementation and example applications focus on adaptation to variations in network bandwidth — this is often the most restrictive resource in a mobile system and provides the greatest opportunities for adaptation.

Applications register interest in resources with the Odyssey *viceroy*, requesting notification for resource changes that would interest them; they need to specify the resource type as well as upper and lower bounds on which to trigger a resource notification. Nobles asserts that most applications operate within a resource *window of tolerance*, defined as a range of possible resource conditions for which it will not adapt — most applications are expected to implement discrete adaptation policies and hence only register to receive notifications at distinct resource window bounds. Overlapping windows can be used to implement hysteresis for applications that achieve adaptation through mode switching.

When an application receives an upcall notifying it of such a change, it can respond by varying its behaviour accordingly. The system does not dictate how this should occur — it is entirely application dependent.

The viceroy provides a central point for all resource management. It performs monitoring for resource availability and handles resource reservation for applications. It cannot currently control an application's use of the resources but can only provide it with an estimate of how much of each resource it may use. In addition to these tasks, the viceroy is responsible for managing data resources. These include object management, communications control and operation with the type-specific wardens.

Type-Specific Data Handling

To control resources efficiently, Odyssey mandates a single point of control in the system for all resources. Hence, the Odyssey platform itself performs all caching, transport and consistency decisions. Naturally, such decisions are best performed with knowledge of the data being dealt with and hence Odyssey incorporates type-specific *wardens* at the system level.

Type-specific wardens offer two key advantages over generic bit-stream handlers:

firstly, the warden can arbitrate between two or more applications that require the same data — it can recognise and aid sharing and deliver improved performance to both; secondly, a greater understanding of the data allows the warden to perform caching, replication, or fabrication of data in a sensible manner not possible without knowledge of the data types involved. Unfortunately, the downside of this approach is that a new warden must be developed for each new data type, placing a further burden on the developer.

Data resources in Odyssey are characterised by a single *fidelity* metric assigned on an arbitrary and obviously type-specific quality. The wardens impose the fidelity measure on the data resources according to the data and type. Although fidelity is a quantitative measure, Odyssey does not impose any restrictions on the value except to say that it should be a monotonic sequence increasing with perceived quality.

Applications

Each warden is responsible for a single type of data. As an example, Noble describes Odyssey wardens and applications for video, web browsing and speech recognition.

The video file player *xanim* was modified to access video data from streams instead of files. The streams are pre-processed and stored as a series of tracks, each at a different fidelity, which are then served by a custom video server as byte streams (the tracks could equally well have been degraded on-line by the server on demand). The video warden performs all of the data requests and fetches for xanim, providing a track listing as well as frame by frame access.

The xanim client accesses the video through the warden. First, it must determine the range of fidelities available and then calculate the bandwidth required to sustain each different fidelity and the appropriate windows of tolerance for the resource. During playback, if xanim receives notification of a change in bandwidth, it can select a different track at a different fidelity.

Other applications include a *web browser proxy* much like that developed using Rover in section 3.4.2 and a speech recogniser. The web proxy is a variation on the xanim video warden in which a web proxy performs page fetches for the web browser. However, the Odyssey proxy uses a remote distillation server to compress or reduce the quality of images before they are transmitted. The quality or size of the image is the fidelity metric in this case. The proxy varies the fidelity of image fetches according to bandwidth availability.

The Janus speech recogniser (also from CMU) was ported to operate on the Odyssey platform and demonstrates the use of a more sophisticated type-specific warden. Janus takes as input phrase utterances recorded from the microphone and processes them to return the text of the words spoken. It is a very expensive process in terms of CPU usage and memory. It is fundamentally different from the previous two applications as it is not so much concerned with data access but rather data is generated at the client and then processed, with the results being presented at the client.

However, the speech recognition process naturally breaks into two stages, vector quantisation to perform initial coding and then the rest of the recognition process, which allows partitioning into a client-server application. Taking advantage of this feature, the

Odyssey implementation creates a *speech warden* which takes utterance data and returns text data. The warden can do one of three things with the utterance: it may process the utterance locally using Janus, process the utterance remotely using a remote instance of Janus, or process the utterance in a hybrid fashion, using the local instance to perform the vector quantisation and pass this smaller data set to the remote instance to complete the recognition. As all three modes of operation produce identical results, the fidelity metric remains the same for all; the only differentiating factor is the latency involved in the recognition. Selection between the modes is performed based on the available bandwidth. Further work may differentiate the modes by using a smaller dictionary in the local case. Unfortunately, the tests carried out only occurred in an environment in which the bandwidth available always remained within the window for the hybrid mode and hence this was the only mode of operation.

These individual application examples do not themselves show the benefits of the Odyssey platform above providing resource monitoring. Further tests running two or more of these applications concurrently show the advantages of having a central point of resource control. Noble demonstrates several combinations including multi-high bandwidth (video) and mixed bandwidth (video and speech) application sets. These tests show that having a centralised, unified resource estimator and allocator achieves higher performance on the average for all of the applications than individual bandwidth estimates. However, the sharing mechanism used allocates more bandwidth to applications that have used more in the immediate past, so the sharing is not inherently fair, shown in the case of two competing video streams in which one consistently performs at a higher fidelity and with less dropped frames than the other.

Summary

In summary, Noble makes a strong case for integrated adaptation using centralised management for both system and data resources. Results show that centralised management of system resources can provide much more accurate resource estimates to applications and hence reduce competition and increase overall performance.

There are several design faults in the general division of duties in this scheme. Firstly, the xanim client is forced to perform all of the resource finding and adaptation itself. The warden is there to support type-specific adaptation, yet the client must still explicitly request track changes. In this case, the client is not just specifying policy but also implementing the mechanics of it. The warden acts only as a blind fetching agent and the viceroy only performs bandwidth measurement.

Additionally, the task of measuring a video stream's fidelity is complicated by the existence of three obvious metrics: frame rate, frame size and colour depth. The access to these tracks only delivers a one dimensional fidelity which is arbitrarily decided by the warden developer. In this case both warden and application are by the same developer but in more general cases this would not be so. Noble gives the example of the video player and video editor both accessing the same video data (as described in section 3.5.1 above) where conflicting notions of fidelity exist. Noble points out that transparent adaptation in the proxy cannot resolve such conflicts and hence integrated adaptation is required.

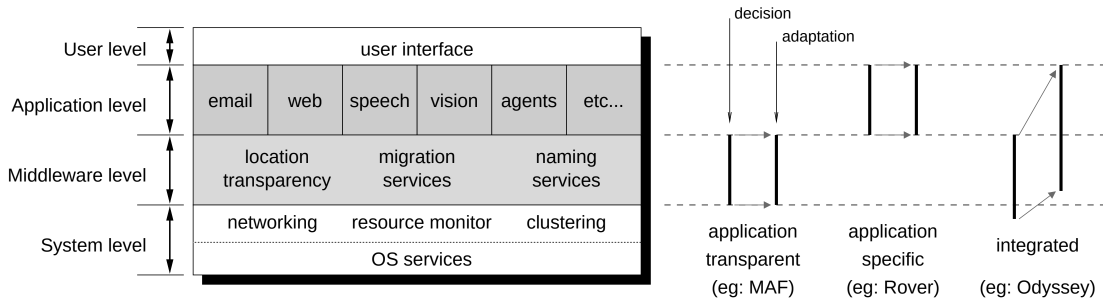


Figure 3.5: Decision positioning and adaptation areas of Application-Transparent, Application-Specific and Integrated adaptation models. Each pair of bars shows the scope of operation for the corresponding model: the first shows the scope of the data collection used to make an adaptation decision while the second indicates which areas the decision affects.

Odyssey places the bulk of the adaptation mechanism inside the application along with the adaptation policy. The application is responsible for its own adaptation and ensuring that it remains within its resource bounds. This adaptation process integrates with the system via the common resource management provided by the Odyssey viceroy. This model provides meaningful and efficient adaptation for applications at the cost of requiring type-specific data handling and explicit adaptation details to be handled by the application developer.

3.5 Adaptation Models

An adaptation model describes the structure and policies of an adaptive system. It defines the levels in a system at which adaptation occurs and also at which levels adaptation decisions are made. This section describes three broad divisions of possible models following those proposed by Noble[66] and argues that only a fully integrated, application-aware model can effectively provide adaptation services to a mobile system. Figure 3.5 illustrates the positioning of these models within the system stack, showing for each model both where the model gathers its decision data from and which layers it directly affects through implementation of its adaptation policies. Application-transparent, application-specific and integrated adaptation are described below.

3.5.1 Application-Transparent Adaptation

Systems following the transparent adaptation model perform adaptation at operating system or session levels in a manner transparent to, and concealed from, the applications. System services are modified to adapt to resource availability in a manner that is transparent to applications using that service.

An application-transparent model is beneficial in that existing applications may be run unmodified and new applications need not deal with adaptation issues. Central control of resources and adaptation avoids competition between applications and aids efficiency in resource use.

Examples of application-transparent adaptive systems include MobileSocket [70], the Coda distributed file system[80] and Hild's Mobile Application Framework[36], all of which provide services for sporadic or disconnected operation.

MobileSocket is an example of a connection/disconnection transparency layer approach to application adaptation. MobileSocket provides Java virtual machine session level support for TCP socket continuity of operation across host network disconnection and reconnection, even if the host changes network access environment before reconnection. The corresponding end of the connection must also use MobileSocket for disconnections and reconnections to operate. Connections to an existing service that does not use MobileSocket must be via a fixed MobileSocket-Socket proxy. In this manner, MobileSocket provides the most primitive of adaptive services: a TCP socket that does not time-out and that may be reconnected from a new address. Applications may continue to run, oblivious of any change in network connectivity.

Note that Mobile Socket also provides optional callbacks to allow for a degree of application-aware adaptation but its key mode of operation is that of a transparent Java socket replacement for unmodified applications.

Coda is a direct descendent of CMU's Andrew File System (AFS)[81, 38, 82]. AFS is a distributed file system designed for scalability in terms of both performance and administration. Coda adds two main features to aid mobile computing: high availability across network or server faults and application-transparent adaptation to smooth over varying network quality. Coda can provide filesystem data availability across sporadic or continual disconnection through data replication, caching and reconciliation: clients cache copies of files from servers and then operate on them locally; updated files are resynchronised with server copies as bandwidth permits. A local cache manager on each host manages this process.

Coda provides a much more sophisticated concept of adaptation than MobileSocket but is very *data specific*: Coda only operates on file-system level data.

Mobile Application Framework (MAF) (described previously in section 3.4.1) is a set of system background services which support wired applications running on mobile hosts. Together these tools and services allow the mobile user to use and control applications not originally intended for mobile use.

Other examples of application-transparent adaptive services include other data specific services and proxies, a popular example of which is the adaptive web-proxy. A mobile user that has a local adaptive web-proxy can use an unmodified web-browser that operates unaware of any adaptation taking place. The proxy is free to perform caching, elimination or fabrication of web data. In association with a suitable web proxy on the fixed network, it can perform scaling or degradation of web images or filtering of inline banner advertising.

Many resources other than network capacity can equally well be handled at system level in an application-transparent manner; it is common for laptop computers to spin down hard drives or turn off their display when they have been idle for some time. The system may even suspend its operation completely with little or no application notification.

While application-transparent adaptation can make application implementations simpler and allows use of non-mobile applications unmodified, such an adaptation model must operate without the application specific information that can help it make decisions concerning the best adaptation strategy. One data type may be used in different ways by different applications, as in the video streaming example given by Noble[66]. In this situation, a video-proxy service cannot know how it should adapt its operation for low bandwidth operation — it must guess. This is the fundamental limitation of application-transparent adaptation: it is unable to efficiently handle situations outside its very limited initial remit.

3.5.2 Application-Specific Adaptation

In contrast to the centralised approach of the application-transparent model, application-specific adaptation places the task of adaptation solely in the application with no explicit system support. While this allows each application to perform adaptation exactly to suit its needs, there is no cooperation or coordination between applications as such. Each application must compete with other applications for resources. In the absence of a system supporting prioritised resource access, an application that adapts will most likely lose resources to one which greedily assumes maximum resource availability; applications that could share common data do not necessarily do so. Examples of application-specific adaptation include:

Media streaming applications such as RealPlayer[76] perform adaptation by selecting media streams at different bit rates depending on available network bandwidth. This is performed independently of other applications' needs and even independently of other instances of the same software. If a different, high bandwidth application such as FTP were running concurrently, then RealPlayer would suffer reduced performance as FTP soaked up all available bandwidth.

Other applications may be implemented to perform individual application specific adaptation but examples are few as this involves rewriting the software (or creating it from scratch) and is especially uncommon for the limited mobile computing market. Most such adaptive applications, such as the media streaming examples above, are designed to adapt to the varying network conditions of very wide area and congested networks (such as the internet). Examples of such applications are generally seen in an end user environment rather than a research one.

Application-specific adaptation can work well when only one application is dominating the resources at any one point in time. This may work well in situations where only one application is running at any one time such as we might expect when using a small PDA or organiser.

3.5.3 Integrated Adaptation

The integrated adaptation model provides a composite of the two above models, combining the system management of resources with the application specific knowledge of exactly what is required by the application. This model allows multiple applications to run concurrently without competing adversely; resources will be monitored and allocated centrally by the operating system and adaptation performed by the application, resulting in efficient use of resources and meaningful adaptation for the applications. Systems employing integrated adaptation include:

Odyssey is an example of such a model where specific application data types are handled by specialised proxies (wardens) and applications receive upcall notifications of resource changes, instructing them to adapt.

Cadmium is a similar project which provides middleware support for mobility and disconnection [7]. Cadmium provides adaptation mechanisms to all levels from system level, through middleware and applications to user level. The system level provides basic abstractions including object manipulation and replication. The middleware manages this replication to ensure data consistency and availability, using strategies that can be supplied by the applications themselves and provides upcall notifications to the applications. The user is able to control the adaptation system by providing application hints and performance feedback.

Integrated adaptation requires especially tight cooperation between the applications and the operating system. To achieve this, most existing applications must be modified to run effectively with the system.

3.6 Summary

This chapter has introduced the concept of adaptation as a powerful solution to overcome resource variations experienced by applications in a mobile context. It has further provided a spectrum of adaptation techniques from simple condition testing through to introspective, evolutionary code and a set of classifications for characterising the adaptive behaviour of an application.

Table 3.1 gives a summary of the mobile and adaptive systems reviewed in both chapter 2 and this chapter. Clearly shown in the table is that coverage has been limited to adaptive systems operating in the lower half of the spectrum. These tend to be the more practical systems; non-application-specific solutions for evolutionary programming are not widely applied in real applications.

Communications bandwidth is the primary target of most mobile application adaptation strategies. Generic resource managers can also handle other resources such as CPU, memory, battery energy[25] or money, either individually or via collated general metrics (cost, performance)[83, 59].

Technology	Model	Condition Test	Online Alg.	Param. Alg.	Alg. Select.	Evol. Prog.	Characterisation
Mobile IP	T						n/a (support only)
IPv6	T						n/a (support only)
MobileSocket	TS	•					discrete
ICOPS,CAGES	T	•					discrete (static)
IDAP	T	•					discrete (static)
Coign	T	•	•				discrete (static)
RealPlayer	S	•	•	•			linear-discrete
MAF	TI	•	•				weak-discrete
Rover	I	•	•				weak-discrete
Odyssey	I	•	•	•			discrete (app. dependent)

Model: **T**ransparent, **S**pecific and/or **I**ntegrated

Table 3.1: Summary of mobile and adaptive systems reviewed.

Generally, adaptive mobile systems achieve adaptation through two key controls: manipulation of locality of data and processing and control of fidelity of data and processing. The adaptation mechanism relocates data to a local cache during disconnection, relocates processing to where spare CPU capacity might exist, and relocates both data and processing to bring them together to minimise communications overheads. The adaptation mechanism may also degrade the quality of data or of processing to reduce CPU use or reduce bandwidth requirements.

Three systems were presented that represent different approaches to adaptation for mobile applications: MAF provides connectivity and communications management whilst swapping between communications mediums as well as cost based caching and marshalling; Rover provides a data object model which provides disconnected operation, synchronisation on reconnection and remote data access; Odyssey provides unified system and data resource management allowing applications to dictate fidelity of operation.

Three models of adaptation are presented which categorise the distribution of adaptation responsibility between applications and the operating system. Two of the models, application-transparent and application-specific, represent opposing ends of adaptation approaches: application-transparent aims to minimise effort of the application developer (sometimes none) while maximising efficiency through centralised resource management. However, such a model does not allow the adaptation mechanism to have true understanding of the nature of the data or the applications, making meaningful adaptation difficult and requiring second-guessing of the application and user's underlying needs. The application-specific model does the opposite: the application performs all adaptation itself and hence ensures that it is meaningful, but it does so without cooperation between applications and hence applications competing for the same resource are not forced to share fairly. This can, in turn, lead to resource wastage and poor application performance. Integrated adaptation provides the compromise but requires tight cooperation between applications and system.

Initial Study: A Distributed Control System

4

With the proliferation of small sensor and actuator devices with built-in connectivity, building control systems with a centralised point of control presents appealing opportunities to reduce the operational cost of systems using these devices. This can be achieved through the use of a shared controller which at the same time may offer increased overall performance through the sharing of information and enhanced cooperation amongst devices. Such devices might be trivial sensor/actuator devices such as temperature sensors with built-in wireless communications or light or heating controllers for building automation. Alternatively, we might consider robot devices which contain only minimal hardware for sensing, actuation and communication. Such thin-client devices can be manufactured cheaply and then integrated into a larger system coordinated by a central controller.

This chapter describes a preliminary experiment with real-time control applications using an adaptive thin-client approach. The aim of the experiment is to determine the utility of such techniques in building control systems such as might be used for distributed robotics. This chapter is derived in part from [23].

The experiment involves the development of an architecture and runtime system to implement distributed control and data processing applications in a thin-client manner, suitable for implementing a thin-client mobile robotics system. The system varies control fidelity and locality to adapt a control application to changes in *quality of service* (QoS) availability and processing resources using a cost benefit model.

An example application is presented in which the architecture is used to implement the distributed control of an inverted pendulum over a shared, wired network. The experiment highlights the developer overhead of implementing adaptation even in this simple application and details the performance benefits achieved. Performance results are compared with non-adaptive distributed control approaches. This confirms the utility of control of process locality and fidelity as a means of providing application adaptation in a thin-client manner.

4.1 Distributed Control for Distributed Robotics

There are many situations in which several mobile robots must work together to achieve a united goal. The control of such systems is typically performed locally in a distributed manner to ensure a simple, scalable, yet robust system. However, such distributed control systems are less efficient and more restricted than a centrally served system. In many situations, a control system with a global view is required where a local view can not provide sufficient information for operation[74].

A thin-client control architecture is useful for multi-robot systems as the robots can be made to be cheap, plentiful and lightweight (in all aspects: mass, interface and power consumption) while a powerful server can control several such robots centrally. However, such a system is vulnerable to the vagaries of wireless and network communications as it requires bandwidth and latency quality of service provisions. For control applications, the raw signals to and from the sensors and actuators will often require greater bandwidth than higher level control abstractions. Additionally, geographic disparity between client and server locations and possible contention over a shared network means that communication latency becomes significant and often unpredictable. While some networks, such as dedicated real-time control buses, can provide some QoS guarantees, these are expensive in relation to a shared network. In addition, these are also often less efficient than a best effort network due to the reservation requirements and in any case the minimum latency or maximum bandwidth that the network can provide may prove to be insufficient in some situations, negating a robust server control design. This is especially the case for mobile systems that use radio networking where the communications characteristics can vary widely and rapidly.

To implement a real-time control system in a thin-client manner, we can take advantage of several distinguishing features of such a system to alleviate the restrictions imposed by the network.

Similar work in adaptation of distributed multimedia applications largely comprises varying the quality of the data by adjusting resolution and bit rate [3, 5]. Control and multimedia applications are similar in that they are both *analoguesque* in nature — the data involved is rarely discrete and can tolerate degradation of rate or fidelity. For example, if a control application or controlled apparatus is performing well, it may be able to reduce the issue rate of its control instructions. Similarly, if the control signals themselves or sensor data are large and scalable, they may be reduced in quality (*c.f.* frame resolution reduction for video signals).

However, while multimedia applications tend to focus on end-to-end streaming from a source to a sink and experience a latency bounded by the performance of the network, control applications characteristically operate in a loop where the sensors and actuators are physically co-located but logically separated by any interposed processing. Hence, a control application can theoretically achieve zero latency by short-cutting the network. This suggests another method to combat latency in the application: move processing to the client.

The thin-client configuration requires that processing should be performed on the server — the client may not be sufficiently powerful to perform its own calculations.

Thus, only lightweight processing could be performed on the client. Experience with thin client systems for robot football[22] has shown that even after sensor, actuator and communications processing is completed, often spare processing cycles are available for lightweight application use, so lower fidelity or lower quality processing could be performed locally during some periods.

Hence, the above points present two variables for adapting a control application to best operate given the resources available to it: fidelity and locality.

4.2 Adapting a Control Application

There are two stages to implementing an adaptive real-time control application in a thin-client manner as described. Firstly, the application must be designed and constructed in such a way that there is fine grained control of application fidelity and locality. This involves analysing the flow of data within the application and using this to decompose it into discrete modules which may be manipulated individually. Secondly, the application must have a runtime system/resource scheduler that is able to balance the available resources and then configure or adapt each client controller's fidelity and/or locality configuration to maximise performance for that client.

4.2.1 Modular Decomposition of the Application

The application is specified in an abstract manner to the runtime system as a collection of *functional elements* or *modules* which the runtime can then manipulate in order to adapt the application.

A functional element is a self-contained processing unit of the application that takes some input, processes it and then produces some output. An element may have no inputs or no outputs, representing a data terminal (*source* or *sink* respectively) in the application. An application is constructed from a collection of such elements by connecting their outputs and inputs with *wires* to form a *graph* of interconnected elements. Elements may also have more than one input or output, and hence the resulting graph may have *branches*, *joins* and *loops*.

As the functional elements are all independent and have well defined interfaces, their exact implementation and location can be made independent from one another as per many distributed system techniques such as CORBA or COM. The isolation of self-contained elements allows the runtime to:

- hot-swap modules dynamically (during the course of application execution), replacing one module with another which may use a different algorithm for a different processing/quality trade-off.
- migrate modules between client and server.
- do both of the above, concurrently.

In order to represent the application as a collection of elements, design techniques such as dataflow analysis[1] (as used in parallel software development) can be used to describe the application as a possibly branching chain of functional elements. The dataflow model views data as temporary values, produced by the source element and consumed by the destination element. The relationships between the computational elements are hence defined by the data dependencies between the data inputs and outputs of connected computational elements. Alternatively, Brooks' subsumption architecture[11] is a another example of an existing modular robot control design method that may be suitable for implementation in this manner.

In dataflow analysis, an application is decomposed into several functional elements where the combination of the functionality of these elements gives the functionality of the whole. A definition of an application in terms of such functional units is not sufficient, but requires an *implementation*. For any one function, there may be a number of different implementations; hence, an application description is comprised of a functional specification (in terms of functional elements) and an implementation specification (in terms of implementation code and also annotations to the functional specification). The annotations specify for each functional element which implementations are available, what their relative performance attributes are (their fidelity) and on which hosts these implementations are able to run. The implementation code provides the actual instructions to implement the functionality along with hooks for the developer to provide more detailed performance feedback and resource requirements at runtime. In a running application, a particular implementation may have more than one *instance* — for example an element may have the same implementation running on both client and server; in this case, there are two instances of the one implementation. Section 4.2.3 provides greater detail on the actual construction of an adaptive control application.

Care must be taken when decomposing an application into its constituent elements. In particular, attention must be paid to the potential for parallel execution in the data path. The passing of data is defined to be *asynchronous* so that data passing occurs once that data is available — no attempt is made to ensure that this occurs at the same time for all interconnects. The *push* model of data flow is used with no acknowledgements, leading to a fully asynchronous implementation. Although a *pull* or hybrid model would be equally suitable, it requires bidirectional communication for each data transaction, doubling the latency. With push, computation in an element blocks, waiting for input from its upstream data suppliers. In this way, the application can run as quickly as possible as it is constrained only by data dependencies, an important consideration when the processing will occur over more than one host with some degree of latency in between. This also allows an application constructed in this way to be automatically parallelised by the runtime system through exploitation of both parallel data flows and pipelined processing.

For a single, non-branching, non-joining stream of data (a linear application graph), the asynchronous nature presents little hindrance. However, for an application with data joins, the element which takes more than one input must consider which *firing rules* it should follow in order to coordinate multiple inputs[1]. These rules dictate what sequence of input events should trigger the calculations performed by the element (and thence produce output). The rules are not dictated by the system, as the optimal rule set will be application dependent, however some standard rules are offered to the developer:

Specific Input — trigger on receipt on a single input; other inputs should be stored for use by the operation until required.

OR — trigger on any one of several inputs; other inputs should be stored for use by the operation until required.

Ordered AND — triggers once input has been received on all input lines in a particular order. For two inputs: receive and store *a*; receive *b*; trigger processing of operations on *a* and *b*.

Unordered AND — when the order of the inputs is indeterminate, Ordered AND may group inputs incorrectly, possibly discarding some in the process. Unordered AND resolves this by counting the inputs (assuming that inputs always come properly paired). For two inputs: receive and store *a*; increment $count_a$; perform operation if $count_a$ equals $count_b$; receive and store *b*; increment $count_b$; perform operation if $count_a$ equals $count_b$.

Combinations — trigger on a combination of the above operators. *e.g.* trigger on (*a* AND *b*) OR (*c* AND *d*).

Different elements require different triggers. For example, consider an `addition` element that takes two numbers, adds them together and then outputs the result. This would require an AND type firing rule to ensure that it was always adding the correct pair of numbers. On the other hand, a real-time `audio mixer` element that takes two digital audio streams and mixes (adds) them together could trigger on a single input only as the exact pairing of samples is not required.

4.2.2 Runtime Support for Application Adaptation

The application runtime must perform the balancing between application performance and available resources. The runtime, as shown in Figure 4.1, uses the knowledge of the application, the network performance, and the client and server machines to determine the best layout of functional elements in terms of exact implementation and locality. It uses feedback from the application proper to determine the level of performance of the application. With the use of a prioritisation policy, it can balance resources between elements as well as between various clients.

The runtime uses a cost-benefit optimisation/tradeoff to calculate the dynamic application reconfiguration requirements. Every resource is given a cost function which can vary according to the demand and availability of the resource. The application provides a model which indicates the benefits it expects to attain from any given configuration. In this way, the runtime can optimise the configuration for best performance given a particular application budget or calculate a least cost configuration to attain a given performance level.

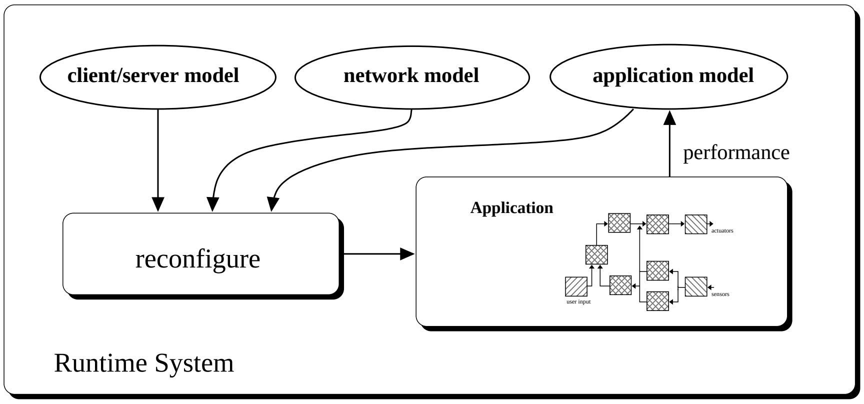


Figure 4.1: Application Runtime Support

The performance benefit gained via a particular configuration too is difficult to quantify and is inherently application dependent and so must be provided by the application itself. While the runtime can monitor communications usage between functional elements the application must provide hints as to how well it is currently performing and how well it could be expected to perform given a particular change in its resource allocation, *e.g.* relocation of some modules or change of data quality etc.

Because the module implementations are available at instantiation of the application, no actual migration of code is required at runtime. Logical migration occurs as one module is swapped into the place of another by redirection of the data flow either internal to one process or across the network, hence providing variation of locality. Typically, there will be two implementations for each module — one tailored for operation on the server (this may be a heavyweight process with large resource demands) and another implementation tailored for operation on the client (this will use a lighter-weight algorithm).

The runtime uses graph-cutting techniques to divide the chain of processing modules between client and server in the most cost-effective manner. The system described in this chapter uses cuts on a purely linear (non-branching) chain to partition the application dynamically. Clearly, this is only suitable to simple applications exhibiting a end to end linear control flow. More sophisticated techniques such as those used by Hunt[39] are required to partition more complicated application designs; these will be able to suggest configurations for data flow arrangements which require branching and rejoining of the data flow.

The runtime system is held separate from the client and server application code, acting as a middleware layer, performing data path connections, data passing and reconfiguration. A simple, custom integer passing mechanism is used as the underlying middleware for the system to facilitate the distributed inter-working of the application, routing all logical data connections over a single TCP/IP connection. Application reconfiguration decisions are performed at a single point in the system using performance information gathered from both client and server. The application elements provide performance feedback data to a special functional element which passes this data into the runtime system. The runtime system interprets this as a *locality hint* and performs element migration accordingly.

4.2.3 A Guide to Building an Adaptive Application

Considering the above, a dynamically adaptive application can be constructed by following the following procedure. The procedure follows the meta-level specification concept used for middleware systems incorporating an IDL specification. Here, the developer outlines the application structure and composition at a higher level before producing the implementation code details. This high level description allows the preprocessor to generate instrumented applications and to specify to the adaptation runtime system the exact characteristics of the application and restrictions on its reconfiguration. Specific examples of the specifications involved are given with the description the inverted pendulum controller in section 4.3.3.

1. perform dataflow analysis and functional decomposition to determine the application topology,
2. specify the data types to be passed,
3. specify each functional element class in terms of its input and output data types, implementation annotations and locality hints,
4. specify an application container that contains all of the required functional elements and describes the interconnections between them,
5. use an application description processor to produce a skeleton application in C++. Each functional element is generated as a C++ class,
6. subclass the element classes to provide implementations for client and server operation,
7. build the complete application code pools and link with the runtime library. It will build a separate binary for each host type (typically just client and server),
8. deploy the application code pools on multiple hosts. The location of the runtime reconfiguration element dictates the central reconfiguration control point. This will typically be on the client (as this is considered to be the essential host assumed to always be present).

An issue for further consideration is that of application or element *state*. An element's state is that quality of an element instance that reflects its operational mode or stage and differentiates it from other instances of the same element implementation. It is usually comprised of some internal variables of that element. A problem occurs if an element containing some state is logically transfered to operate on a different host or under a different implementation. Under the current regime, the state is not transfered and is lost. Hence, *stateless* elements are encouraged in which the element has no modes nor stages and the output is dependent solely on the inputs given. The operation of a stateless element can be easily transfered or hot-swapped dynamically. State could be transfered with the element but this becomes a non-trivial exercise to maintain state consistency and allow for the transfer overhead (which may be large). For these reasons, along with the observation that state plays only a peripheral role in many closed loop control systems, this experimental system does not explicitly support state but offers these workarounds:

- elements requiring a held state must be marked as 'non-migratory': *i.e.* restricted to operation on one host only. This forces there to be a single state instance and hence ensure state consistency.
- element state should be held separate from element function. This state may be contained in a dedicated *state element* which is marked as 'non-migratory'. Ideally this should provide state information via a pull data model but this can be emulated in the current environment using an emitter on a timed loop.

The sample application in this chapter fits well with the former method.

4.3 An Inverted Pendulum Controller

An inverted pendulum system is a classic control problem with hard real-time characteristics. Figure 4.2 illustrates the physical setup of such a system. The system consists of a pendulum mounted on a trolley which is constrained to movement along a track of finite length. The control aim for this system is to maintain the pendulum in an inverted, upright position with the trolley at a designated point. The inputs to the system are limited to forces applied to the trolley while sensors give feedback on the angular position and velocity of the pendulum relative to the trolley and the position and velocity of the trolley along the track.

The inverted pendulum system is highly unstable and is very sensitive to small control loop timing fluctuations or variations in feedback latency. This system may be termed a *hard* real-time system as failure to meet feedback deadlines can result in failure (collapse) of the system. The inverted pendulum is often used as a control problem for teaching or research purposes and as such, there exists many control algorithms ranging from simple proportional controllers to much more complicated adaptive or fuzzy-logic based controllers. However, the high susceptibility of the system to timing variations or latency effects means that control of such a system over distance or over a generic network has proved to be ineffective without real-time network infrastructure and delay compensating control algorithms[89, 63].

4.3.1 Distributing Control

The fact that the inverted pendulum is difficult when control loop time characteristics are unknown and varying makes it an ideal test case for this distributed real-time archi-

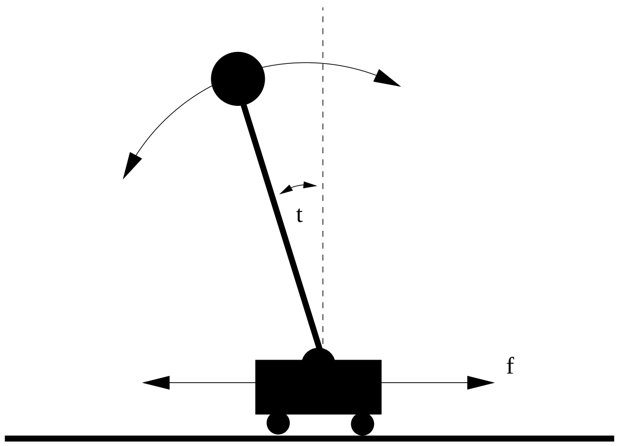


Figure 4.2: An inverted pendulum-trolley system.

tecture. The single client configuration of this example makes a client server controller a contrived and very much artificial implementation, however it still exhibits properties which make it a good demonstration case of the intended application areas of the runtime system while the hard real-time nature of the problem makes quantifying errors and performance easier.

In a thin client, thick server configuration, there are advantages and disadvantages to both placing control functionality in the client and placing control functionality in the server. Generally, the client benefits from having low latency while being limited in raw processing capability. On the other hand, the server has ample processing capability but is subject to the relatively large and variable latency of the communications network. The specific impacts of these aspects on the control strategy for an inverted pendulum system are detailed further in section 4.3.6.

4.3.2 Simulated Pendulum

The system was implemented using a simulated pendulum with controller components communicating via sockets over an Ethernet network using the distributed real-time runtime system described previously.

The pendulum proper is simulated in a process of its own using a discrete time simulation with a time-step period of $T_p = 2.5ms$ (400 iterations per second) as given in the equations below where f_t is the acceleration applied to the trolley at time t and G is the gravitational acceleration.

For θ being the angular displacement of the pendulum from the vertical:

$$\ddot{\theta}_t = f_t \sin\theta_{t-1} - G\cos\theta_{t-1} \tag{4.1}$$

$$\dot{\theta}_t = \dot{\theta}_{t-1} \times 0.99^{|\dot{\theta}_{t-1}|} + \ddot{\theta}_t \tag{4.2}$$

$$\theta_t = \theta_{t-1} + \dot{\theta}_t \tag{4.3}$$

For x_t being the position of the trolley on the track:

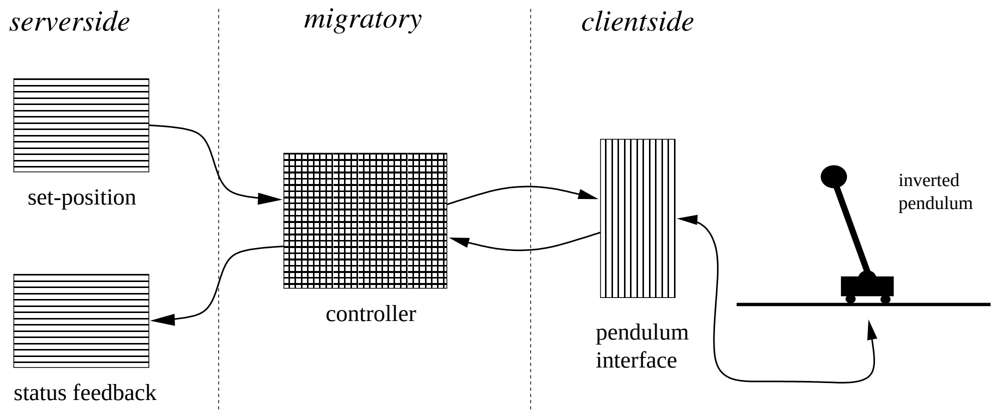


Figure 4.3: Pendulum Controller architecture

$$x_t = x_{t-1} + 100f_t \tag{4.4}$$

A friction element is included for only the movement of the pendulum and is a function of the angular velocity of the pendulum ($0.99^{|\dot{\theta}_{t-1}|}$). The rate of movement of the trolley along the track is unbounded in the simulated pendulum but is constrained to the length of the track (being 200 units in this case).

The resulting angle of the pendulum and position of the trolley are exported to the controller every iteration. Force inputs are received asynchronously, allowing the controller the flexibility of reducing its control feedback rate and also ensuring that simulation rate of the pendulum is not constrained by the processing rate of the controller.

4.3.3 Pendulum Controller

The controller is implemented in a thin client, thick server manner. In this case, the control algorithm is implemented as a single fully migratable processing element and a pendulum interface element which is of course constrained to operation on the client only. Other ancillary elements provide user interface components such as set-point control, performance feedback graphs, and runtime interface elements which allow the application to indicate the measured performance which is fed back into the adaptation system to trigger locality changes. This configuration is shown in figure 4.3 and as a dataflow model with instrumentation in figure 4.4. Figure 4.5 gives the application specification to be processed by the preprocessor describing this configuration. As an example, the element specification of the controller element is given in figure 4.6. Details of the exact syntax of these descriptions are given in Appendix A.

The migratory module hence has different implementations for clientside and serverside operation to reflect the low processing and low latency characteristics of the client. As the application processing is broken into only one stage, the granularity of adaptation is coarse and one dimensional but sufficient to demonstrate the utility of adaptation.

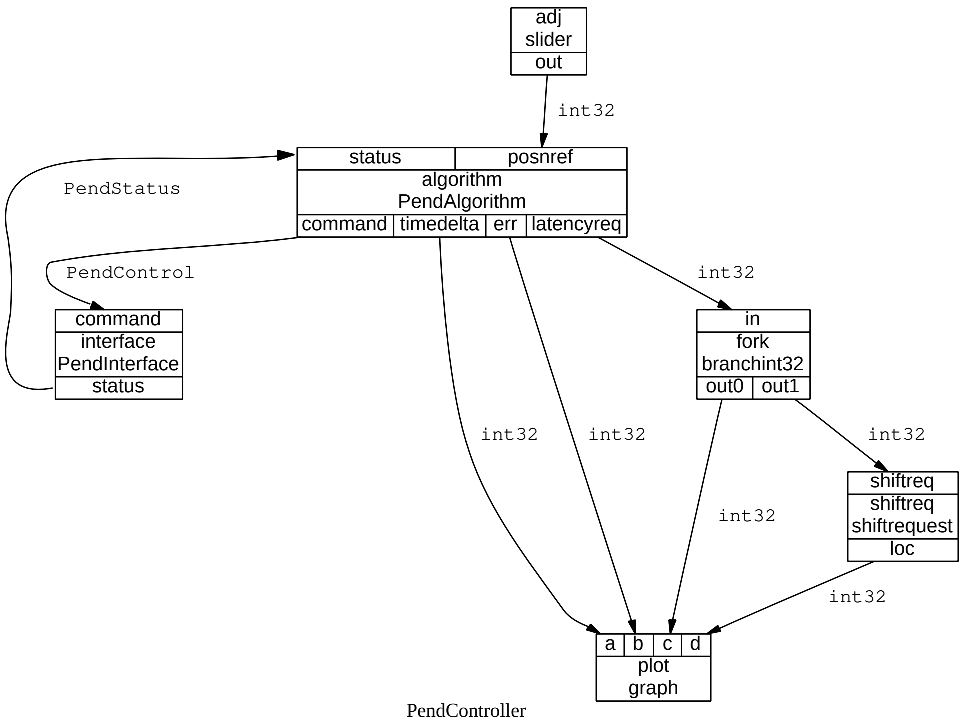


Figure 4.4: Pendulum Controller inter-module dataflow including profile instrumentation

4.3.4 Clientside Control

Executing on the thin client hardware, the clientside control algorithm needs to be simple but can take advantage of the low and fixed latency that exists between client and pendulum proper. For the client side, a simple proportional feedback control loop is used in which the force feedback is a value directly proportional to the angular deviation of the pendulum from upright.

This algorithm requires only three arithmetic operations per cycle which is easily achievable even on thin client hardware. The algorithm is able to take advantage of the known and small control latency to centre the pendulum with only small oscillations but is unable to handle more difficult conditions such as might occur when the trolley reaches the limits of its track — these may cause the pendulum to tip into an unrecoverable failure mode.

More sophisticated client-side control algorithms with similar computation requirements are possible in this situation but are not investigated further due to the limited scope of this experiment. A proportional-integral (PID) controller would perform well in this situation only requiring the additional storage of one variable.

4.3.5 Serverside Control

The serverside control algorithm must control the pendulum remotely and hence must handle the added complexity of varying latency in the control loop due to network char-

```
Module PendController {

    type container
    language cpp
    property string     serverside    General
    property dummyType  type          container
    property int32      application   1
    property string     clientside    General
    container
        module branchint32    fork        (347,278)
        module PendInterface  interface   (276,62)
        module PendAlgorithm  algorithm   (250,200)
        module shiftrequest   shiftreq    (472,312)
        module slider         adj         (29,149)
        module graph          plot        (579,216)
        wire   interface.status – algorithm.status
        wire   algorithm.command – interface.command
        wire   adj.out – algorithm.posnref
        wire   algorithm.timedelta – plot.a
        wire   algorithm.err – plot.b
        wire   algorithm.latencyreq – fork.in
        wire   fork.out1 – shiftreq.shiftreq
        wire   fork.out0 – plot.c
        wire   shiftreq.loc – plot.d
        modproperty adj.minval        –80
        modproperty adj.maxval         80
        modproperty plot.clientside    0

}
```

Figure 4.5: PendController application specification.

```
Module PendAlgorithm {

    type normal
    language cpp
    property string      serverside    Server
    property dummyType   type          normal
    property string      clientside    Client
        import  PendStatus     status
        import  int32          posnref
        export  PendControl    command
        export  int32          timedelta
        export  int32          err
        export  int32          latencyreq

}
```

Figure 4.6: PendAlgorithm element specification.

acteristics and process timing. Timing measurements revealed that the one way latency between client and server varied between $0ms$ and $20ms$ which is a variation of up to 8 sample periods, or up to 16 sample periods for the round trip. The server's control algorithm must be robust enough to be able to compensate for these discrepancies.

For these reasons, the server implements a variation of Sutton et al.'s ACE/ASE learning controller[8] originally implemented for a simulated pendulum system but also practical for a physical pendulum[45].

The ACE/ASE controller uses a reinforcement learning controller in which control is realised simply by specification of the control goals and allowing the adaptation mechanism to attribute positive and negative reinforcement to its control parameters based on its success/failure feedback variable, operating in a neural-network-like manner. The basic ACE/ASE controller structure as described in [8] and [45] is illustrated in figure 4.7. Here the three key elements of the controller are visible: the state decoder, the Adaptive Critical Element (ACE) and the Adaptive Search Element (ASE).

The state decoder maps the state vector received from the pendulum system into a fixed number of uniformly spaced regions, sometimes referred to as *boxes*[58]. These boxes represent rough categorisation of the pendulum system state. The output of the state decoder is a vector which represents the current occupancy of the boxes: the box representing the current system state holds a 1 while the others all hold 0 (zero). In this case, the state space is limited to two dimensions: pendulum position and pendulum angular deviation. These are quantised into 32 and 8 discrete states respectively resulting in a total of $32 \times 8 = 256$ states in the box vector.

The box vector is passed to the ASE and the ACE as inputs. The ASE is responsible for generating the control signal given the output from the state decoder and its own

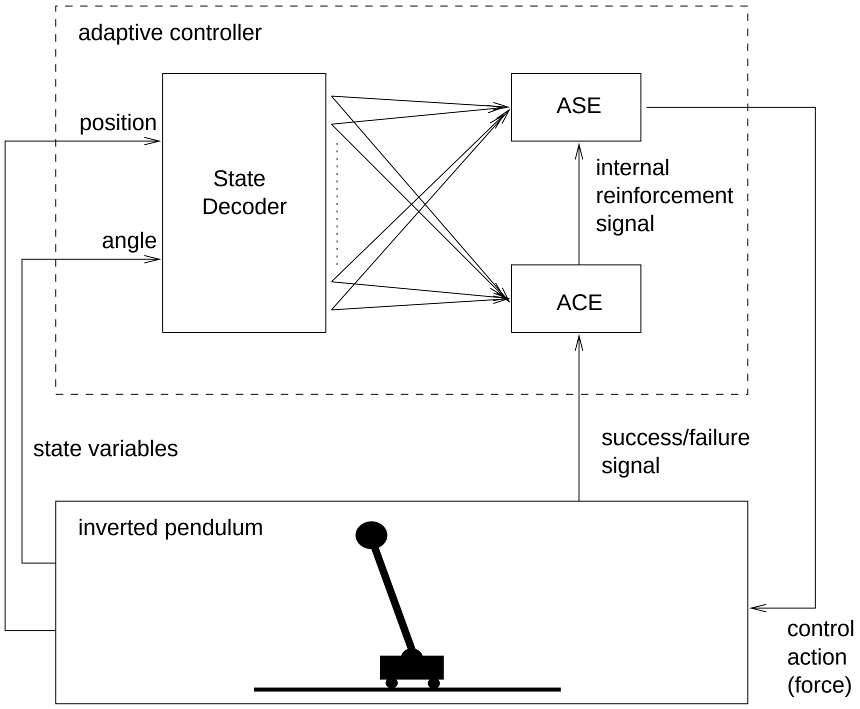


Figure 4.7: The ACE/ASE controller in place

internally held set of weights which apply to the box vector input. These weights are adjusted according to the internal reinforcement signal provided by the ACE. The corrections to the weights are applied using time decaying traces of the output of the ASE to ensure that the corrections are applied to the history of states which the system has been in and not purely the current state. The operation equations of the ASE, as with the ACE described below, are taken from [45]. The initial weighting values were derived from the appropriate actions which would have resulted if the above described simple proportional controller were implemented.

The ACE is responsible for the determination of *blame* or *credit* by manipulating the externally supplied success/failure signal in light of the history of the system in order to apportion proper temporal credit assignment. As the state of the system is more likely attributable to previous control actions than the current control action, this is an important feature to ensure proper assignment of blame or credit and to ensure correct learning of a successful control policy. This is achieved through a decaying trace of the system state history and a *danger map* which shows the expected danger or penalty associated with each state. The ACE adapts these danger values according to the performance and history of the system and then passes an appropriate internal reinforcement signal to the ASE. A more detailed explanation of the operation of the ACE/ASE controller is given in [45].

Several modifications are made to the learning controller implementation to allow for the simplicity of the simulator used here and also the learning environment. As this controller is intended to learn quickly and then also adapt quickly to changing network or communications characteristics, the following changes were made:

- The simulator only provides position and not velocity feedback. Hence, the state decoder need only partition a two dimensional state vector and not a four dimensional one. Having only two state variables makes the control task less stable and hence more difficult.

- The success/failure feedback is non-binary in that it gives magnitude of error as opposed to a simple success or failure indicator. This allows the ACE to adapt in a quicker fashion. The success/failure feedback is a measure of the performance of the pendulum system and is determined by combining the angular displacement of the pendulum from the centre with the linear displacement of the trolley from its set-point. The penalty contributed by the angular displacement is directly proportional to the angular displacement. That of the trolley displacement is exponentially proportional to the actual displacement. The penalties are weighted so that the angular displacement is the dominant feedback variable when the trolley is near its set-point while the trolley displacement will be dominant towards the ends of the track. The overriding concern of the controller when the trolley is towards the ends of the track is to avoid running off the end of the track with the upstandingness of the pendulum being only of secondary concern.

- Similarly, the internal reinforcement signal is non-binary giving the ASE similar characteristics. The high speed of initial learning is visible in the results.

- The control action output is also non-binary as opposed to the simple negative or positive output of the original controller. This too allows finer control of the pendulum system.

This algorithm is both processor and memory intensive in a manner that would probably make it unsuitable for implementation on a thin client device. Analysis of the algorithm reveals that for a state space of dimensions 32×8 (256 regions), it requires approximately 2048 operations per sample and a state storage area of approximately 1024 floating point variables.

4.3.6 Relocation Policy

From the above descriptions we can see that each of the two control algorithms have particular characteristics as outlined in Table 4.1. The proportional and adaptive controllers are suitable for clientside (local) and serverside (global) control localities respectively. However, each is limited in its capabilities in different areas. Hence, a suitable relocation policy would favour the clientside controller for situations where the communications performance is poor or when the server is loaded and the client is of low priority or only requires coarse control and would favour serverside control for other cases or for situations which the clientside controller cannot handle such as end conditions where the trolley hits the end of the track.

Proportional Control	Learning Control
• low latency only	• bounded varying latency capable
• not affected by communications variations	• substantial instability if communications latency varies wildly
• low processing requirements (3 operations per sample)	• high processing requirements (2048 operations per sample)
• zero state storage requirements	• substantial state storage requirements (1024 variables)
• reasonable control accuracy	• good control accuracy
• unable to handle end conditions	• can handle end conditions

Table 4.1: Proportional vs. Learning control

Hence, a suitable migration policy, and the one implemented here, is to migrate control serverside if the pendulum is near the ends of the tracks or is maintaining the pendulum upright within a certain tolerance and to migrate control clientside when the communications lag force the serverside controller to consistently cause instability in the pendulum system. A hysteresis function is used to avoid thrashing of the system (*i.e.* throwing control back and forth between client and server in rapid succession).

In summary:

- The clientside controller requests control shift to the server if the pendulum is held within 14 degrees (0.25 radians) of the upright position or if the trolley has hit the end of the track and fallen over. The client exerts a negative shift signal (-1) to request a serverside shift.

- The serverside controller requests control shift to the client if the pendulum falls more than 28 degrees (0.5 radians) from the upright position. The server exerts a positive shift signal ($+1$) to request a clientside shift.

- Control shift requests from both the clientside and serverside controllers are processed by the ShiftReq block. The shift decision variable s_t at time t determined by
$$s_t = 0.8s_{t-1} + 0.2s_r \tag{4.5}$$
where s_r is the shift request from the current controller module. The shift request then is only passed on to the runtime system if the absolute value of s_t exceeds a predefine threshold (± 0.5, *i.e.* requiring 4 consecutive requests to instigate a shift from a neutral state).

4.4 Performance

The application was deployed as three distinct processes communicating via TCP/IP sockets over switched 100Mbit Ethernet. The three processes comprise the simulated

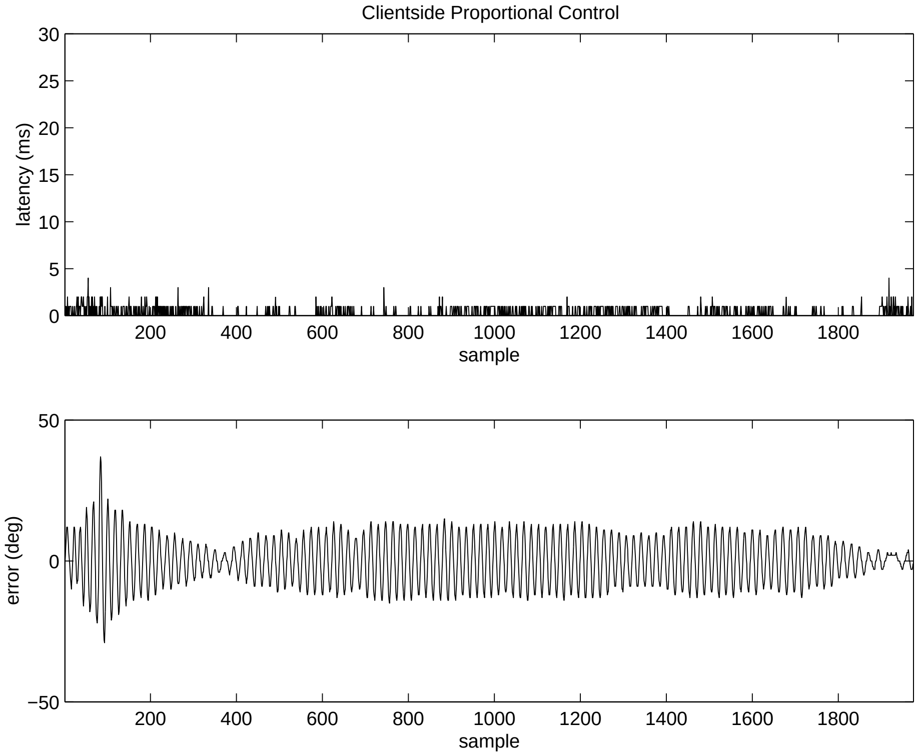


Figure 4.8: Clientside only operation showing latency (top) and error

pendulum, the application client and the application server. The following describes typical performance results for the simulated pendulum using three distinct configurations of the controller: client side only, server side only and dynamic adaptive mode.

Figure 4.8 shows the system in pure client mode where the control algorithm module is the simple proportional controller located on the client. The user interface and monitoring modules are still located on the server but communications with these do not impinge on the critical control loop. Here, latency is attributable to the task switching between controller and simulated pendulum processes. It is low (average 0-1ms) and jitter is low so the simple controller performs consistently with a steady-state rms error of 7.8 degrees from upright.

Figure 4.9 shows the system in pure server mode where the control algorithm module is the learning controller located on the server. The learning characteristic of the algorithm is visible as the high error region near the start point. In this case, the latency varies rapidly between 1ms and 21ms. Operating the simple proportional controller under such timing jitter conditions is infeasible and produced no recordable results. With the learning controller, after the learning process has stabilised, the rms error is approximately 6.2 degrees.

Figure 4.10 shows the system using application performance feedback to dynamically reconfigure itself. Included is a plot of clientside activation where a 1 indicates execution on the client and a 0 indicates serverside execution. As the ACE/ASE controller performs poorly at first while learning, the adaptation mechanism shifts control

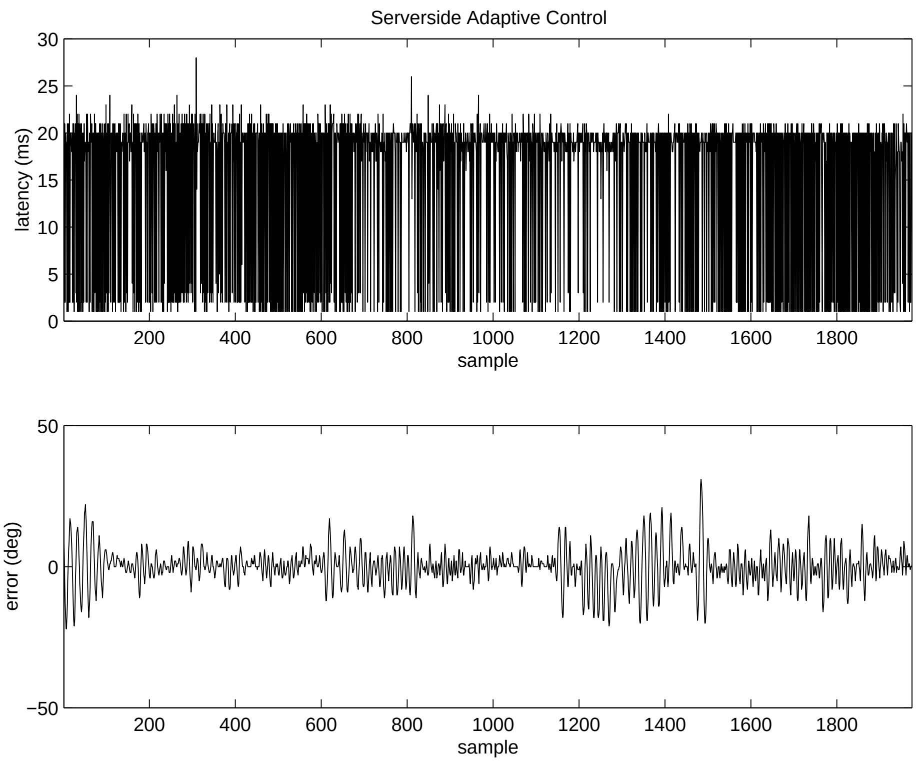


Figure 4.9: Serverside only operation showing latency (top) and error

towards the client, adversely affecting the learning process, resulting in the large initial error at startup. However, after this slow start, the system settles into primarily serverside operation. Sudden spikes in the error measurement result in transfers clientside for rapid correction after which control reverts serverside. Using this adaptation, the rms error (after initial learning) is 4.8 degrees, an improvement of almost 24% over the purely server based controller.

Figure 4.11 shows a side by side comparison of the rms errors with the nominal 24% decrease in rms error for the adaptive controller over the server based controller. While the decrease in error is important, the key point is that the adaptation mechanism is able to reduce the error and increase reliability in the face of varying network and application performance by trading locality (and hence latency) for processing capability (hence fidelity).

4.5 Summary

This chapter has outlined a development methodology and runtime system designed to aid the construction of dynamically adapting systems. This has enabled the development of an experimental sample application, resulting in an application which adapts to the available processing and network resources by dynamically reconfiguring the lo-

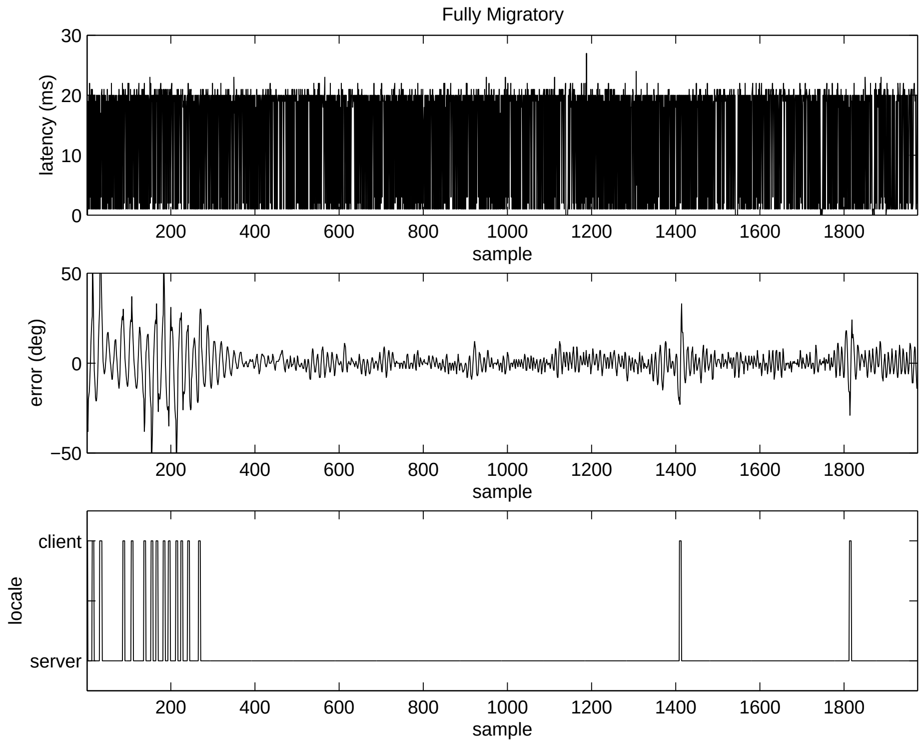


Figure 4.10: Adaptive controller operation showing latency (top), error (middle) and clientside activation (lower)

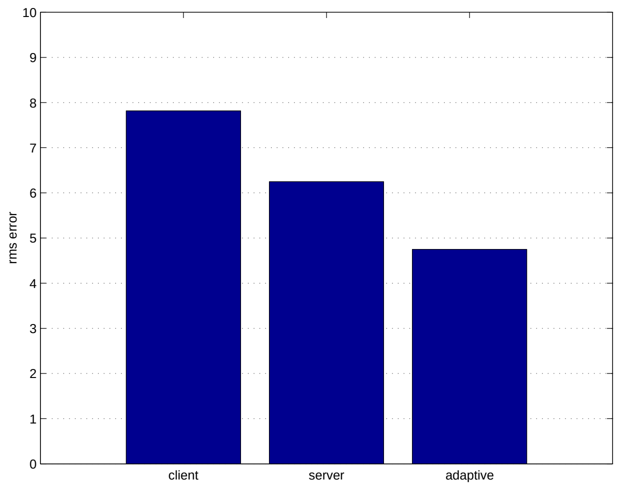


Figure 4.11: Comparison of rms errors for client, server and adaptive configurations

cality and fidelity of operation.

The example application of the distributed control of an inverted pendulum, while slightly contrived, illustrates that dynamic reconfiguration of application fidelity and locality can be used to effectively combat latency and other resource issues in a distributed real-time system. The conclusion then is that adaptation can be usefully applied to a real-time control problem in a dynamic thin client environment, displaying distinct performance gains over static application topologies in the same environment. However, the controller and runtime still offer much scope for improvement. In particular:

- The granularity of the adaptation is poor and could be improved by breaking the controller into smaller elements. However, the disparity between the natural decomposition of the learning ACE/ASE controller and the proportional controller means that new abstractions and mechanisms would be required to allow hot-swap substitution of a single element proportional controller with a multiple element equivalent learning controller (*one-for-many* substitution). With such an abstraction, further refinement of the granularity may be possible.

- The adaptation dimensions of locality and fidelity are strongly coupled in this example. A further example should decouple these by allowing as many elements and as many implementations to operate freely on as many hosts as possible.

- The runtime system does not explicitly understand latency which is the key environmental characteristic determining system performance in this case. Instead, this is implicit in the dynamic performance feedback from the application. Latency is difficult to directly optimise for in general but is resolved through optimisation of other criteria (reducing bandwidth usage or maximising dynamic performance metrics).

The methodologies and runtime system developed in creating this control application reduce the onus on the application developer by separating the concerns of operation and functionality. However, distributed control applications such as the inverted pendulum controller are not well suited to adaptation and thin-client operation due to their high level of dynamicism and sensitivity to latency. Less dynamic control applications which are less susceptible to such variations or more general purpose, softer real time applications however could be adapted using these techniques. The tools and system could equally well be made more general and then applied to applications outside of the control domain that exhibit similar characteristics. The system lends itself to applications which have the following properties:

- the client is thin and interested in preserving or minimising its own resources,

- are control-like in that their data and may be subject to degradation of fidelity,

- where inputs and outputs are generally co-located on the same host, and

- where communications characteristics of the client-server channel vary widely.

Furthermore, in order to act as a general dynamic adaptation platform, the system requires the following advances:

- a more general resource management system allowing the application to be adapted to other varying resource constraints such as processing capability, network availability and power consumption,
- a more advanced partitioning algorithm capable of dealing with graphs exhibiting arbitrary connectivity,
- a hierarchical one-for-many, many-for-many element equivalency allowing fine grained adaptation of multiple implementations with vastly different internal structures whilst also increasing element re-use,
- some element state management facility,
- a more general data type system (wider ranging than the current integer-only system), and
- a notion of multi-application co-adaptation to allow concurrent applications to adapt cooperatively.

The following chapter describes such a runtime system: the DPROJ framework, developed to aid the constructions of applications for mobile or wearable devices. These applications and devices fit the above criteria of being lightweight with limited power and having varying levels of network access as they move between access mediums.

The DProj Framework: Concept and Design

5

Chapter 4 concluded through experiment that fine control applications can be unstable as a result of large and varying latency. To be effectively implemented in a thin-client manner, dedicated QoS provision is preferable. However, while this applies to some mobile or remote control applications, it is not true of mobile applications in general.

This chapter describes the DPROJ framework developed to implement the principles of chapter 4 and hence provide adaptation for mobile applications. The aim is to be able to simplify the development of adaptive mobile applications and enhance their functionality beyond that available if operation were constrained to the mobile device alone. The framework aims to facilitate applications that can harness resources beyond the mobile device without making assumptions about or being limited by their varying availability. The framework is integrated in a system structure that allows the execution and mutual adaptation of several such applications concurrently. As such, the framework provides a means for multi-host, multi-resource, multi-application synergistic adaptation.

This chapter outlines some of the guiding principles behind the development of the DPROJ framework. The following sections describe the intricacies of the actual framework design, including how it is expected to be used and how it may be expected to perform. Chapter 6 goes on to describe actual implementation details with corresponding application development procedures and guidelines. Chapter 7 describes performance tests including results of experiments using the framework for simple fundamental test cases and the implementation of a sample mobile application.

5.1 PRINCIPLES OF OPERATION

The following principles of operation outline the basic features and mechanisms of the DPROJ framework. These points raise some usage and usability issues and reflect on how these affect the framework's eventual design and implementation.

5.1.1 Adaptation Mechanics

As per the experimental system of chapter 4, adaptation occurs through control of the application's parameters in the two dimensions of *locality* and *fidelity*. Control of locality involves migrating the execution of functional elements to a host where resources are available. Control of fidelity involves exchanging one functional element for another to perform the same task in a different manner or using a different algorithm which has a different resource demand. This may involve trading off resources (*e.g.* bandwidth against CPU usage) or degradation in quality (*e.g.* using a less expensive image processing algorithm):

> **Principle 5.1 (Adaptation Mechanics):** *Adaptation can be performed through control of the locality and fidelity of application execution.*

5.1.2 Remote Service Access

Remote service access covers a greater scope than remote data access. Adaptive application infrastructures such as Odyssey (section 3.4.3) focus on providing remote data access across varying network conditions. Remote service access involves facilitating application access to external processing hosts and services. Creating adaptive applications which employ remote service access involves creating distributed applications and distributed systems.

Remote service access enables applications executing on limited hosts such as mobile devices to perform beyond the capabilities of that host alone. As described in section 2.1, mobile devices are often limited in resources such as processing power, battery power and communications throughput due to the constraints imposed upon them by their mobility requirements. Extending application scope to execution on remote hosts via remote service access can allow increased application performance and new applications not possible on an isolated mobile host. This leads to the second principle:

> **Principle 5.2 (Remote Service Access):** *Remote Service Access should be a key feature of any application framework. Remote Service Access is envisaged to be as important as remote data access to next generation mobile applications.*

Further to this, it is the author's view that such remote functionality will in the future be offered as a service by third parties, most likely on a commercial basis. In fact, some examples of this are already emerging:

- spellchecker.net offers online spell-checking and grammar checking for free although it is sometimes slow. This can be useful for mobile users as such services require large dictionaries and large memories and processing (for grammar) making it ideal for implementation as a remote service.

- in a similar vein, Babelfish[6] provides online text translation services capable of translating between several European languages.
- WorkSpot provides internet access to Linux hosts which users may use to run their own applications. While this is currently a free service (and suffers from heavy loading), they also offer *application rental* services and potentially other charged services[99].
- SourceForge provides remote project build services for open source projects[87].
- In a reverse sense, there is the possibilty for users to provide services to the internet public at large. Java-market[2] allows any host with a web browser to connect and offer their Java Virtual Machine for others to use — this is both simple and secure. Alternatively users can deploy their own Java applications onto the offered hosts. Similar endeavours with commercial backing such as Popular Power[75] or Parabon[73] deploy certified native code applets onto client hosts and pay a small amount to the owner of the client host for their contribution.

While most of these examples are very specific and one-off batch processes, some such as WorkSpot and Popular Power represent commercial user-pay services that serve as models for future third party service provision. This *market model* leads to principle 5.3:

Principle 5.3 (Third Party Service Provision): *Remote Services may be provided by third parties on a pay per use basis. The framework should be prepared to incorporate such possibilities into its decision making.*

Such extended capability provides greater opportunity and scope for mobile applications, particularly with the advent of new third generation wireless infrastructure (section 2.2) which provide high rate, always on communications. However, constructing an application to use such resources is a non-trivial exercise. The application is now a distributed application and must be written to cope with latency, failure, network disruption, network disconnection and many other distributed system issues[94].

5.1.3 Pervasive Adaptation

In describing software strategies for portable computer energy management, [54] state that application adaptation requires a mix of application and system information and hence should be provided as an intermediate layer (middleware or runtime support). Similarly, [27] claims that integration of adaptation mechanisms from the OS to middleware to application level will provide more effective adaptation and operation. As discussed in section 3.5, only integrated adaptation is able to provide fair and efficient distribution of resources in a system.

To perform effectively, an integrated adaptation mechanism requires information from both the operating system level and the application level. Figure 5.1 shows the

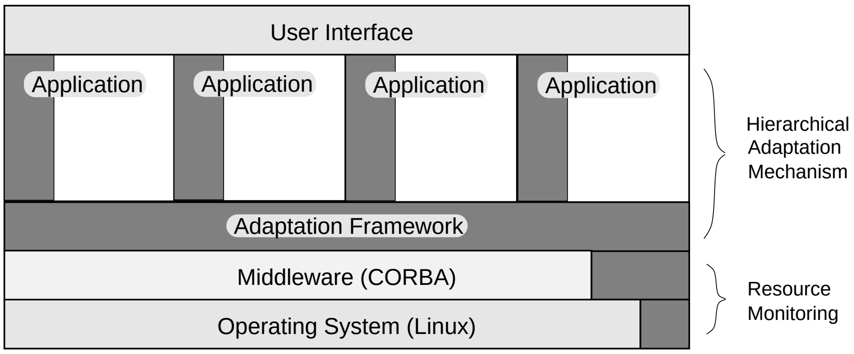


Figure 5.1: Pervasive Adaptation mechanism intrudes into applications space and operation.

positioning of a pervasive adaptation framework within the system stack. The framework acts primarily as an intermediate, middleware, layer but intrudes into application and system layers: it gains information about resource use and availability from the operating system, while from the applications it can gain information about resource demand and application performance. A mechanism without either of these sources falls under the application-transparent or the application-specific adaptation models.

By being placed as an intermediate layer, the adaptation arbitrator can draw upon all information sources and provides a single point of determination to ensure fair and effective distribution of resources amongst applications. However, such a layer can only provide the application with adaptation hints and cannot provide a useful adaptation mechanism for each application. Integrated adaptation should mean adaptation at all levels, not just pooling of information from all levels. Such integrated adaptation operating at all layers may be termed *pervasive adaptation*, embodied in principle 5.4:

> **Principle 5.4 (Pervasive Adaptation):** *Monitoring, decision making, and adaptation are required at all levels of the system. Only pervasive adaptation can bring about effective integrated adaptation.*

Pervasive adaptation must affect not only how applications use bandwidth but also how they use processing resources: spanning all configurations from fully distributed to standalone. With roaming connectivity and service provided commercially by third parties, adaptation must also provide adaptation against monetary costs, and for mobile systems, other limited resources such as battery power must also be accounted for. The pervasive adaptation mechanism must provide adaptation for all layers at a fine granularity against multiple concurrent resource demands.

Pervasive adaptation presents some difficult architectural challenges to the implementor. In particular, the major challenge is in gathering information across many layers, aggregating this into a comprehensible form without losing too much information and then finally performing adaptation at all levels in a maningful manner with minimal intervention into the operation of the application. Above all, this must be performed through some tractable and hence scalable means. Fortunately, the structure and components of a computing system exhibit a high degree of self-similarity when examined at different levels of detail or at different operational layers. In the same way that a system is composed of several applications which must share resources, an application is composed of several functional elements and each of those elements may be further subdivided into finer functional elements etc. This scale invariance means the same adaptation mechanism can be used internally to an application, internally to a functional element or to provide resource distribution and mutual adaptation amongst applications or even between systems, resulting in a hierarchical adaptation mechanism that operates from top-level down to an arbitrarily fine level:

> **Principle 5.5 (Self-similarity):** *Self-similarity in the system structure should be exploited in the adaptation framework to allow the same adaptation mechanics to operate and inter-operate at a range of system levels.*

5.1.4 Monitoring and Modeling

Closed Loop Control

The mobile system environment is expected to be highly dynamic with large and frequent variations in the availability of resources. To ensure *agile* adaptation in such a variable and often unknown system, the framework must rely on a closed loop adaptation control system:

> **Principle 5.6 (Closed Loop Control):** *Adequate control of the applications' adaptation process can only be performed using a closed loop feedback adaptation controller.*

Agility implies responsiveness, the ability of the system to respond to changes in its environment and operation. However, the system also requires stability to avoid unwanted thrashing between similar but conflicting configurations. Stability and responsiveness are contradictory qualities which can be resolved only through using system feedback to control adaptation.

The reconfiguration process is reliant on feedback to guide it towards the best configuration. The framework's models of the applications and the systems can be weak at best given the unknowns involved and the unpredictability of the system. Chains and nests of feedback loops allow iterative approach to the ideal configuration. Each iteration feeds back into the evaluation and process for the next configuration iteration. The characteristics of the feedback loop will determine the balance between responsiveness and stability in the system.

The feedback loops exist in several strata within the framework. Notions of performance, cost and use all rely on each other and the resultant interactions during reconfiguration will require multiple iterations to settle on an acceptable configuration. The cost function of each resource is controlled by a negative feedback loop: as resource use escalates, the effective cost of that resource will rise accordingly to drive down demand and offer the resource only to those functional elements that truly need it.

Application Feedback

Application feedback is vital as the performance of an application is inherently application specific[64]. The application author must provide some feedback information or general performance metric as only the application is able to measure this very application-specific quantity. The application may have to refer to the user, directly or indirectly, to determine its perceived performance, hence:

> **Principle 5.7 (Application Feedback):** *The application must provide explicit feedback to the adaptation system concerning its performance characteristics.*

This feedback is a single arbitrary quantity. If a particular application has more than one performance metric, then the developer is responsible for combining them into a single one, perhaps utilising user input to help this. For example, when dealing with a video stream, metrics may be frame rate and resolution; the application or user can then balance how important frame rate is versus resolution and various elements will then alter their cost-performance curves appropriately, perhaps forcing a new configuration.

Resource Competition

It is expected that whatever resources are determined to be available are available to the framework exclusively, *i.e.* freeing resources in one area will make them available to another. This would not be strictly true in a competitive shared network environment.

> **Principle 5.8 (Fair Resource Allocation):** *It is necessary to assume fair distribution of resources between competing systems.*

In effect, it is necessary to assume that either:

a) the entirety of a resource is allocated to the adaptive system. Hence, the determined resource availability is the actual resource limit.

- This *is true* for resources such as mobile battery power, mobile host CPU availability and some peer to peer or allocated slot wireless communications (*e.g.* GSM).
- This *does not apply* to resource such as shared server CPU availability and shared medium wireless communications (*e.g.* GPRS or waveLAN).

or

b) the resource consumption of the adaptive system constitutes a minor part of a much greater resource. As such, the global resource availability may be expected to maintain a steady state and any such variations in the determined resource availability are due to local conditions and local use of the resources.

- This *is true* for massively shared resources such as internet backbones, public servers and widely shared wireless communications (*e.g.* GPRS).
- This *does not apply* to any resource which is shared amongst only a small number of systems.

In the intermediate case, where neither of the above assumptions hold true, a mechanism which can provide fair resource service is required. Fair resource allocation involves the distribution of resources to those systems that need it most, and not to those that merely request the most resources. See principle 5.10 for further details.

5.1.5 Metric Unification

An important goal is to abstract over the specifics of any particular resource or any dimensionality to performance metrics. This enables the resource use and performance of an application or a system as a simple cost-benefit relation in two dimensions. Reducing the dimensions of the decision space simplifies the reconfiguration process. As the concepts of cost and performance are both very fluid, it is difficult to evaluate how this impacts upon the adaptation process. Section 5.4.2 elaborates further on this point. Reduction of the decision space to a cost-benefit function requires the unification of resource costs to a single linear currency and the unification of performance measures to a single linear metric.

Resource Cost Unification

The number and variety of resources depends greatly upon the exact system for which the framework is being deployed. In some systems the user may be concerned only with managing use of bandwidth while on others they may be more concerned about achieving the most financially cost effective application configuration. While the basic resource set of interest will be the same across most systems, the framework core should aim to be resource non-specific and function equally well without explicit knowledge of the exact characteristics and interrelationships of the various resources.

To achieve this, the framework core implements generic resource handling to deal with resource collation and demand processing facilitated by a unified resource cost metric. In this scheme, each resource is handled by a resource specific handler which performs resource monitoring and conversion of resource cost into a common metric based upon the prevailing policy driven equivalence functions (see figure 5.2). Following the market model of section 5.1.2, the chosen unified metric for the DPROJ framework is that of money:

> **Principle 5.9 (Resource Cost Unification):** *Resources should be unified into a single currency to maintain a simple, generic framework core.*

And further, implementing a real, hard currency, market arbitration for resources will ensure that resource price inflation will adequately control resource use. Real penalties ensure that the system will behave predictably in a competitive environment:

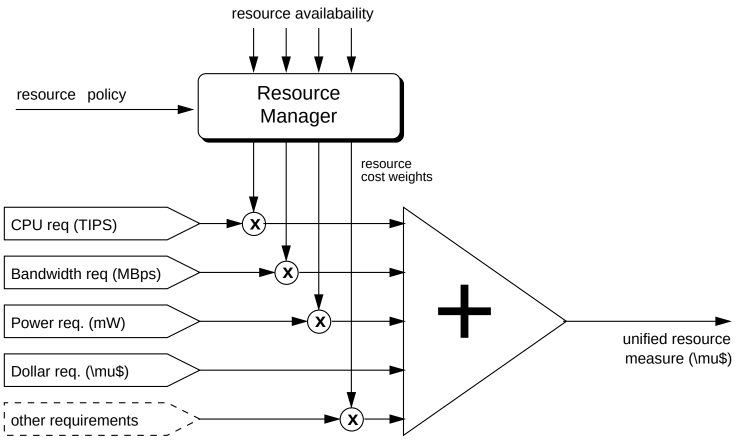


Figure 5.2: The Resource Monitors provide equivalency curves to reduce multiple resource demands into a single currency.

Principle 5.10 (Real Penalties): *Costs incurred should be real costs and not notional costs.*

In this scheme, all resources are reduced to their financial cost in dollars (actually microdollars). This works well for communications usage (which especially for wireless services is a saleable commodity) and for remote processing services (which can be charge-per-process style services).

Other resources map less easily onto the currency metric. Resources such as local CPU time or battery power have a much more abstract inherent value which becomes much more dependent on the user. To combat this, user input may be used to value these resources: *e.g.* a simple slider per abstract resource allowing the user to set the relative value of that resource. For a battery resource, users may select a low cost (*e.g.* I'm on the train to work where the battery will soon be recharged) or a high cost (*e.g.* I'm lost in the Amazon and I'd really like to keep my navigation software running as long as possible) as appropriate to their particular situations. Theoretically, it would be possible for a user agent to handle these details for the user: an intelligent agent might analyse the user's work or travel patterns and thus estimate how long the battery should be made to last — such a scheme may be interfaced with the framework but remains beyond the scope of this work.

Performance Metric Unification

For many applications performance is a multi-dimensional quantity and the dimensions involved or at least the measured dimensions involved may not even be orthogonal. Despite this, any one particular application configuration can represent only one point in that performance space and the range of configurations available can represent only a series of points in that space. These points will form a monottonic range from lower overall performance to higher overall performance. As the adaptation reconfigures the application within its configuration space, the resulting performance will be in this range in the performance space — the values in this range could then be termed the *Unified Performance Metric* and be derived from the multi-dimensional performance metric using an appropriate transform:

> **Principle 5.11 (Performance Metric Unification):** *Performance metrics should be combined to a one dimensional abstract performance metric to maintain a simple, generic, application-independent framework core.*

Performance is an abstract quality difficult to grasp even within the context of a single application or functional element. It may be also have different meanings for different applications and aggregating total application and system performance becomes a fuzzy area. Reducing performance measures to a common dimensionless metric and baseline can simplify the process without overly compromising the utility of the measure. A streaming video player is a common and useful example of an application with multi-dimensional performance: such a player's performance is characterised by both the video frame rate and the video frame size. The overall perceived performance of the player is a combination of both of these measures: a high frame rate at a low size is just as poor as a low frame rate at large size. The unified performance metric would be some combined measure of the two of these reported by the application.

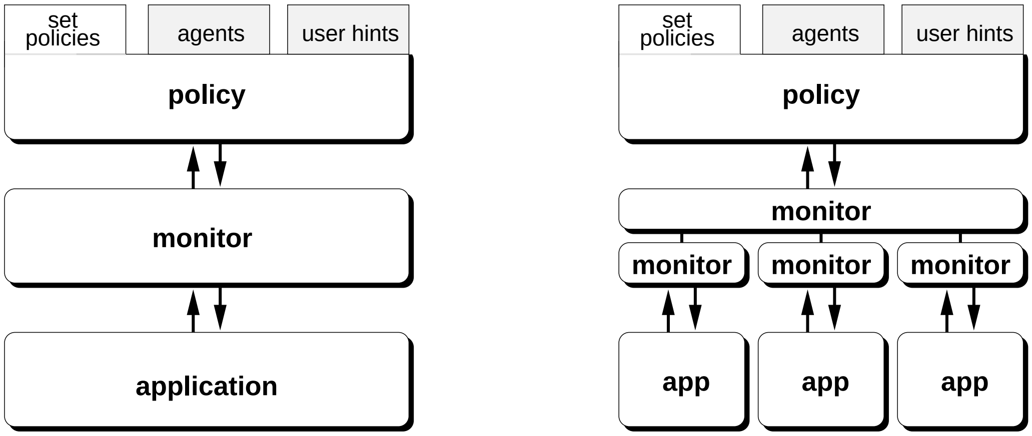
Devolution of performance metric unification to the application or functional element removes any application specific notions of performance from the framework. A video player would be required to reduce the two performance measures into a single metric. It could use a simple conversion (performance $= A \times$ rate $+ B \times$ size) or maybe something more complicated based on the user's scaling of the display framebuffer (note that the displayed resolution and the streamed resolution may not necessarily be the same).

This simple unified performance metric allows combinatorial operators to determine the overall performance of related functional elements, the overall performance of unrelated applications executing simultaneously and also the relative performance benefits of alternate functional elements without needing to discriminate between differing notions of performance.

5.2 Design Overview

The DPROJ framework provides a set of system components to facilitate and orchestrate application-aware adaptation for suitably constructed applications. This section describes these components, their interactions and their respective responsibilities. The framework is designed to follow the principles outlined above and these are referenced appropriately.

At a high level, the DPROJ adaptive platform is viewed as a three tier system. At the base are the applications themselves, upon which is layered the *Application Monitor* which performs the modelling, analysis and control of an application's reconfiguration;



(a) general layer view

(b) horizontally integrated layer view

Figure 5.3: Three tier DPROJ platform architecture.

this reconfiguration is controlled by an *Adaptation Policy* component (figure 5.3(a)). A symmetric hierarchical approach is taken to both application and monitor architecture providing for multi-application adaptation to be integrated into the framework using the same mechanisms and policies as for single-application adaptation (figure 5.3(b)). The policy component may be driven through set policies, hints from the mobile user or through user agents. DPROJ focuses on the adaptation mechanics and so the policy components covered here are simple and held detached from the the monitor. Similarly, application service discovery, host classification and elemental migration are only provided for through rudimentary middleware approaches. Object migration and platform heterogeneity are not driving concerns.

Figure 5.4 augments figure 3.5 of section 3.5 with the positioning of the DPROJ adaptation decision and adaptation scopes within the system stack. This illustrates its integrated approach in which both system and application layers are present in the overall adaptation scope. This differs from the integrated approach of Odyssey in that the scope of data collection is intended to be as wide as possible so that information concerning application performance and resource consumption and user preferences is collected in addition to the resource monitoring operations as performed by Odyssey. Additionally, the adaptation scope is restricted to the application itself — adaptation is not performed on system services. This is more a matter of semantics than limitation: whereas Odyssey implements application specific services as middleware or system layer services, DPROJ does not encompass any application specific services but instead considers an application to be the complete end to end data path, including both server and client components.

The framework aims to deliver a separation of concerns between the application functionality, the adaptation mechanism and the adaptation policy which is essential in maintaining a generic application adaptation system.

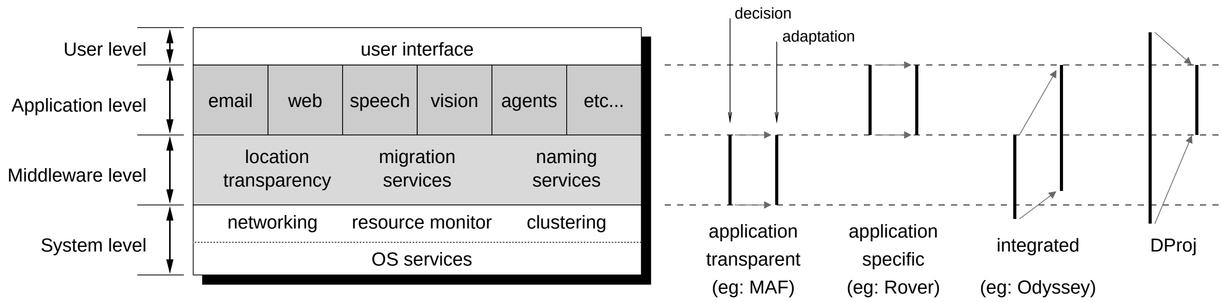


Figure 5.4: Comparison of the DPROJ integrated adaptation approach with Application-Transparent, Application-Specific and Integrated adaptation models. Each pair of bars shows the scope of operation for the corresponding model: the first shows the scope of the data collection used to make an adaptation decision while the second indicates which areas the decision affects.

5.2.1 Application Decomposition

The core of the adaptation process lies in principle 5.1 (Adaptation Mechanics), the control of an application's fidelity and locality of execution. This requirement to be able to substitute or relocate functional units of an application leads to an obvious breakdown of applications into functional elements.

To meet this requirement, the author proposes a hierarchical, modular approach to application design and deployment. In this approach, each application is analysed to determine the dataflow within it and is then designed as a graph of connected *functional elements* which pass data between them asynchronously using a common middleware platform. With well defined interfaces, each of these functional elements may be relocated or replaced by the runtime system dynamically during the execution of the application. This design is a logical progression from the simple system outlined in chapter 4, adding generic data type support and a hierarchical structure.

This design is modular in that each logical functional element is self-contained and the internal operations of that element are hidden from external interfaces, following common object-oriented ideologies. Following principle 5.2 (Remote Service Access), some of these modules may be located remotely and this also fulfils principle 5.3 (Third Party Service Provision) as these modules could equally well be provided by third party providers.

To achieve a fine granularity of adaptation, it is required that an application be decomposed into as many elements as possible. However, it is envisaged that over-decomposition of an application may detract from overall performance due to the increased overhead of the instrumentation and adaptation process. Some compromise between the two is advised.

The design is hierarchical in that each logical functional element may be either an atomic processing unit in itself or be a *container* for a number of contained functional elements which each pass data to each other and also communicate with the *outer* functional elements. Of course, these contained elements may themselves be containers for

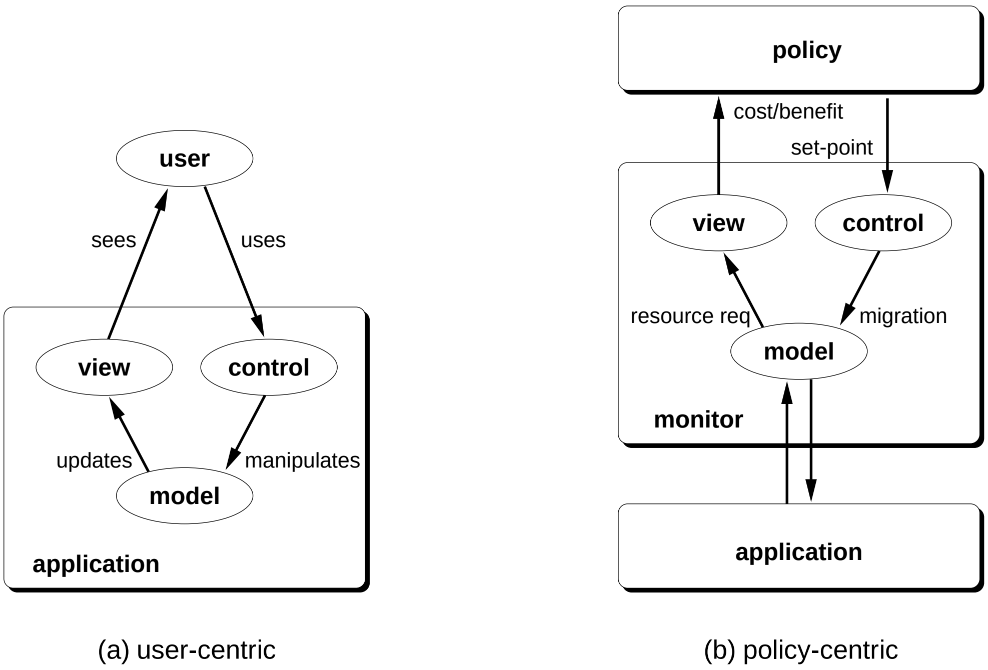


Figure 5.5: Model-view-control patterns applied to application development and application adaptation.

other elements. In this way, an application is constructed hierarchically from basic elements in the same manner in which an application might be constructed from an abstract hierarchy of objects or components. The application itself is represented as a single, top-level container element. As adaptation via relocation and substitution can be applied to any of these elements at any level in the hierarchy, the framework supports Pervasive Adaptation (principle 5.4) while the hierarchical structure and common element interface between application elements and applications themselves supports the Self-similarity requirement (principle 5.5).

5.2.2 Application Monitor

Following the separation of concerns, the actual adaptation of an application is performed by an external *Application Monitor*. The monitor constructs and maintains a model of the application which is then used to generate suitable adaptation decisions. The Application Monitor is based on the familiar *model-view-control* pattern (MVC). Traditionally, the MVC pattern is used to describe an applications structure as oriented towards a user (MVC originates in the Smalltalk-80 user interface). Figure 5.5(a) shows this arrangement where the model represents the application core maintaining state and data; it updates the view when the model state changes. The view presents this information to the user who may then interact with the controller to manipulate the model.

Figure 5.5(b) shows the mapping of this pattern onto the application adaptation model where the MVC applies to the application *Monitor*. This mapping serves as a convenient translation of terminology to better fit with the adaptation process where the term 'application' refers to the application being adapted.

Here, the model remains at the core of the pattern, but reflects the meta-state of the application. Using application meta-data, the model is built to echo the structure of the application. Each element in the application has a corresponding *proxy* in the model. This proxy gathers statistic information from the element concerning resource use and performance — it can do this by direct query or through instrumentation of the application. The resource and performance metrics are unified and presented as a cost-benefit function for that element. Cost-benefit functions can be aggregated at any level of the hierarchy to produce a summary cost-benefit view of that level. In this way, each level is presented in the same form and hence the same observers and actions can operate at each level whether it be a view of an application subsystem, an entire application or the entire mobile system incorporating several applications.

For a single application, the resulting view is a cost-benefit function describing the overall estimated cost for perceived benefit of all useful combinations of modes of operation of the application. This is a reduced and simplified view of the application model which can easily be used by the *policy* component to make a decision: the policy chooses a particular cost-performance point from the cost-benefit function according to the particular aims of the policy. It returns this decision to the controller which is propagated back through the model, resulting in reconfiguration of the application and completing the control loop (principle 5.6 Closed Loop Control).

The monitor iteratively updates the model using the current resource availability determined from system resource monitors and application performance feedback from the applications themselves (principle 5.7 Application Feedback). As performance is taken from the application elements as a single dimensionless quantity, this hands off the task of performance metric unification to the components themselves (principle 5.11).

In reality, the implementation of the MVC pattern might have the model, view and controller more tightly coupled than good software engineering practice would allow but this more intimate association can provide better scaling and distribution of the monitor to more closely match that of the application.

5.2.3 Adaptation Policy

The adaptation policy performs two key roles, providing policy for:

1. deciding how resources are balanced against one another, and

2. deriving an application operating point from the application view presented to it.

Although the policies for these two roles need not be related, they are both separate from the core of the adaptation mechanism and are likely to be context, user and application dependent. These policies are not the focus of this thesis so only simple examples are provided.

The resource balancing policy dictates the relative importance of various resources available to the system — it provides equivalence functions outlining how resources should be prioritised should there be an opportunity to trade usage of one resource for usage of another to achieve lower cost or higher performance operation. These equivalence functions serve to unify the resource costs to a single metric (principle 5.9).

For example, an application streaming video from a remote server to the mobile device may have the option of using a range of different compression levels for the transmission across the wireless communications. Using low compression uses more bandwidth while high compression saves bandwidth but at the cost of CPU capacity and battery energy on the mobile device. The arbitration between the network, CPU and battery usage is subjective and sensible balancing can only derive from the context by the user or a user agent. To revisit the sample scenario of section 5.1.5 for a battery resource where users may select a low equivalent cost (*e.g.* I'm on the train to work where the battery will soon be recharged) or a high equivalent cost (*e.g.* I'm lost in the Amazon and I'd really like to keep my navigation software running as long as possible) as appropriate to their particular situations or have an agent that is familiar with their work and travel patterns to perform such policy making independently.

The second, and perhaps more important, role of the adaptation policy is to derive an adaptation operating point from the application cost-benefit view presented to it. Possible policies that may be easily implemented include:

Minimum Performance — set for operation at a specified performance level regardless of cost.

Cost Capped — set for operation at the highest performance possible for a given cost.

By Ratio — set for operation at a point that provides the best cost/performance ratio.

Budget Based — amortise a given cost budget over a period of time while maximising performance over the duration of the budget period.

Combinations — some mix of the above. *e.g.* provide best value within a cost ceiling or above a lower performance bound.

5.3 Operational Overview

Superficially, applications operating within the DProj framework appear as a number of processes executing on several hosts. Figure 5.6 gives an example distribution of the processes required to run a mobile application that operates across a mobile client connected by a wireless link to a fixed network with three servers.

The framework establishes, on each host, basic runtime support with a collection of application code called a *code pool*. A single application monitor is established to control the application's configuration and hence adaptation. The runtime support comprises a *resource monitor* and a *code pool loader*. There is a single resource monitor per host which provides monitoring on availability of host-specific resources such as CPU or battery energy. This data is reported to the application monitor upon request. Figure 5.6 shows the application monitor based on the mobile host — this allows adaptation to continue

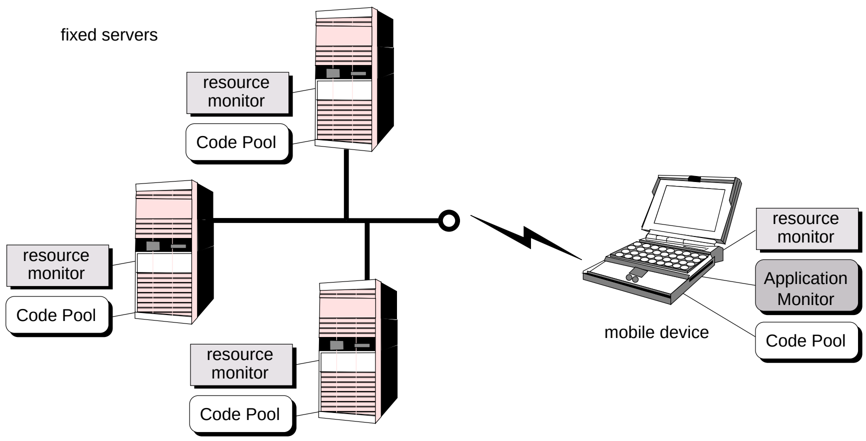


Figure 5.6: Framework component processes in the system.

to function usefully for the mobile user even in a disconnected environment. A monitor implementation that allows for itself to be distributed may have a distribution which more closely matches that of the application and hence more efficient operation in both connected and disconnected states.

The code pool loader is used to instantiate code pools for a specified application on a host. Each code pool contains implementation code for the elements of a single application. The code pool and code pool loader concept provides a primitive form of remote object instantiation and invocation. A code pool is instantiated by a loader which imports the application's Container description for an application which in turn will import the required sub-element code and so on in a hierarchical manner. Application code pools contain both the code for and a structural description of the application. A single code pool need not contain the full functionality of an application but will generally contain a large portion of it. The code pool contains a full hierarchy of the application elements from the application's top-level Container although it may be only partially populated with some functional elements being held as *nulls*. Each code pool holds the necessary element interconnection information as part of its hierarchical structure which may be used at to control the dataflows between the component elements (see section 5.4.1 below).

Hence, a code pool does not necessarily contain the full functionality of the application. Also, one code pool may contain duplicate functionality in the form of alternative implementations of the same functional element. A running application may use the element implementations from one or more of these code pools. The application may add or drop elements from various pools during its operation. Similarly, code pools may be added or dropped from consideration during runtime as their availability or accessibility varies. The application has no *home* pool and its running set at any point may be disjoint from its initial set.

While the code pools provide the application functionality, its configuration is controlled independently by the application monitor. The application monitor is responsible for the start up and continuing reconfiguration of the application. An application is non-functional until an application monitor takes control of it through a model and performs its initial configuration and start-up. Launch and operation of an application proceeds as follows:

0. user requests launch of an application,
1. the launch request is relayed to the system runtime of all relevant client and server machines,
2. host runtime fetches appropriate code pool and launches it, registering it with the CORBA Naming Service. The exact contents of each code pool will be dependent on the class of machine it is hosted on,
3. an application monitor is launched which then connects to the code pools and constructs a model for each pool,
4. the application monitor merges these models to produce an overall application model,
5. the application monitor uses the model to wire the default application configuration and commence its operation.
6. the application is iteratively adapted. The application monitor uses the application model to:
   (a) monitor resource availability, resource usage and track application element performance,
   (b) construct a cost-benefit function summarising the potential operational configurations of the application in terms of the perceived performance and the cost of each configuration,
   (c) use some adaptation policy to determine the optimal configuration from the given cost-benefit function,
   (d) propagate this reconfiguration to the application elements,
   (e) update resource availability (update, delete or re-balance),
   (f) update the application model (add, delete, or re-balance elements, code pools or hosts as reported by some service discovery unit).

5.4 Framework Components

Following the design overview of section 5.2, this section describes the components that make up the DProj adaptation framework. These include the elemental approach to application structuring, the application monitor which provides the adaptation service to these applications and also the centralised resource management structure. Finally, it details the mechanism behind the operation of the adaptation service provided by the application monitor; this occurs in two phases: summarisation of the application through cost-benefit aggregation and then post-decision reconfiguration of the application.

5.4.1 Application Elements

The DPROJ application decomposition stems from the structuring of the adaptive distributed control platform developed in chapter 4 in which the basic building block of a mobile application is the *functional element*. An element represents a unit of functionality of indeterminate scale and is defined as being some executable entity that: optionally takes some input (an *import*), performs some calculations, and optionally produces some output (*export*). An element may be characterised by its state of activation, resource requirements, cost-performance function, logical location, and location constraints.

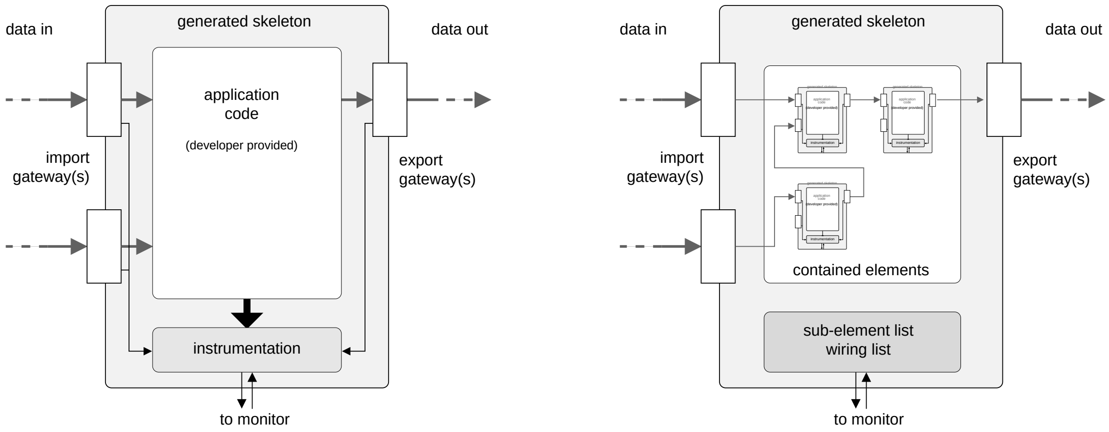
DPROJ extends the element concept to use an element abstraction known as a *virtual element*. A virtual element retains all the properties and characteristics of an element but provides an abstract superclass interface to three distinct fundamental element types: the *Basic* element, *Set* element and *Container* element. Set and Container element types may be termed *collective elements*, and provide hierarchical structuring capabilities whilst retaining the element-based application paradigm. The hierarchical structure allows for fine grained elemental decomposition of an application whilst also allowing one-for-many type substitutions and maintaining scalability, reusability and maintainability. Set and Container element types are purely structural elements while Basic elements provide the fundamental functional unit.

The following notation is used in the descriptions of element composition and operations in this chapter:

generic element	e	activation of an element	$A(e)$
particular element i	e_i	deactivation of an element	$A'(e)$
Basic element	e^b	resource use of an element	R_e
Set element	e^s	operational cost of an element	C_e
Container element	e^c		
collection of elements	$\{e_a, \ldots, e_n\}$		
sub-element j of e_i	e_{ij}		

Basic Element – e^b

The Basic element is the fundamental functional unit of application construction and the only element type to contain actual functional code contributing towards the intended functionality of the application. Hence it is also the only element to handle data: the collective elements never touch the data, not even to relay it to their sub-elements — the data is always routed directly. Figure 5.7(a) shows a conceptual Basic element with imports, exports, developer provided implementation code and instrumentation interfacing to the application monitor. The points of import and export may be generically termed *gateways*.



(a) Basic element (b) Container element

Figure 5.7: Functional element types.

The characteristics of a Basic element derive from the implementation of the functionality it possesses. Its costs are derived by estimation or monitoring of the element's resource usage (with the addition of appropriate instrumentation if necessary). Its performance estimate may be specified as fixed at a specific value by the developer or dynamically provided by the element's implementation code. The possible instance locations of the element are also application specific. The element developer may specify which hosts this element should be allowed to execute on (it may be obvious from its purpose or may require special resources) but this may be overridden by any collective element which contains it. This allows a single implementation of a Basic element to be used in different applications or different parts of the same application with different instance location criteria without modification to the actual element specification.

***Set Element* – e^s**

A Set element is a collective of generic elements representing alternatives to fulfil a common function. Hence a Set element with n sub-elements is defined as:

$$e_i^s \equiv \{e_{i0}, e_{i1}, \ldots, e_{in}\}$$

while the activation of a Set element implies the activation of exactly one of its sub-elements:

$$A(e_i^s) \Rightarrow \begin{cases} A(e_{ij}) & j \in (0, n) \\ A'(e_{ik}) & \forall\, k \in (0, n),\ k \neq j \end{cases}$$

A Set element is used to collect together alternative elements that may be used to implement the functionality of a single functional element. A Set is not explicitly specified by the application developer but is implicit in the application specification where there exists as sub-elements of one Container more than one element of the same name and functionality. These sub-elements may differ in implementation, location or structure (sub-elements may themselves be any of Basic, Set or Container elements). It is the presence of such sets of alternative elements that provides for the basic reconfiguration options available for adaptation.

The characteristics of a Set element derive from those of its sub-elements. Its costs and benefits are the set of minimum costs, maximum performances of its sub-elements. Its set of possible locations is the union of the location sets of its sub-elements although this has little meaning for a collective element which has no functional implementation of its own.

Container Element

A Container element is a collective of generic element representing elements that operate concurrently and are logically grouped together to form a higher level functional unit. Hence, a Container element with n sub-elements is defined as:

$$e_i^c \equiv \{e_{i0}, e_{i1}, \ldots, e_{in}\}$$

while the activation of a Container element implies the activation of all of its sub-elements:

$$A(e_i^c) \Rightarrow A(e_{i0}), A(e_{i1}), \ldots, A(e_{in})$$

A Container element is used to collect together separate functional elements that are linked together to form a greater functional unit. Hence, a Container element comprises both a set of sub-elements and a *wiring* list of the data connections between them. Figure 5.7(b) shows a conceptual Container element with developer defined sub-elements. The Container can provide a list of contained elements and a wiring list to the application monitor. The wiring list comprises both the interconnections between sub-elements but also wires reflecting how the Container's inputs and outputs map to particular inputs and outputs of its sub-elements. The sub-elements run concurrently with implementations and locations unconstrained by other sub-elements of the Container and may themselves be any of Basic, Set or Container elements.

The characteristics of a Container element derive from those of its sub-elements. Its costs are the sum of its sub-element costs, while its performance is a linear combination of its sub-elements' performances. Its set of possible locations is the intersection of the location sets of its sub-elements, however the sub-elements comprising a collective element need not be co-located, this location is notional only.

5.4.2 Application Monitor

The running and adaptation of an application is controlled by an independent process called the *application monitor*. While the scale invariant nature of the application element composition means that a monitor may be applied to operate at any level within the element hierarchy, typically one application monitor process is dedicated to each running application while another monitor presides over this collection of monitors to dictate overall system adaptation.

At the heart of the application monitor lies the *application model*, performing both application monitoring and application reconfiguration tasks. It exists as a separate entity from the application but is intimately associated with an application through *hooks* into each of the elements comprising the application. These hooks allow the application model to query information from the application code pool about its structural specification, resource requirements, resource usage and performance expectations. It tracks the dynamic changes to the application both through reconfiguration and also through the dynamic addition or dropping of components as hosts or services come into or out of availability.

At startup time, the monitor performs initial default configuration of the application and thereafter performs reconfigurations at regular intervals according to the adaptation strategy at hand. In the case of the loss of the monitor process after startup, the application will continue to run but will not be able to adapt further; if the loss is recognised, another monitor process may connect to the code pools of the running application, build a new model and assume the adaptation role.

Model Construction

The model is held by the application monitor as a logical representation of the structure of the application and configuration information. As such it models the full hierarchical structure of the application including elements on all participating hosts by building a corresponding tree of element proxies and keeping records of the required data connections between elements. Each of these proxies corresponds to some logical virtual element in the application proper or some collection of the same. The records of the required data connections for each Container are termed its *wiring set* where each single connection is considered to be a *wire*.

A model is constructed from the application specification held in an application code pool. A minimum of one code pool is required to run an application although usually there will be at least one pool for each participating host. For each code pool, a model builder constructs a corresponding hierarchical model of the application as defined by that pool. As new code pools are discovered, their models are merged into the existing model to produce an overall view of the application. As the models are merged, their associated wiring sets are optimised to allow direct connection between Basic elements, irrespective of their relative levels in the hierarchy.

Model Merging and Expulsion

The model manipulates *element proxies* corresponding to the application elements. To combine the model trees of code pools from different hosts or to dynamically add discovered services at runtime, the model trees must be *merged* or have elements or element trees *expelled*.

Merging of trees may be performed by descending the hierarchy using these rules for merging any two elements e_i and e_j which may be of different element types:

$e_i^b + e_j^b \to e^s\{e_i^b, e_j^b\}$

Basic + Basic gives a Set containing e_i^b and e_j^b as sub-elements.

$e_i^b + e_j^s \to e^s\{e_i^b, e_{j0}, e_{j1}, \ldots, e_{jn}\}$

Basic + Set gives a Set containing e_i^b along with all of the original sub-elements of e_j^s.

$e_i^b + e_j^c \to e^s\{e_i^b, e_j^c\}$

Basic + Container gives a Set containing both e_i^b and e_j^c as sub-elements.

$e_i^s + e_j^s \to e^s\{e_{i0}, e_{i1}, \ldots, e_{in}, e_{j0}, e_{j1}, \ldots, e_{jn}\}$

Set + Set gives a Set containing all of the original sub-elements of both e_i^s and e_j^s.

$e_i^s + e_j^c \to e^s\{e_{i0}, e_{i1}, \ldots, e_{in}, e_j^c\}$

Set + Container gives a Set containing e_j^c along with all of the original sub-elements of e_i^s.

$e_i^c + e_j^c \to e^c\{e_{i0} + e_{j0}, e_{i1} + e_{j1}, \ldots, e_{in} + e_{jn}\}$

Container + Container where there is a one to one correspondence by name and type between the sub-elements of e_i^s and e_j^s gives a Container where each sub-element e_k is the result from merging the pair of corresponding sub-elements e_{ik} and e_{jk}, or

$e_i^c + e_j^c \to e^s\{e_i^c, e_j^c\}$

Container + Container where the two Containers have disparate sub-element composition gives a set containing e_i^s and e_j^s as sub-elements.

Note that, following these rules, the only merge condition that requires recursive descent is the merging of two Containers with corresponding sub-elements. All other merges are effectively merging of alternative implementations (or instances of implementations) and result in a Set. Merging corresponding containers is effectively amalgamation of two instances of the same implementation such as when the same implementation is available on more than one host. Merging the Containers to form a single new Container allows a single implementation of the Container to be distributed over multiple hosts. Applications are a notable case of this as, at the top level, applications are logically identical to Containers with no external imports or exports.

Expulsion of elements or entire code pools from the application requires the expulsion of their corresponding proxies from the model. Expulsion from the model and subsequent reconfiguration of the application is sufficient to halt application use of those elements.

Legal expulsion is limited to element proxies that are the direct sub-element of a Set element containing at least one other sub-element. This is required to assure that there continues to exist an alternative instance to provide the element functionality. If this is not present and the expulsion is forced (*e.g.* due to forced disconnection from the network), then the application will become inoperable and will be issued a `pause` directive. If an application developer wishes an application to continue operation even without the presence of some component, a null element with zero performance should be specified as an alternative implementation and incorporated with the home code pool (usually the code pool on the mobile client).

Wiring Set Optimisation

A Container's *wiring set* defines the data dependencies between the immediate sub-elements of that Container. An individual wire specifies a data source and a data sink as endpoints and these must be of the same data type. A data source or sink corresponds to an input or output *gateway* of a virtual element. During adaptation, as individual elements are activated and deactivated, their data connections too must be connected or redirected appropriately. However as the data source and sink belong to sub-elements which may themselves be Container or Set collective elements, the true data endpoint may be several levels deeper or higher in the element tree, passing through many Container or Set gateways in the process.

In a naïve implementation, a collective element might have actual gateways that forward the received data to its real destination. However, this has many drawbacks:

1. this introduces many levels of redirection in a great number of data paths which is clearly inefficient,
2. redirection may cause the data-path to cross and re-cross communications boundaries unnecessarily as the real source and ultimate destination may be colocated yet the forwarding gateways are on remote hosts,
3. there is no obvious method to resolve the location of the forwarding gateways in the case where two containers are merged to produce a single container, and
4. this would require the dynamic generation of forwarding gateways for Set elements which have no counterpart in the code pools and are logically formulated in the model during merges.

A more advanced method of providing trans-hierarchy wiring in an efficient manner is therefore required. Moving the data forwarding into the model allows it to perform forwarding analysis internally and then issue detailed data wiring orders directly to the Basic elements of the application.

The model builds wiring records with the proxies as it constructs the model tree. For each gateway of each element it will build a *wire endpoint* to represent it. Wire endpoints are grouped together into *wire endpoint sets* which represent the potential gateways for one end of an individual wire. A wire endpoint can then determine which wire endpoint it should correspond with by querying the wire endpoint set at the other end of its wire. This will return a reference to the currently active wire endpoint. Forwarding gateways are represented in this model by a special subclass of wire endpoint called a *duplicitous wire endpoint*. Duplicitous wire endpoints come in pairs, one represents the endpoint of the wire coming into the gateway while its mate represents the endpoint of the wire from the gateway to that of a contained sub-element. When a query is made on a duplicitous wire endpoint, it forwards that query to the correspondent of its mate.

Once an endpoint has determined its correspondent endpoint set, it may store this reference to avoid retracing the path through the forwarding gateway. The duplicitous wire endpoint can then be removed from the model. New merges into the model tree will automatically add endpoints into the endpoint sets in the case where they interact with existing data flows or, in the case of entirely new model sub-trees, will create new duplicitous wire endpoints as needed.

The originating endpoint can therefore find its ultimate destination with no knowledge of the number or nature of any forwarding gateways in between. These will necessarily be Basic elements as these are the only real originators and consumers of data. Once the corresponding pair has been determined, the actual data connection can be prepared between the two gateways.

Figure 5.8 demonstrates a simple case of wiring set optimisation. In this example, redundant data link wires are removed to minimise data transfers.

5.4.3 Resource Management

Resource management works in parallel with the application model, providing it with up to date information on resource availability and costing. The resource manager is composed of interfaces to allow resource policy manipulation and queries of resource costing along with a set of resource-specific *resource handlers*.

The resource manager provides a generic resource structure in which multiple resources can be monitored and queried. It provides resource costing for all applications being controlled by the monitor and does not distinguish between which application is enquiring. This very generic structure allows greater flexibility in resource management.

A query asks for a cost estimate for a given quantity of a specified resource. The query may also be parameterised by other criteria; for example, a query on the cost of N kilobytes per second of bandwidth may be parameterised by the host pair. There is no explicit notion of universal and item-specific resources but this parametric approach effectively allows the same functionality.

The resource manager implements the resource equivalency functions of the adaptation policy provider by adjusting the cost estimates of the individual resource handlers before returning the value to the caller.

Resource Handlers

A resource handler is specific to a single resource such as CPU, power, or bandwidth between a pair of hosts. For example, the processor resource handler monitors the usage

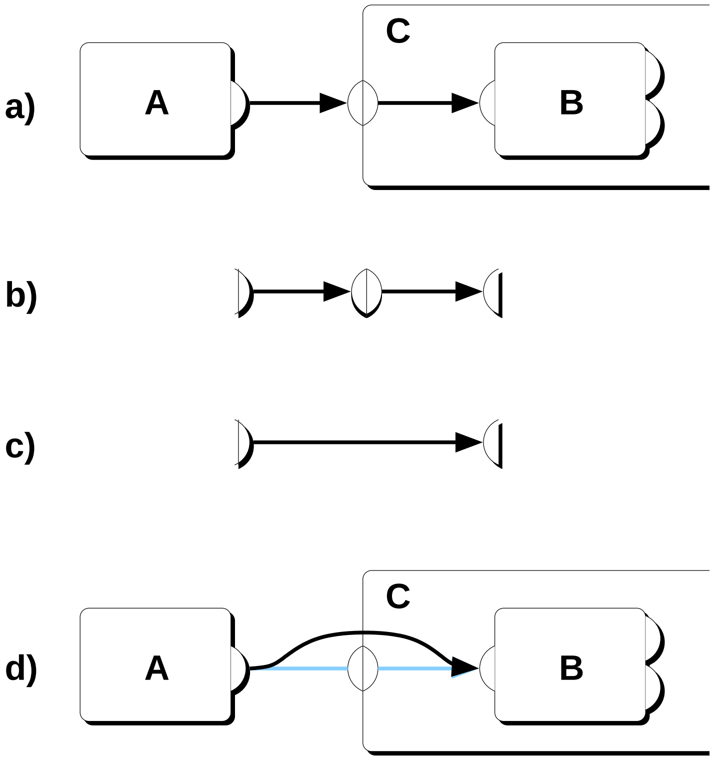


Figure 5.8: Wiring set optimisation by elimination of duplicitous endpoints. The application structure in (a) includes a data dependency from element A to element C which is a Container containing element B, the ultimate destination of the data link — the complete data link appears as two logical links. When the application monitor builds the model, the two logical links appear joined by a pair of duplicitous wire endpoints, (b). Wiring set optimisation results in (c) in which the duplicitous endpoints are removed resulting in a single data link. In (d), the monitor sets up actual data connections in the application; the single link in the model translates to a direct link in the application, hence removing a redundant stage in data transferal.

of CPU time and sets a cost function for use of that processing. Application elements specify their processor resource requirements to the handler via the resource manager which returns the expected cost.

In this manner, the exact monitoring, allocation and costing of each resource is held independently and targeted specifically towards that resource. This allows a large range of resource management styles and policies, *e.g.* for a CPU resource, while the current handler for processing demands currently just prices CPU usage according to the ratio of the requested capacity to that available, in the future if the framework is deployed on a real-time operating system, the handler may perform the exact scheduling of the process and cost the request by priority and schedulability.

In accordance with the principles of real penalties, fair allocation and closed loop control, a supply-demand type pricing scheme is necessary to control over subscription of resources. In such a scheme, the cost of a resource increases with increasing demand or decreasing supply, creating a negative feedback regulation of resource usage.

This inflation of costs may continue until resource demand is driven to a value within that available. In some cases, where a resource is exhausted or is unavailable, its cost will escalate to an effectively infinite amount. This would be the case, for example, when a CPU is fully loaded or a network becomes unreachable. Any application element that relies on that resource with no lower cost alternative will then also assume a high (infinite) operational cost and will likely be suspended or culled by the adaptation policy.

Resource Costing

Care must be taken to implement sensible cost controls for resources such as network access in order to maximise use of the resource if it must be taken in set units. For example, if using a packet based communications medium such as GPRS it may be appropriate to price the communications traffic on a per-byte basis to reflect actual communications cost. However, in the case of connection based links such as GSM modem, pricing must reflect the fact that a fixed capacity is available and paid for even if only a small portion of it is used.

Hence, the calculated cost per byte should be the actual cost of the connection divided over the used capacity — with this scheme, the cost per byte will decrease as more of the capacity is used (while the total cost remains constant). This serves to encourage configurations to use the spare capacity. However, as the connection traffic approaches its capacity, the resource handler may instigate a *jump* to a (more expensive) higher rate network and hence the cost per byte will jump too (until more of that spare capacity is used too). Note that a network here is taken to be a notional communications setup and hence a jump to a new network may involve the bonding of more than one channels in parallel.

Figure 5.9 shows the expected cost profile for an example communications scenario where the mobile user has access to up to four 10kbps channels that may be used in parallel where each channel has the same cost. The profile shows relative cost which is the ratio of the effective price per byte reported by the handler compared to the nominal cost per byte if the channel capacity were fully utilised. In this scenario, the cost approaches the optimum when the active channels are fully utilised. Opening a new

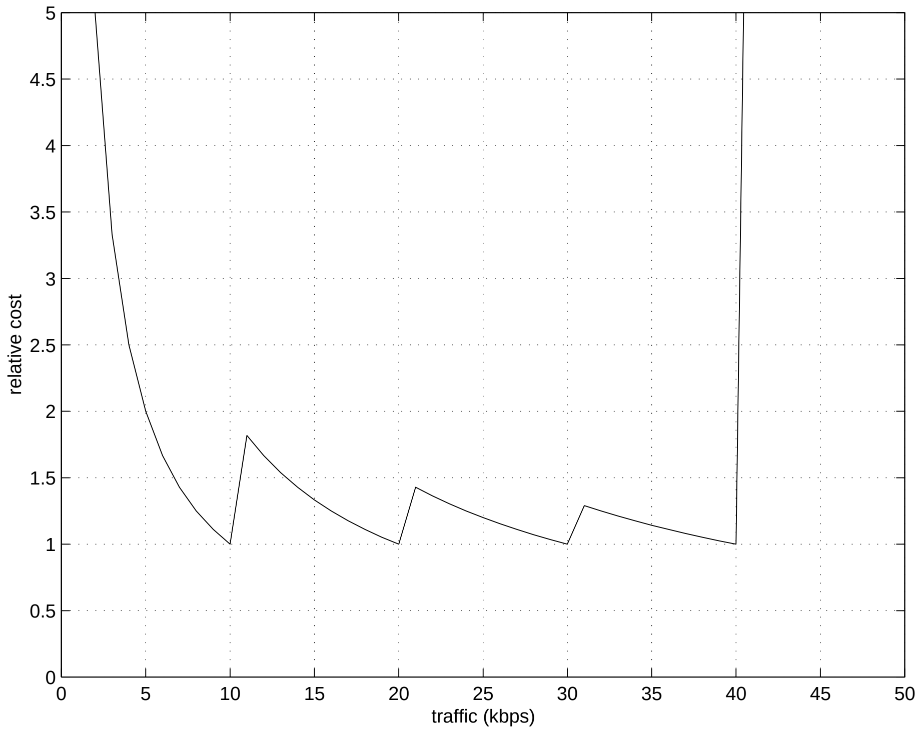


Figure 5.9: Relative cost (effective/nominal) per traffic over one to four bonded 10kbps channels showing the increases in effective cost as new channels are added.

channel causes the effective price per byte to increase.

If communications traffic exceeds the upper limit of all channels available, then a notional further channel of high (infinite) cost is considered to exist to artificially inflate the access price until demand is reduced below the available capacity. It may be instructive for a handler to shift the price structure to the new network scheme before actual transition to ensure that expectations drop accordingly. Similarly, dependent on the exact nature of the resource, resource handlers may wish to implement some hysteresis function or low pass filtering of requests to avoid thrashing between resource modes.

It is also possible for resource handlers to access handlers for other resources should they be dependent on them. For example, use of a CPU resource also implies use of battery energy as does use of a communications resource. In this way, if a battery resource becomes expensive then so too will CPU and this in turn should reduce demand.

Requirements Determination

An element's resource requirements derive from its stated needs. A Basic element states its needs explicitly. Collective elements derive their needs from the needs of their sub-elements; this occurs implicitly through the process of cost-benefit aggregation.

Basic elements are polled for their resource requirements. An element responds with its exact requirements of each resource and the relevant parameters. In addition, requirements for communications and CPU may be obtained through instrumentation of the element and extrapolating future requirements from its recorded behaviour. An element may indicate also additional usage of these inferred requirements. Both active and inactive elements are polled for their requirements when building a cost-benefit function.

Effectively, each Basic element returns a simple cost-benefit function in which it describes its possible operating performance points and the associated resource requirements at each point. The resource usage of an element, e, at performance point, p, is noted as $R_{e^b}(p)$. The resource usage can be directly translated to a unified cost metric using the above-described equivalence functions.

5.4.4 Cost-Benefit Function

A *cost-benefit function* gives the corresponding cost versus perceived performance for a given element. When presented with the cost-benefit function of a given application, an adaptation policy decider can select an operating point within the function with a given resource cost and performance.

The cost-benefit function is derived from the application model and is specific to a given virtual element proxy. The model allows cost-benefit functions to be calculated for each Basic Element whilst those for collective elements are derived through combination of the functions of its sub-elements.

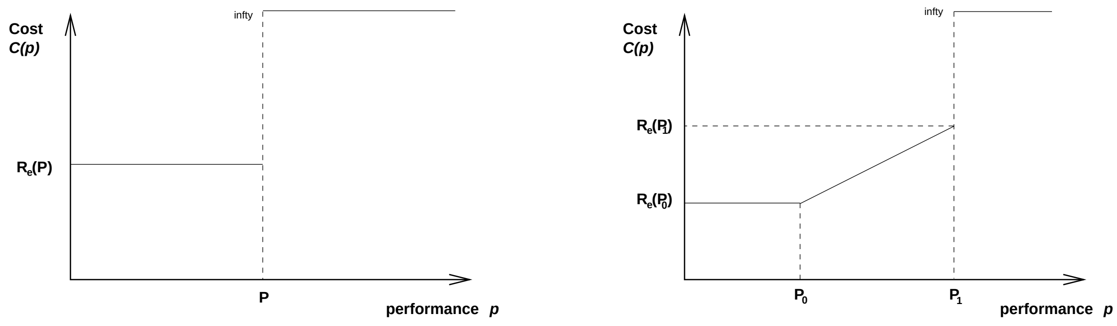
A cost-benefit function has the following properties:

- cost is measured in terms of the unified resource metric and has an unconstrained value (negative values are allowed),
- performance is measured in terms of the unified performance metric and is indicated on a fixed (but arbitrary) integer scale of $0 \rightarrow 1000$,
- the cost function is *monotonically increasing* with performance,
- the cost function is *piecewise linear*.

The first two of these points follow the metric unification to use unified resource and performance metrics. These metrics are arbitrary but provide a uniform, simplified view of an element's profile reduced to single dimensionally quantities on each axis.

The two latter points are of particular interest. A monotonically increasing function is one which never decreases, *i.e.* is always increasing or at the same value. Cost-benefit functions are defined to fit this as having an element which can operate at better performance at a lower cost than it can perform poorly is not a sensible proposition. Such an element should perform at the lower cost, higher performance point if requested to operate at the lower point. The formation of cost-benefit functions assumes that this is the case and the cost of any point of performance dictates an upper bound for the cost of any lower performance point.

Additionally, a piecewise linear function allows true representation of discrete cost levels as a discontinuous function but continues to allow non-linear overall functionality. It is easier to store and manipulate than a true curve which is important in the



(a) singular operating point, P. (b) linear operating range, P_0 to P_1.

Figure 5.10: Basic element cost-benefit profiles.

procedure of function aggregation, described below, where the *detail* (number of pieces) of a function becomes greater with the level of aggregation.

Basic Elements

The cost-benefit function of a Basic element is derived from analysis of its resource usage and its indicated performance capabilities. It is expected that any one Basic element implementation or instance be able to operate at only one performance point, P, and hence at a fixed cost equivalent to the resource usage at P: $R_{e^b}(P)$. Such an element would have cost-benefit function of:

$$C_{e^b}(p) = \begin{cases} R_{e^b}(P) & 0 \leq p \leq P \\ \infty & P < p \end{cases}$$

where the monotonically increasing property extends the point of $(P, R_{e^b}(P))$ to be $R_{e^b}.rect(0, P)$. Operating points above its capability are caused to be undefined by setting the cost to infinity (figure 5.10(a)).

It is possible for a Basic element to have a non-fixed operating point as might be the case for an element implemented using some parameterised algorithm. In this case the element may report a set of possible performance ranges and corresponding resource requirements. Figure 5.10(b) shows a cost-benefit function which corresponds to an element which can perform at a range of performance points from P_0 to P_1 with a linear cost-benefit characteristic. Once again, the monotonically increasing property extends the lower performance point as a constant piece back to zero while performance points above the upper bound of the range are caused to be undefined by setting the cost to infinity.

Hierarchical Aggregation

The cost-benefit functions of collective elements (Sets and Containers) derive directly from the profiles of their sub-elements. Hence, these are obtained by *aggregation* of the cost-benefit functions of their sub-elements. Cost and performance are measured using a common metric through-out the element hierarchy and so these must be preserved in the aggregation process: costs must represent actual costs — artificial inflation should not persist through aggregation; additionally, the method of aggregation chosen will define the concept of aggregated performance.

As Sets represent a collective of alternative elements, the Set aggregated function can be thought of as a minimum logical *OR* ($\oplus$) of the sub-elements' individual cost-benefit functions. As Containers represent a collective of elements that run together, the Container aggregated function can be thought of as a logical *AND* ($\odot$) of the sub-elements' functions.

Sets perform aggregation through minimum cost aggregation of their sub-elements' cost-benefit functions. Hence:

$$\begin{aligned} C_{e^s}(p) &= C_{e_0}(p) \oplus C_{e_1}(p) \oplus \ldots \oplus C_{e_n}(p) \\ &= \text{Min}\,\{C_{e_0}(p), C_{e_1}(p), \ldots, C_{e_n}(p)\} \end{aligned}$$

This indicates the selection of the minimum cost sub-element for each performance point. An example combination is shown as figure 5.11. The Set aggregator is commutative and associative. The monotonic increasing property is preserved through Set aggregation.

Containers require more complex aggregation algorithms as the Container cost-benefit function must represent the total cost for the total perceived performance of all sub-elements operating concurrently. Obviously, this depends greatly on how the performance indicators of the sub-elements are combined to determine the overall performance of the Container. Any performance combinations must preserve the relative costs and benefits of sub-elements while being sure that overall costs or benefits are not inflated or deflated through successive layers of aggregation. This ensures two important capabilities: firstly, the balance of resource allocation and performance attribution between sub-elements of a Container may be dynamically variable under external control to allow a user to prioritise some applications ahead of others — any aggregation algorithm must fairly implement this prioritisation; secondly, as cost-benefit functions form the basis of adaptation decisions, consistency of these functions at all levels allows such decisions to be made at any level (scale invariance).

Two schemes are described here. They are both equally valid but derive from different notions of collective performance. This notion and hence the aggregation process will also affect how a developer will decompose an application into elements and how performance metrics are determined. The two schemes lead to different aggregation algorithms with differing computational complexity:

1. ***Independent performance:*** in which the performance cost of a Container element is a function that represents the minimum cost of the combined functions such that the performance of the whole Container equals the sum of the performances of the individual sub-elements. This is the case when the sub-elements of the Container can add or detract from the overall Container performance independently of the operation of other sub-elements. *i.e.*

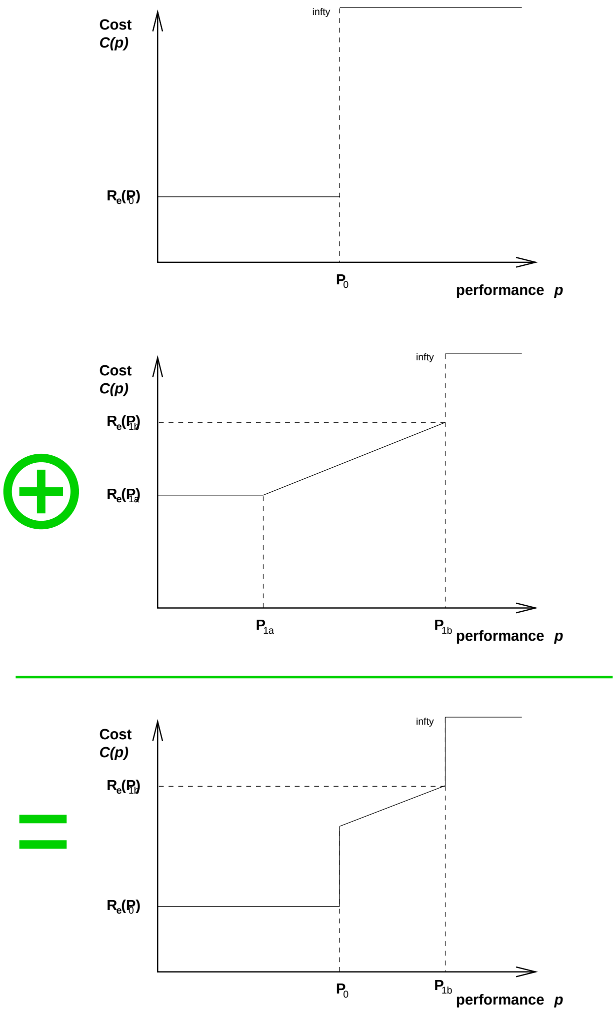


Figure 5.11: Set element cost-benefit aggregation.

$$C_{e^c}(p) = C_{e_0}(p_0) \odot C_{e_1}(p_1) \odot \ldots \odot C_{e_n}(p_n) \text{ where } p = \sum_{i=0}^{n} p_i$$

This naturally lends itself to a very computationally expensive algorithm as it involves minimisation of an arbitrary number of non-linear functions in an equal number of variables. A combinator, M, may take the following approaches to reduce the complexity:

- ensure a distributive property such that:

  $$M(C_0, C_1, \ldots, C_n) = M(C_0, M(C_1, M(\ldots M(C_{n-1}, C_n) \ldots)))$$

  If the complexity of performing M for n functions is $O(M_n)$ then this reduces the complexity from $O(M_n)$ to $n \times O(M_2)$ which gives a true reduction in the case where $O(M_n)$ is greater than polynomial of order 1 (linear). This can be expected to be true for minimisation of non-linear functions.

- minimise the number of performance points p at which the function must be calculated by operating on linear pieces rather than individual points. Combined in pairs according to the above point, this can further reduce the overall complexity by reducing $O(M_n)$.

One algorithm fitting this notion of accumulative performance is *minimum sum convolution* (MinSumConv). MinSumConv achieves aggregation through iteration over sub-element cost-benefit function pairs. For each pair, it determines the minimum cost at each performance point:

$$\begin{aligned} M(C_0, C_1) \to C(p) &= \text{Min}(C_0(p_0) + C_1(p_1)) \\ &\quad \text{where } p_0 + p_1 = p \\ ie: \quad C(p) &= \text{Min}(C_0(p_0) + C_1(p - p_0)) \end{aligned}$$

With reference to figure 5.12, we can imagine this as a convolution type approach in which C_1 is flipped horizontally to produce C_1' and gradually shifted past C_0: for an overlap of p, $p_0 + p_1' = p$ for all points in the overlap. Hence $C(p)$ is the minimum of $C_0(p_0) + C_1'(p_0)$ in the overlap range $0 < p_0 < p$.

If C_0 and C_1 are both piecewise constant functions, then the above can be evaluated equivalently by considering only the values of p which are also piece boundary points of C_0 or C_1 or for which any boundary point of $C_1(p - p_0)$ coincides with a boundary point of C_0. For each evaluation point, p_e:

(a) create $C_1'(p_0) = C_1(p_e - p_0)$ by flipping C_1 about pivot $p_e/2$,

(b) add $C_0(p) + C_1'(p)$ for the range $0 \leq p \leq p_e$,

(c) $C(p_e) = \text{Min}(C_0(p) + C_1(p))$ for p where $0 \leq p \leq p_e$

(d) this value is extended for the range from p_e to the next evaluation point, p_{e+1}: $C(p) = C(p_e)$ for $p_e \leq p \leq p_{e+1}$.

Hence evaluation of $M(C_0, C_1)$ is reduced to $O(b_0 + b_1 + b_0.b_1) = O(b^2)$ where b_0 and b_1 are the number of constant pieces comprising C_0 and C_1 respectively.

However, MinSumConv has the failing that for aggregation of functions that include non-constant pieces, the above selection of evaluation points is not sufficient and full calculations across the range of p must be made. Current implementations perform this for every integer value of p in the range. This may be further approximated by using a larger sampling granularity. Additionally, the function produced from each MinSumConv is likely to have $O(b^2)$ pieces and hence for aggregation of cost-benefit functions for a Container with n sub-elements using the above described distributive approach will result in a function with $O(b^n)$ pieces. Any application hierarchy with more than several levels of composition will result

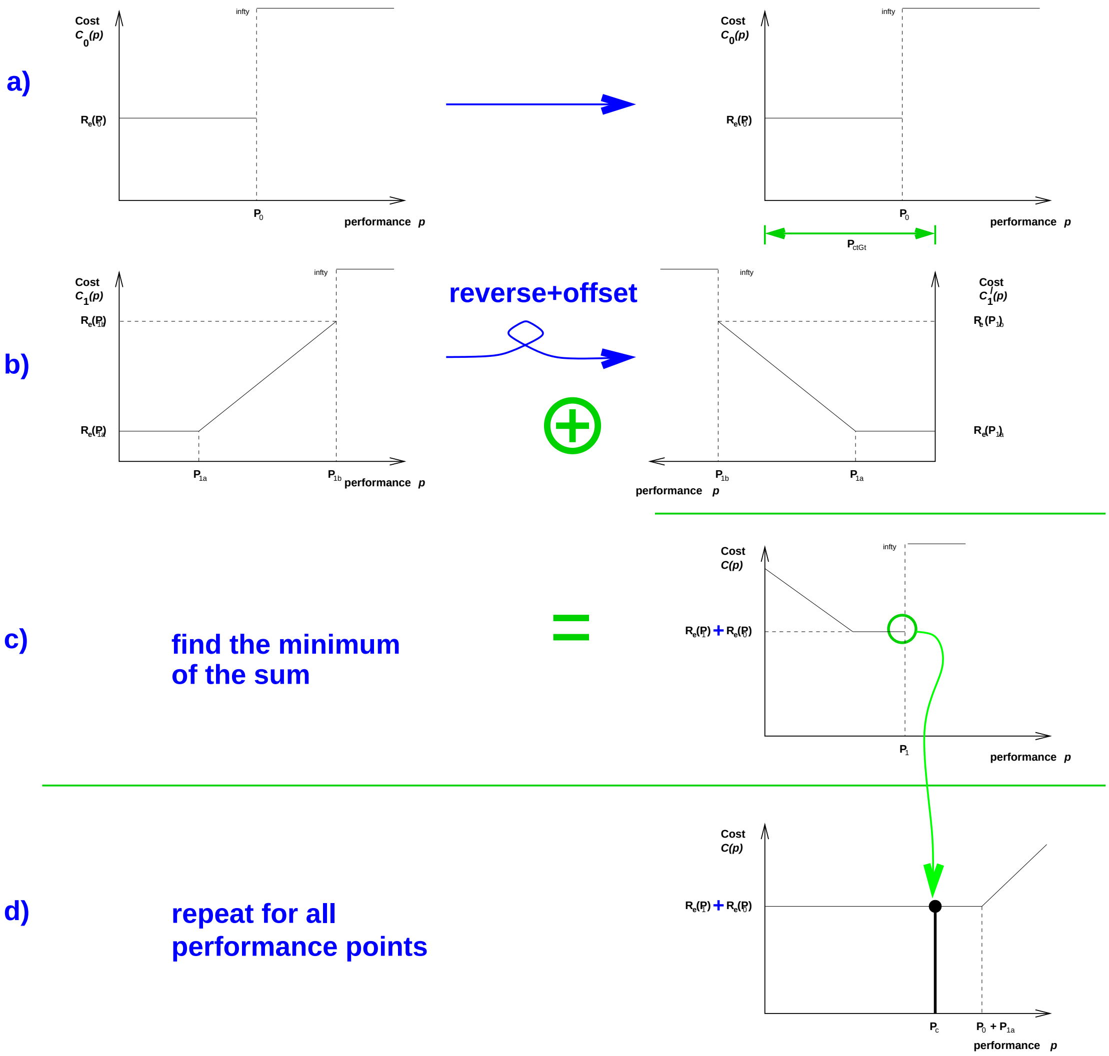


Figure 5.12: Container element cost-benefit aggregation of cost-benefit functions of two sub-elements, C_0 and C_1 by Minimum Sum Convolution. For evaluation of overall cost at overall performance point P_c: a) the first function C_0 remains as is; b) the second function, C_1 is flipped and offset by P_c to produce C_1'; c) these are summed giving the cost profile for various distributions of P_c over C_0 and C_1 — at all points the total performance is still P_c; d) the minimum cost point from this sum is the effective cost of operating the two sub-elements at P_c and is added as a point on the Container cost-benefit function. These steps are repeated to evaluate the cost at each point in the performance range. Some optimisations may reduce the number of points at which evaluation is required.

in a cost-benefit function which has a large number of pieces which may be unwieldy for analysis and calculation. Hence, this specific definition of performance is not suitable for applications or systems comprising more than a few levels or elements without further simplification of the aggregation algorithm.

2. ***Cooperative performance:*** provides a possible alternative definition of performance for a Container in which the performance of a Container is considered to be the minimum of the performances of its sub-elements. This is the case when all sub-elements contribute to the functionality of the Container as a whole in equal parts in a manner such that degradation of performance of one part degrades the overall performance similarly. *i.e.*

$$C_{e^c}(p) = \sum_{i=1}^{n} C_{e_i}(p)$$

This requires only a simple accumulation of the sub-element functions — an $O(n)$ operation.

The selection of one aggregator over another depends on the developer's attitude towards performance definition in application decomposition. However, as it is hoped that the one adaptation system will operate for all applications, only a single scheme should be used. While the MinSumConv scheme is perhaps more intuitive and flexible and the DPROJ application monitor implements both schemes, work in this thesis follows the second scheme which is conceptually and computationally much simpler.

5.4.5 Reconfiguration

The monitor profiles an application's operation and configurations space, resulting in a summary in the form of a cost-benefit function such as that shown in figure 5.13. As described above, an adaptation policy takes this function and uses it to deliver a configuration response in the form of a *performance set-point*. This is a single point along the function which can be achieved by some configuration of the application under the current resource conditions. The monitor must take this point and translate it into a particular application configuration in the reverse process of building the cost-benefit function.

To achieve this, the performance set-point must be propagated back down the application element model hierarchy with appropriate transformations to reflect how the cost-benefit function was originally derived. This in turn forces some activation and deactivation of elements, reconfiguring the application.

A Basic element proxy receives the performance set-point and passes it directly to the element it represents. If the element offers a range of performance capabilities (*i.e.* a parameterised algorithm such as image compression parameterised by compression), it can use this value to determine how it should perform.

A Set element proxy must use the performance set-point to select and activate one element from amongst its sub-elements. The sub-element to select is the one which

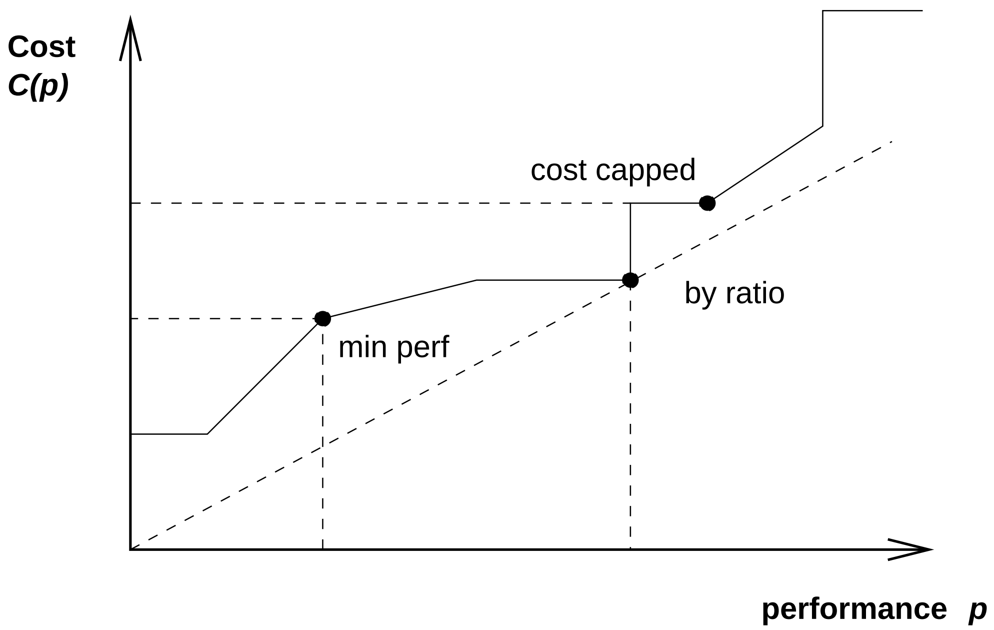


Figure 5.13: Application cost-benefit function with possible performance points dictated by different policies.

has the least cost at this point. Selection profiles may be built in whilst building the Set's cost-benefit function, providing a fast and convenient reverse transformation from performance point to a sub-element selection. If the selected sub-element differs from the currently active sub-element, reconfiguration is triggered. Reconfiguration involves the deactivation of the currently active element and substitution with the new selection. Reconfiguration proceeds as follows:

- the currently active sub-element is issued with a *deactivate* command,
- the new selection is issued with an *activate* command implicitly by passing the performance set-point,
- data imports to the Set element are rerouted to the newly activated sub-element elements. This is performed by informing those elements that provide the data of the new data destination,
- the data exports of the newly activated sub-element are directed towards the data destinations of the Set element by informing the sub-element of the appropriate destinations. The exports of the previously active sub-element remain intact, duplicating the data export paths of the new element. This ensures that any remaining data exports are still passed in the case of switching *glitches*; an element receiving the deactivate command should halt the generation or export of any additional data.

A Container element proxy must propagate the set-point to all of its sub-elements according to the defined notion of performance. Referring to section 5.4.4, for the two proposed notions of performance:

1. Independent performance with MinSumConv aggregation of the cost-benefit function requires determining the individual performance set-point for each sub-element such that the sum of the sub-element set-points equals the overall Container set-point and also produces minimum cost at this point. Such determination requires practically the same process as for aggregating the cost-benefit function and hence it is best to track the individual contributions of each sub-element in composing the Container cost-benefit function. In this way, determining the set-point to pass to each sub-element is a simple lookup in the saved composition information.

2. Cooperative performance defines the performance of the Container to be equal to the performance of each sub-element. Hence, propagation of the set-point is the trivial passing on of the Container set-point to each sub-element.

5.5 Summary

Building on the experience gained from the experiments of chapter 4, this chapter has outlined the principles underlying the DProj framework and described the operation of the framework to provide meaningful adaptation to applications.

Adaptation is provided in a pervasive manner by requiring that applications be constructed in a modular, hierarchical manner through decomposition into Basic, Set and Container elements. Dynamic manipulation of the tree through substitution of alternate functional elements during operation allows implementation maps to a wide variety of adaptation approaches. Table 5.1 (duplicated in part from from table 3.1) compares the DProj design to those reviewed in chapter 3.

The rigid application structure and the availability of detailed meta-data allow for an application to be monitored and manipulated by an external application monitor. The DProj application monitor uses a model of the application that mirrors its structure and allows aggregation of cost-benefit functions to provide a profile of an application's operational characteristics.

Basing the characterisation on unified cost and performance metrics is both natural and powerful, removing dependencies of the adaptation mechanism from resource specific concerns and from application specific notions of performance.

The invariance of the adaptation mechanism to the levels of the hierarchical mean that the granularity of adaptation is limited only by the granularity of the elemental decomposition of the application. The self-similarity in the model and its hierarchy structure allows further aggregation of multiple application models together into a single model structure, hence providing horizontally integrated adaptation to applications running in parallel.

The following chapter describes the exact implementation details of the framework, including tools and application development methodologies used to develop applications to operate within and take advantage of this framework.

Technology	Model	Condition Test	Online Alg.	Param. Alg.	Alg. Select.	Evol. Prog.	Characterisation
Mobile IP	T						n/a (support only)
IPv6	T						n/a (support only)
MobileSocket	TS	•					discrete
ICOPS,CAGES	T	•					discrete (static)
IDAP	T	•					discrete (static)
Coign	T	•	•				discrete (static)
RealPlayer	S	•	•	•			linear-discrete
MAF	TI	•	•				weak-discrete
Rover	I	•	•				weak-discrete
Odyssey	I	•	•	•			discrete (app. depend)
DProj	I	•	•	•	•		discrete (app. depend)

Model: **T**ransparent, **S**pecific and/or **I**ntegrated

Table 5.1: Spectrum and characterisation of DProj capabilities compared to reviewed mobile and adaptive systems.

DProj Implementation: Core and Peripheral Services

6

Chapter 5 described the basic design and structure of the DPROJ framework and the application structure required to operate with fully automatic pervasive adaptation. This chapter describes the implementation details of the DPROJ framework, including its underlying CORBA component system, component distribution patterns, service discovery facility and system instrumentation and modelling. In parallel, it describes how the implementation of the framework relates to the stages of application development. This includes the specification of application meta-data, details of the tools available to support this process and guidelines for mapping common application patterns and adaptation strategies onto application structures suitable for operation within the framework.

Further, the chapter outlines the implementation of several peripheral services that are required to support operation of applications using the DPROJ framework. These include application launch facilities, service discovery, resource handlers and management, adaptation policies, and associated user interfaces. The implementations of these services are minimal, sufficient for evaluation purposes but requiring a more comprehensive feature set for deployment in a user environment. Finally, some examples of future improvements required for operation in an end user environment are given.

6.1 OVERVIEW

The DPROJ framework as described in chapter 5 encompasses the development, deployment and runtime stages of an application. The implementation of the DPROJ framework presented here is built upon the foundations of existing services wherever possible. A well encapsulated, modular approach is aimed for to allow simple upgrading or improvement of the underlying services to more featureful or suitable implementations where possible and desirable.

DPROJ uses CORBA as its underlying middleware platform. As detailed in section 2.5, CORBA provides location, language and implementation transparent interfaces to CORBA objects and services. Additionally, OMG defines optional standard CORBA services which provide commonly required functionality for distributed systems development. Using CORBA allows rapid development of a DPROJ research platform, whilst allowing for future expansion of functionality.

The DPROJ framework is developed as four major components plus test applications, following the design of chapter 5:

1. **EC**: an application *Element Composition* language and toolset providing developer support in constructing applications conforming to the adaptation guidelines. The developer outlines the application structure in the EC language to provide the adaptation system with correct application meta-data. The EC toolset can generate application frameworks in CORBA and C++ from this description ensuring that the meta-data and the application proper are always consistent.

2. **Application Runtime Support**: providing the required adaptation interfaces and reconfiguration functionality to applications developed using the software structure automatically generated by the EC tools. The runtime support has application structure querying and reconfiguration primitives embodied in the C++ `StandardModule` superclass from which all EC generated elements are derived. Data passing, marshalling and location transparency are all handled by the underlying CORBA middleware.

3. **Application Monitor**: connects to and manages an application, performing monitoring of the resource use and performance of individual elements. The results of this monitoring are collated for use in adaptation decision making and resulting in the issue of reconfiguration orders to application elements by the monitor. The application monitor incorporates a simple resource manager with the capability to use an external resource manager if available.

4. **Policy Unit**: provides both resource and adaptation policy to the application monitor, using the monitor's cost-benefit aggregate to determine and issue performance set-points for applications. The unit offers a user interface providing feedback to the user of the current application cost-benefit profiles and also allows the user to to override the policy set-point.

Of these, the Application Runtime Support and the Application Monitor provide the core of the DPROJ framework. The EC language and toolset exists only as developer support while the Policy Unit is simple and would likely be replaced by a 'smarter' and more comprehensive equivalent in a real user environment.

Implementation details of these components follow — the implementations take advantage of the object oriented features of C++ and CORBA with an aim to maximise inter-operability, plug-ability of components, and to simplify the application development process as much as possible.

6.1.1 Implementation Language

The DPROJ system components take advantage of CORBA's cross-language specification by using different programming languages for different components according to suitability. To this end, the core application runtime is written in C++ for interaction with applications also written in C++. The application modelling and adaptation mechanism of the monitor are also in C++ for efficiency while user interface and policy control interfaces are in Python. The EC development tools are in Java.

Coupled with C++, CORBA provides a fast and powerful set of rigid interfaces and data types in a clean cross platform and cross language manner. However, for a system that requires so much application meta-data and introspection, a language with greater reflection and self-analysis capabilities such as Java might be more applicable.

Java's `java.lang` and `java.lang.reflect` packages provide access to an application's structure, class information, byte code and to the *Java Virtual Machine* (JVM). Java includes cross-platform execution, dynamic class loading and object serialisation which are all useful to a dynamic distributed system. Such features make it an obvious choice for research into distributed and mobile computing. Unfortunately, at the time of this project's commencement Java virtual machines suffered from poor performance. Even now, JVM performance lags behind that of native code and the nature of the virtual machine approach ensures that this will remain generally true. The initial stages of the project did target Java as the application language of choice before moving to C++ for improved speed at the cost of greater developer input in formalising the application structure and providing application meta-data.

With the application description of structure and datatypes pre-specified using the EC tools, the DPROJ framework has access to sufficient information to perform adaptation and reconfiguration for applications written in C++. Additionally, such an approach can apply greater restrictions to the composition of an application and the interactions between the parts. For example, the current scheme restricts element data dependencies to be fundamentally asynchronous operating using a push model, enabling a simpler adaptation process while at the same time forcing developers to develop applications in a manner more aware of their distributed nature.

6.1.2 CORBA Services

Operation of the framework relies upon CORBA as its underlying middleware and also upon some standard CORBA services for specific functionality. In particular, it uses the CORBA COS Naming Service as its coordination medium.

When components are activated, they register themselves with a central COS naming service. Other components may discover relevant parts by name. For example, an application monitor queries the naming service to discover all of the elements comprising an application. It may also then check availability of a resource manager for each host.

6.1.3 Development Process

The DPROJ framework provides automatic adaptation to suitably constructed applications. To do so, it provides not only runtime adaptation support for such applications but also a tool set and guidelines for application development.

The following sections describe the implementation of the tools and application runtime which enable the construction of these applications. In order to understand their positioning within the system, it is useful to examine the overall application development flow.

Figure 6.1 outlines the basic developmental procedure including necessary inputs, tools and products. Development originates with a meta-level description of the application elements provided by the developer using the EC language. This is processed by the ECgen processor, producing CORBA IDL descriptions of the data types and interfaces involved. Additionally it produces C++ superclasses for each application element, conforming to the interfaces defined. These superclasses themselves are sufficient to produce a runnable application although it will contain no useful functionality. The developer subclasses the element superclass for each specific element implementation. Finally the build system compiles the element implementations and collects them into a code pool. Code pools consist of a collection of binary object code, with each pool targeted to a different host class and each containing sufficient information to dynamically construct the application from the elements contained within the code pool.

6.2 ELEMENT COMPOSITION

Applications are described in terms of their elemental make-up using the simple Element Composition language, EC designed for this purpose. The EC description processor, `ECgen`, parses the EC description to generate skeleton source code for the application.

EC allows a developer to specify application datatype, elemental and composition meta-data separate from element implementation concerns much in the same way that CORBA datatypes and interfaces are specified in IDL. Datatype and elemental Descriptions written in EC hence have low interdependencies; this encourages code reuse between applications, an important aspect of development particularly when common third party services or elements may be available to many applications.

EC works with two major components: data types and modules. Each component is specified independently in its own file. Dependencies between components are by reference and even though applications are composed hierarchically, modules may not be fully specified hierarchically within the one file. Keeping the specifications independent simplifies EC processing tool implementation, increases maintainability and encourages code reuse at the expense of a little developer tedium. Tables 6.1, 6.2 and 6.3 give BNF descriptions of the EC language[60]. Example EC definitions for the SerialTest application are given in appendix A.

EC Datatypes

An EC datatype represents a unit of data that may be passed between application elements. A datatype may be one of three sub-types: primary, struct or array, allowing elements to interface through useful data structures as opposed to the integer-only limi-

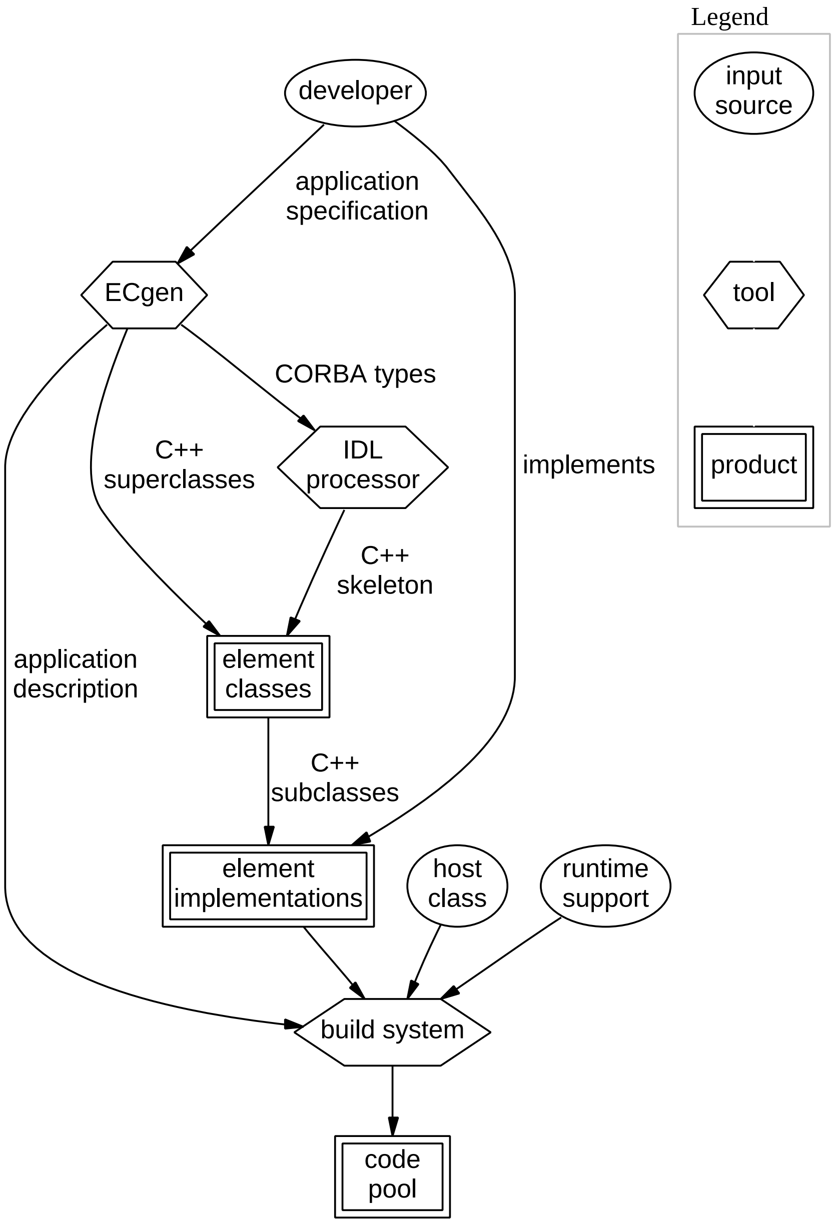


Figure 6.1: Application generation procedure.

component	::= datatype | module

Table 6.1: EC component specification in BNF.

datatype	::= **DataType** <typename> "{" class description "}"
class	::= **type** subtype
subtype	::= **primary** | **struct** | **array**
description	::= primary_desc | struct_desc | array_desc
primary_desc	::= **size** <int>
struct_desc	::= struct_member { struct_member }
struct_member	::= <membername> <typename>
array_desc	::= **size** <int> **of** <typename>

Table 6.2: EC datatype component specification in BNF.

tations of the adaptive control system of chapter 4. A primary datatype represents a fundamental datatype of the target language such as integers, characters and strings. ECgen knows Java and C++ mappings for the following predefined primary types: int32, int16, int8, string, and buffer. The first four of these are self-explanatory. A buffer is a special primary type which maps to a CORBA Sequence of bytes. A struct datatype represents a structure containing other EC defined datatypes as members. This is equivalent to structs in C++ or CORBA. An array datatype is equivalent to arrays in C++ or Sequences in CORBA, comprising a fixed length vector of some EC defined datatype.

Datatype specifications include a size attribute. For primary types, this gives the size in bytes of the datatype and is used to calculate communications usage when passing that data type. The size of structs is calculated from the size of its member datatypes while the size attribute of an array gives the number of members.

EC Modules

A module is equivalent to a virtual element[1] and may be one of two sub-types: normal and container, corresponding to Basic and Container elements respectively. Note that there is no Set module as Sets are implicit within the element hierarchy.

All modules specify some attributes, a set of modifiable *properties*, and their data dependency *imports* and *exports*. Currently the only module attribute is the optional language specification which indicates the implementation language of the module.

The property list of a module define some variables of the module which may be varied and set at element instantiation time. A property may be of any primary datatype and a default value is given. Additionally, a type `dummyType` is also provided allowing the encoding of EC processor directives that do not translate directly to module implementations. Some standard properties are defined by the DProj framework and

[1] the difference in name has risen only through a historical oddity; for the purposes of clarity, the term *module* is used to refer to a module definition in the EC specification while *element* will refer to an element as a component of an application.

<table>
<tr><td>module</td><td>::= Module <modulename> "{" class lang_spec description "}"</td></tr>
<tr><td>class</td><td>::= type subtype</td></tr>
<tr><td>subtype</td><td>::= normal | container</td></tr>
<tr><td>lang_spec</td><td>::= language lang_type</td></tr>
<tr><td>lang_type</td><td>::= C++ | Java</td></tr>
<tr><td>description</td><td>::= property_list [port_list] [container_desc]</td></tr>
<tr><td>property_list</td><td>::= { property }</td></tr>
<tr><td>property</td><td>::= property <propertyname> <typename> <value></td></tr>
<tr><td>port_list</td><td>::= port_desc { port_desc }</td></tr>
<tr><td>port_desc</td><td>::= export_desc | import_desc</td></tr>
<tr><td>export_desc</td><td>::= export <exportname> <typename></td></tr>
<tr><td>import_desc</td><td>::= import <importname> <typename></td></tr>
<tr><td>container_desc</td><td>::= container content_item { content_item }</td></tr>
<tr><td>content_item</td><td>::= module_item | wire_item | property_item</td></tr>
<tr><td>module_item</td><td>::= module <member> <module> [(<int> , <int>)]</td></tr>
<tr><td>wire_item</td><td>::= wire <member>.<exportname> – <member>.<importname></td></tr>
<tr><td>property_item</td><td>::= modproperty<member>.<propertyname> <value></td></tr>
</table>

Table 6.3: EC module component specification in BNF.

modification of these values allows control over some aspects of the distribution and adaptation of an application. In particular, the *serverside* and *clientside* properties list implementation variants of the module which may operate on server or client host classes respectively. This allows the developer to specify default implementations for each of these host classes. A null value indicates that there is no implementation of the module for that host class. As will be described below, a container module can modify the properties of its member modules; this allows customisation of modules in different conditions and encourages code reuse: different instances of the same element may thus be parameterised according to their place in the application or the application itself.

The import and export lists of a module define its data interfaces by name and datatype. The trigger conditions are not specified here but are implementation specific.

A Container module specifies in addition its contents which may be any number and combination of other modules, data dependency wiring between those modules and modifications to the properties of its member modules. A Container module may also have imports and exports with corresponding wiring between them and those of the contained modules. An application is a Container with no imports or exports (*i.e.* is self contained).

ECgen

ECgen processes module and datatype descriptions to produce a framework of code for the application. From an application specification, it recurses the module and datatype dependencies to produce a complete source code framework for all required components.

Figure 6.2 shows the code generation performed by ECgen to produce CORBA definitions and C++ skeleton code for an application. This figure shows how the generated C++ code fits with the runtime provided components, primarily through the use of subclassing runtime components. For each datatype, it generates a CORBA IDL equivalent which defines the corresponding data structure passed between elements in the application. A standard CORBA IDL processor will use this to generate the actual C++ implementation for the datatype. Additionally, for each datatype, the IDL output describes *gateway* interfaces which export and import that datatype. Again, the standard CORBA IDL processor will generate skeleton and stub implementations of these interfaces. Gateways provide controlled datatype specific data dispatch and receive for application elements. However, as CORBA interfaces have fixed names, an element would be unable to have more than one gateway of the same datatype. To overcome this, each gateway interface is implemented by a gateway object which is then owned by an element. As several such gateway objects can be owned by the one element and the objects are distinguishable (by name), the element can achieve any number of similar imports or exports. To this end, ECgen generates the C++ gateway object implementations for these datatypes.

For each module, ECgen generates a C++ class which provides a skeleton for that module. The class contains initialisers, properties and the required import and export gateway objects. The skeleton is derived from the runtime supplied StandardModule class that provides runtime functionality. These generated structures are described further in sections 6.2.1 and 6.2.2 below.

Additional Tools

Additional tools build on ECgen to provide greater developer support in application specification and construction:

ECtool provides graphical representation of EC datatypes and modules, showing interfaces and data dependencies within a Container. The developer may use ECtool to manipulate element composition graphically and generate both EC specifications and thence the IDL and C++ skeletons as when per ECgen.

ECdot translates EC module specifications to a format suitable for automatic graph generation using the GraphViz package[51]. It provides full hierarchy recursion and can expand the graph to a selected level, resulting in complete element hierarchy graphs for visualisation. All application graphs in this dissertation are produced using ECdot.

6.2.1 Datatypes and Gateways

Table 6.4 shows the mapping from EC primary datatypes to CORBA primary types. EC structs map directly to CORBA structs and EC arrays map to CORBA sequences.

Import and export gateway interfaces are defined in IDL for each EC datatype. Gateways provide datatype specific encapsulated data channel handling for inter-element

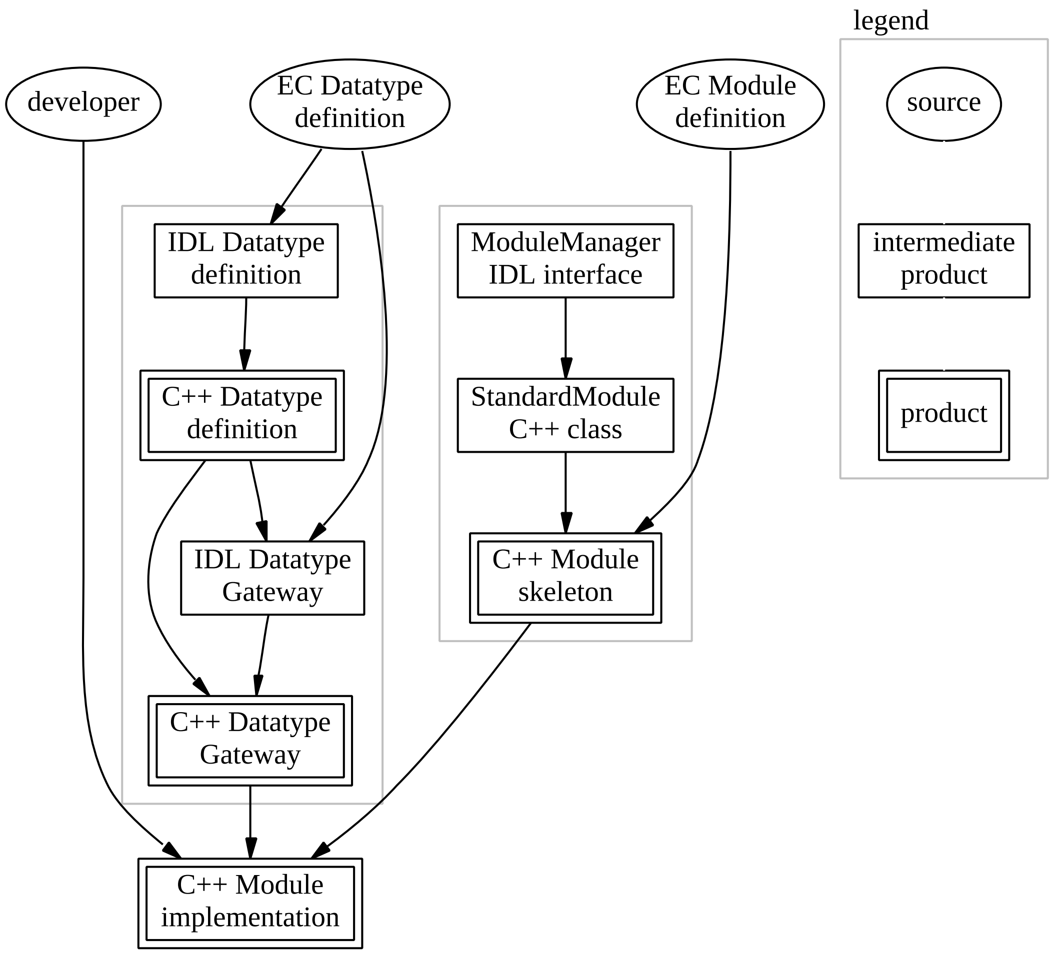


Figure 6.2: ECgen code generation products.

EC primary	CORBA IDL
boolean	boolean
int8	char
int16	short
int32	long
string	string
buffer	sequence<octet>

Table 6.4: Mapping of EC datatypes onto CORBA IDL types.

data passing. They enable logging of data traffic levels passing through that gateway and also redirection of data when instructed by a reconfiguration order. All Gateways support the following query to report the current amount of traffic being generated by the gateway:

Gateway Queries	
query	**returns**
`traffic`	total communications traffic through this gateway in bytes

In addition, export gateways support reconfiguration of their data destination using the following method:

ExportGateway invocations	
invocation	**action**
`setDest`	set the export destination for this dataflow. may be NULL to cancel exports

Import and export gateways are implemented as individual C++ classes which are then owned by an element. An element's gateways are reported to the application monitor upon request, allowing it to act directly upon the gateways to perform traffic monitoring or data redirection.

6.2.2 C++ Element Skeleton

For each module, ECgen produces an element skeleton as the minimal implementation of that module. This skeleton inherits standard CORBA ModuleManager element interface and standard element operations from the runtime provided StandardModule superclass. These provide an interface through which an application monitor can gather information about the element and then manipulate its operation.

All elements support the following queries:

Standard element queries	
query	**returns**
`name`	the base name of the element
`variant`	the exact implementation name of the element
`kind`	the element *kind*, being one of *basic, set or container*.
`location`	a flag indicating the current location of this element.
`getImports`	a list of dataflow imports which the element accepts.
`getExports`	a list of dataflow exports which the element provides.
`getResourceReq`	the resource profile and performance estimate of the element.

In addition, a Container element supports queries on its contents:

Container element queries	
query	**returns**
`numModules`	the number of contained functional elements.
`childModule(n)`	a reference to sub-element `n`.
`numWires`	the number of internal wires between contained sub-elements.
`childWire(n)`	a description of internal wire `n`.

Container elements also expose a set of methods allowing the application monitor to control its operational state:

Container element invocations	
invocation	**action**
`init`	initialise the container and children.
`run`	commence execution of container and children.
`stop`	halt the container and children.
`pause`	pause the container and children.
`resume`	resume execution of container and children from a pause.

For Basic elements, the element skeleton contains the required import and export gateways for the function but has no actual functionality, per se. Functional implementation must be developer provided through subclassing.

For Container elements, the element skeleton contains no gateways but instead a list of contained sub-elements and the data dependency wiring between them.

6.2.3 C++ Element Implementation

An element implementation is produced by subclassing the element skeleton and overloading its methods to provide the required functionality. By convention, the implementation's class name is the concatenation of the name of the skeleton and the implementation variant.

There are three points at which a developer may provide element functionality at three points:

1. `init`**-time**: for clarity, the developer should not overload the C++ constructor but perform all initialisation through overloading of the `init` method. As elements are contained in dynamic libraries which are loaded at runtime, initialisation is performed outside of the constructor to allow greater control of the timing of its execution. Any code that initialises element variables, hardware or conditions should be placed here.

2. `run`**-time**: the `run` method is invoked when the application is instructed to begin execution by the application monitor. At this opportunity, the element may commence operation. The method must return so if the element requires some

continuous operation, it may spawn its own thread here. If so, the element implementation should also overload the `stop`, `pause` and `resume` methods appropriately.

3. **data import time**: whenever new data arrives at one of an element's import gateways, the corresponding element import function is invoked with the data passed as a parameter. At this point the element may process the data or process its trigger conditions if it is waiting for data from more than one import gateway. This will normally be where the bulk of an element's processing will occur for any application that is data driven.

Additionally, an implementation may overload the `getResourceReq` method to report its resource requirements and performance expectations. The element can report a required quantity for any number of supported resource types above that which is implicit in its operation (for example, communications and CPU usage may be measured independently through instrumentation of the element).

The procedure for the reception of a data import is as follows, for an import named 'example' of datatype 'int32': the data is received by the gateway it is directed to; this gateway then makes a callback into its owning element of the method named `import_example(int32 example)`'. This method is overloaded by the developer to provide element functionality. Export of data may be performed at any point by calling the relevant method (`import_example(int32 example)`) in this case). This will relay through the export gateway which will forward the data to its correspondent. As the data forwarding is one-way, the export may be expected to return immediately although this is typically an ORB dependent characteristic. The traditional rules of CORBA interfaces regarding data ownership apply: the 'caller' retains ownership of the data; if an element wishes to keep the data it receives, it must make a copy. Strict adherence to memory management is essential to prevent memory leaks in a distributed system.

6.3 Application Runtime

The DProj application runtime support provides interfacing and configuration functionality to application code automatically generated by the ECgen processor. The application runtime supports on-demand loading of element implementations into the running application. In this manner, the runtime can fetch implementations into the application code pools from remote filesystems or web servers much in the same way that the java ClassLoader can dynamically fetch and load java classes from internet web servers, as used in Jini's code pull mechanism[93]. The current implementation features an application launcher, AppLaunch, which fetches implementations from a distributed file system by name to form a running application code pool. Dynamic loading requires that each element skeleton and implementation be compiled as a separate dynamically linked library. Additionally, for each Container element, a dynamic library containing the datatype definitions (including gateways etc) is built.

Starting from the application's top-level Container, AppLaunch recursively fetches, loads and initialises all of the elements and datatypes in an application code pool. Each Container contains information as to which libraries are required for its sub-elements and the required datatype libraries. Additional non-DPROJ specific libraries may also be specified as dependencies of an element.

AppLaunch is run on the host on which the application code pool is to be launched and supports three command line parameters:

AppLaunch parameters	
parameter	**controls**
`appname`	the name of the application to launch, used to identify the top-level Container of the application.
`mode`	the host classification to be used for this CORBA marshalling, host. Available modes include client and server.
`id`	the identifier under which the application should be registered with the naming service.

The runtime registers the code pool's top-level Container with the CORBA naming service as a CORBA ModuleManager interface. It is named according to its application id and mode:

`DProj.Applications/`<*id*>`.POOL`<*n*>_<*mode*>_<*hostname*>

where *n* indicates a unique host number. The id will default to the be the same as the name of the application, however in the case where more than one instance of the same application are running concurrently, it may be specified to distinguish between them. This allows the application monitor to correctly associate code pools with application instances.

6.4 APPLICATION MONITOR

The application monitor performs monitoring of resource use, application performance, adaptation decision making and issuing of reconfiguration orders to the application.

The application monitor discovers the code pools via the CORBA naming service. In particular, the DPROJ framework does not make any attempts to determine which application elements are able to run on which host nor how hosts should be classified. Nor does it provide facilities to search for services beyond those registered with the CORBA Naming Service. Such facilities are obvious extensions to the framework but lie outside its current scope. The current framework implementation focuses on making the best use of the services offered to it.

The application monitor is centred about the hierchical application model, constructed from many element *proxies* echoing the hierarchical structure of the application. Each proxy is a CORBA object acting as an interface to and manager for an element (either Basic or collective) in the application. Through its CORBA interface, the model may itself be distributed across the system, although typically it is based on the mobile client — this allows adaptation to continue to function usefully (for the mobile user) even in a disconnected environment.

Each proxy presents the same interface:

Proxy Model Interface	
method	**action**
`name`	get or set the name of the element represented by this proxy.
`variant`	the exact implementation name of the element represented by this proxy.
`kind`	the proxy *kind*, being one of *basic, set or container*.
`location`	a set of flags indicating the potential host for execution of the element(s) represented by this proxy.
`getPerformance`	get the cost-benefit function of this proxy.
`configure`	order the proxy to reconfigure its representative elements to perform at a given point on the cost-benefit function.
`getMod`	get a reference to the actual element that this proxy represents.
`getMembers`	get a list of references to the child proxies of this proxy1.

This interface allows monitor control of that proxy and the full tree of proxies under it. General operation of the monitor involves iterative re-configurations of the application through the proxies' configure method whilst gaining feedback in the form of a cost-benefit function via the getPerformance method. The getMod and getMembers methods provide instrumentation support, allow for greater interrogation and monitoring of the the application structure and performance by external processes such as debugging tools.

The application monitor exposes the interface of its top-level proxy by registering it with the naming service, allowing an external process to make the top-level adaptation decisions. The decision maker will be either a policy unit of some choosing or another application monitor which may adopt this monitor as a sub-proxy in its set (and hence provide a method by which the models for several applications may be brought together into a single system model). In addition to the top-level proxy interface, the monitor exposes an *application monitor interface* through which its resource manager and policy may be manipulated:

Application Monitor Interface	
method	**action**
`getResourceManager`	get a reference to the resource manager currently in use.
`setResourceManager`	set the monitor to use a specific resource manager.

By default, an application monitor will look for a resource manager via the CORBA naming service. If a resource manager is registered in the naming service, it will be the default resource manager to all monitors which are then launched.

6.4.1 Resource Monitoring

To build an accurate model of the application, the monitor must know or estimate both the resource requirements and the expected performance of each element.

The resource manager can only give a cost for the resources, it does not determine usage. To this end, the application monitor implements monitoring of an element's communications; use of other resources may be explicitly specified by an element. Elements may also specify an update period, giving a measure of how frequently it expects its requirements to change. This can be used by the monitor to economise on information gathering. Current resource monitoring focusses on CPU, communications and battery.

CPU requirement is reported directly by an element. A declaration of CPU usage also implies some battery usage. A developer must determine the approximate requirements of an element either by specifying it explicitly or adding a function to the element to calculate it. Ideally, the runtime should be able to actually measure the CPU use of an element; this would be possible if using an interpreted language for the implementation (such as Python or Java) or an operating system extension, for example to make use of the instruction counting registers on the Intel Pentium or DEC Alpha processors.

The monitor does not police estimates but relies on accurate reporting from the elements. If the requirements of an element are dependent on some operational or data parameter, the requirements may vary over time. An element that is inactive does not receive data and so its best estimate will usually be the value previously reported. If an element has never been activated, then it should be able to report a usage based on its default parameters.

Communications requirements are actively instrumented by the application monitor by using the traffic logging facility of the import and export Gateways. These report the total traffic through that gateway in bytes and are used by the monitor to calculate the traffic rate. This value is filtered using a first order low pass filter to smooth the readings.

Figure 6.3 shows the effect of the filtering on both a sudden spike in traffic (presumed to be an anomoly) and to a step in traffic (more likely a genuine change in traffic pattern). Also shown is the effect of a second order filter. Both are equally acceptable and the choice of filter should be driven by the expected fluctuations in traffic, the reconfiguration period, and the rate of adaptation required. The first order filter has transfer function:

$$y_t = 0.75y_{t-1} + 0.25x_t \tag{6.1}$$

where x is the measured reading and y is the reported reading. With these values, the reading will approach 90% of the correct value within 8 samples. Note that the filtering is applied to estimates of the communications traffic usage and not to the communications cost.

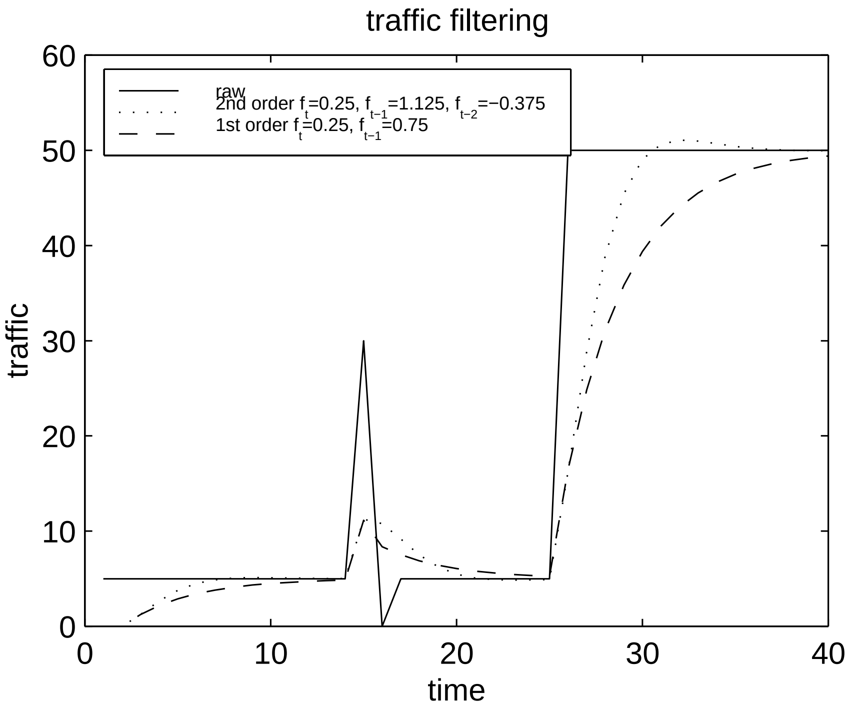


Figure 6.3: Traffic monitor filtering.

As per CPU usage, communications requirements can be dependent on some operational or data parameters and so inactive elements need special consideration. In this case, the traffic of a particular gateway is given based on previous estimates. If an element has never been activated then there is no prior estimate and the result of the currently active element is used in its place.

Battery requirements are implicit in both CPU and communications requirements but some elements may wish to report extra requirements in special cases. For example, if an element implements or represents some functionality that uses special hardware, then this will be reflected here.

6.5 Resource Manager and Policy Unit

The resource manager and policy unit implemented for testing and research deployment are simple with a high level of user interaction and a heavy reliance on user input for decisions. They provides adequate functionality for a test environment with a high degree of user control. Written in python, these both benefit from simple extensibility for implementation of custom scenarios.

Figure 6.4: User guided resource manager — the user may manually set the cost of individual resources and also the available communications capacity. The relationship between communications capacity and cost is determined by the currently operating resource policy.

6.5.1 Resource Manager

A simple resource manager is provided, allowing user control of resource costing and availability. This allows fine control of the resource profile presented to running applications and hence observation of the application's *open loop* response to resource availability (evaluation criteria is covered further in section 7.1). The resource manager may be easily modified to provide *closed loop* or scenario driven resource profiles.

A snapshot of the resource manager user interface is shown in figure 6.4. Here, the user has control over the processing costs on each of four possible hosts (S, C, W and X) and over the capacity and cost of the shared communications medium.

6.5.2 Policy Unit

The policy unit provides the top-level adaptation decision to the application monitor through the exposed top-level proxy interface. It may also, optionally, override the application monitor's built-in resource manager with one of its own choosing.

The policy unit acts as top-level decision maker for all application monitors registered in the naming service. It allows detailed inspection of the application element proxies currently operating in the system. In particular, it allows selection of a *focus* proxy which is then regarded as the top-level proxy in the system. This allows the user to narrow the operation of the policy unit and hence the adaptation process to focus on this proxy: further operation of the policy unit, including determination and selection of performance set-points, is through this proxy and hence restricted to the model sub-tree at that level; the remainder of the model tree is excluded from analysis and adaptation. If the user does not select a particular focus proxy, the top-level proxy of the first application is selected by default.

Figure 6.5 shows the test policy unit user interface. Its window is divided into two three main panels.

1. The *model browser* is the tree style widget on the left of the window. It presents an expandable view of the element proxy hierarchy for all running applications, expandable/collapsible at collective element granularity.

   Proxies for application Containers, collective elements and Basic elements are each represented by different icons and labeled by the name and variant of the proxy. The user may select an icon to expand/collapse the tree or select the name of the proxy to make it the focus proxy. The new focus proxy is opened for examination in the *distribution* and *performance panels*.

2. The *distribution panel* located in the upper right of the window shows the member proxies of the currently selected proxy. These are arranged using the same coordinate system as the ECtool viewer and colour coded to show the current location of activation for each member. If a member is a proxy for a Container element, its location colouring may be misleading as the locations of the sub-elements of such an element need not be all the same.

3. The *performance panel* gives graphical feedback on the performance characteristics of the currently selected proxy as two graphs. The upper graph shows the cost-benefit function as reported by that element, the y-axis plots the estimated cost in microdollars per second versus performance on the x-axis (values 0 to 1000). If the selected proxy represents a Set, the lower graph is used to show the selection profile for the proxy, indicating which of its sub-elements would be activated at each performance point.

The policy unit allows the user to set the performance point by clicking on the appropriate point in the performance panel. This chosen point is applied to the focus proxy and is relayed to it using its `configure` method. The reconfiguration process is hence constrained to that proxy and those below it in the application model tree. The reconfiguration order is repeated at one second intervals to ensure continuous adaptation.

Note that the current implementation is limited in its support for multiple concurrent applications in that it provides neither a top-level super-proxy to aggregate individual application models into a system model, nor does it support simultaneous adaptation in more than one disjoint model sub-trees. However user directed adaptation of multiple applications may be performed through concurrent operation of more than one policy unit, each focused on a different

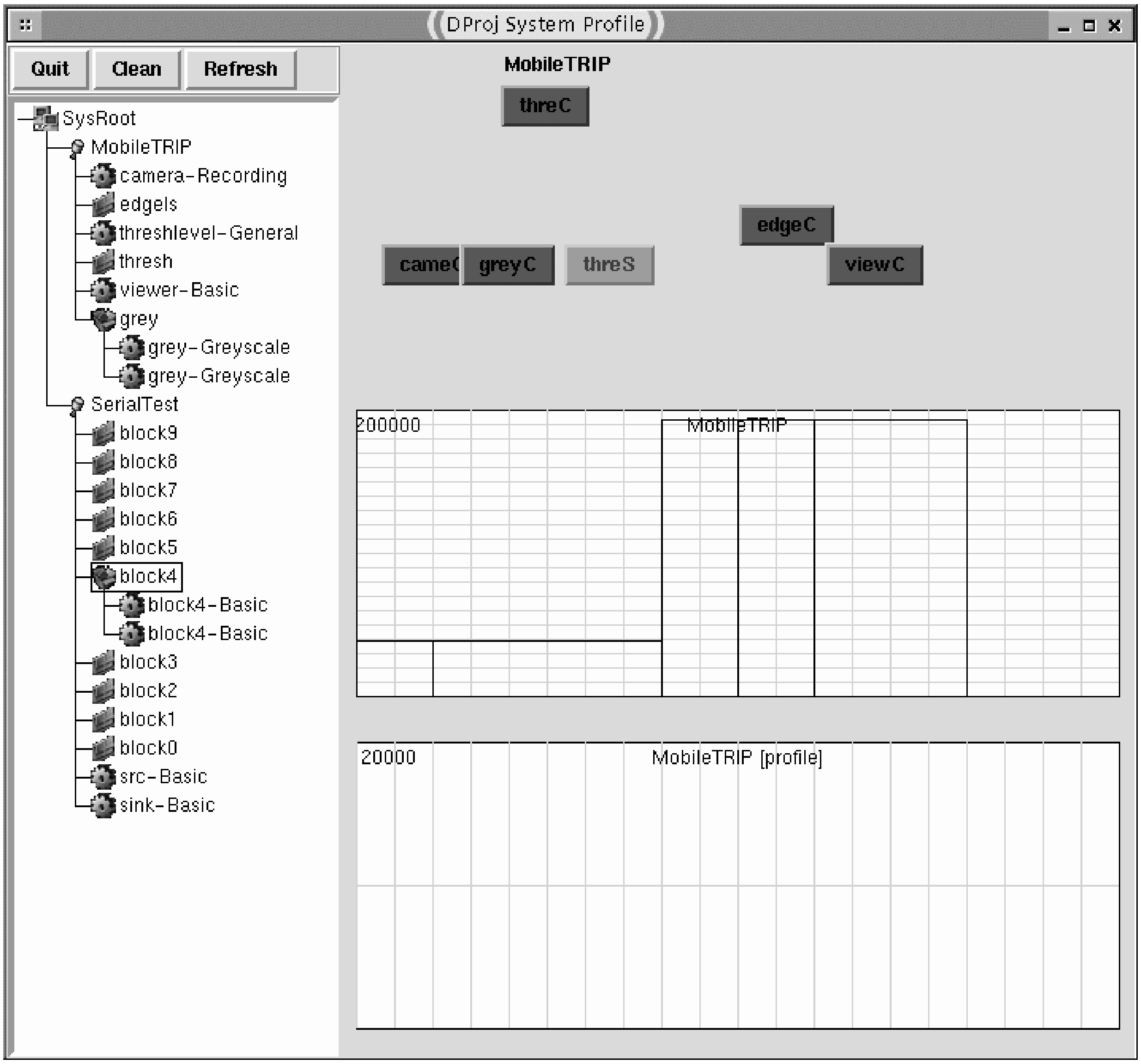


Figure 6.5: Policy Unit user interface. The model browser tree on the left enables inspection of the complete system model while the distribution and performance panels show the location of sub-elements and cost-benefit function of the selected *focus element.*

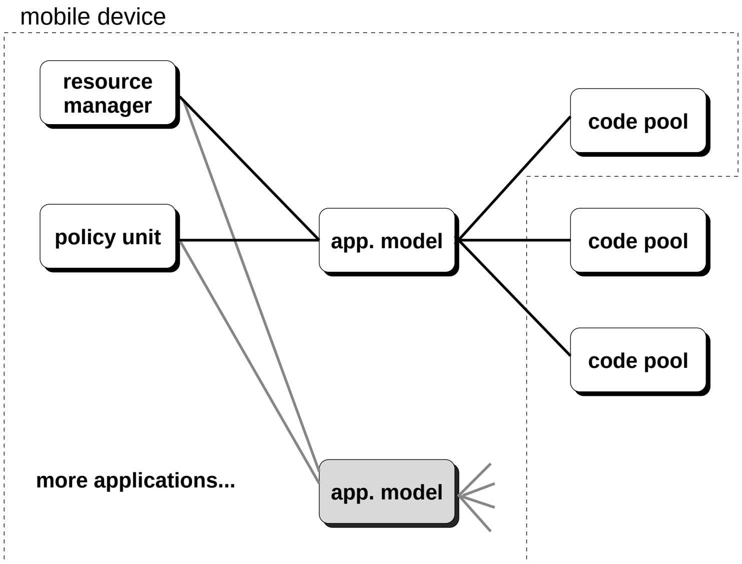


Figure 6.6: A typical distribution of DPROJ application and runtime components; the presence of additional concurrent applications are shown in grey.

6.6 Application Operation

A running application is composed of several process components operating on one or more hosts, but with the user focus on the mobile host. Figure 6.6 shows a basic distribution in which the majority of the components are located on the mobile host. This case is the simplest configuration which allows disconnected operation; the mobile host code pool contains sufficient functionality to keep the application operational while the complete application monitor (model), resource manager and policy unit are always available to the mobile user. The actual distribution or 'allowed execution areas' for application components is entirely at the discretion of the application developer. The two host classes of 'client' and 'server' provide minimal support for general capability based element distribution while finer control again must be explicitly provided by the developer.

6.6.1 Overhead of Adaptation

To be useful, an adaptation service should save more resources than it consumes. Hence, quantifying the overhead of adaptation is an important concern. The exact nature of the resource savings are of course highly dependent on the nature of the application, the structuring of the application by the developer and the mobile system in which it runs.

It is more instructive then to examine in general terms the cost involved in providing the adaptation service. This cost is incurred by the application monitor through the use of two key resources: CPU and communications. For demonstration, consider a single application with a hierarchical structure consisting of N_e elements in a T-tier hierarchy deployed in such a manner that there are N_i element instances available. In a well structured application in the average case, $N_i \propto N_e$ and T will be of the order of $log(N_e)$. Generalising to a single variable N:

$$\begin{array}{ll} \text{number of elements} & N_e = N \\ \text{number of element instances} & N_i \propto N \\ \text{application structural depth} & T \ \propto log(N) \end{array} \tag{6.2}$$

As the model constructed by the application monitor echos the structure of the application, it will have $O(N)$ proxies and a depth of $O(log(N))$.

Each cycle of the application monitor progresses in two main stages: cost-benefit aggregation and performance set-point propagation. Both involve complete spanning of the model tree although some optimisations are available to reduce this. Spanning the tree involves $O(N)$ proxy actions. The 'cooperative performance' scheme described in 5.4.4, used for aggregation in this implementation of the DPROJ application monitor, has complexity proportional to the tree fan-out and hence may be approximated as constant for each action. Similarly, the set-point propagation will also have a constant average complexity.

Thus, the overall CPU requirements of the adaptation process will be $O(N)$, *i.e.* proportional to the granularity of the application's elemental composition. Similarly, when the application monitor is resident on the mobile host, its communications usage consists of polling each of the activated elements for communications traffic (both import and export) and performance statistics, representing an $O(N)$ cost.

In summary, the overall requirement of the adaptation process is therefore $O(N)$, proportional to the granularity of adaptation. This shows that the performance of the adaptation mechanism should scale well with application complexity.

The actual requirements in a working system will be dependent on the system parameters, with particular consideration of update frequency and application monitor distribution:

6.6.2 Update Frequency and Rate of Adaptation

The application monitor adaptation mechanism proceeds incrementally through iterations of the cost-benefit aggregation and performance set-point propagation processes. The rate at which the application monitor performs this is termed the *update frequency* and determines the rate of adaptation of the application.

The update frequency has a direct (and proportional) effect on the resources required by the adaptation process and so a lower frequency implies lower resource use but the frequency must also be maintained sufficiently high as it will affect how fast the adaptation process can approach the best configuration for an application and the responsiveness of the adaptation to a sudden change in the environment.

The choice of update frequency is therefore dependent on both the environment, in particular the expected rate of change of resource availability, and the application, in terms of its precise nature and expected performance. Different update frequencies can be maintained for each application in the system with the overall system adaptation operating at the greatest frequency at the top level — the update frequency may even be derived from the character of the application's cost-benefit function. Similarly, the update frequency can be manipulated for different sub-trees of the application's structure, emphasising or deemphasising adaptation in selected functional areas. The current application monitor updates across the whole tree at the rate dictated by the policy unit.

6.6.3 Application Monitor Distribution

In a true user environment with multiple concurrent applications of greater structural complexity, the application monitors (and in particular the application models) can become equally complex and strain the capabilities of the mobile host.

To aid this case, the application monitor may itself be distributed to echo the distribution of the application itself. In this situation, the application model tree is partitioned and distributed to minimise transaction 'distance' between: a) model proxy and element; and b) model proxy and parent/sub-proxies in the tree. Such minimisation is dependent upon the application context, in particular the impact such distribution will have upon resource availability and responsiveness of adaptation.

Although the current implementation does not provide for such distributions, this has the theoretical potential to spread the CPU usage of the monitor and reduce communications overhead. To implement this, the proxies that comprise the application model would be distributed according to the following rules:

1. if the proxy represents a Basic element, then it should be colocated with that element, or
2. if the proxy represents a collective element (Set or Container) then:
   (a) if all of its sub-proxies share the same location, it should be colocated similarly, or
   (b) if one or more of its sub-proxies is located on the mobile host then it too should be on the mobile host, otherwise
   (c) the proxy may be located on one of the fixed servers, preferably the one which is best represented amongst the locations of the sub-proxies.

6.7 Application Design

Applications operating within the DProj framework must be constructed to meet the criteria of the framework. They must be highly componentised according to dataflow, must be aware of the distributed nature of the system and must pay strict attention to the storage of any state within its component elements.

It is instructive to follow some general guidelines for mapping application designs onto DProj structures. This section detail how the adaptation spectrum and characterisations of section 3.2 equate to operation within the DProj framework. It continues by describing a general approach to DProj application design, in particular how to develop applications that comply with the limitations of the framework.

6.7.1 Mapping the Adaptation Spectrum

Section 3.2.1 and figure 3.1 showed the spectrum of adaptation capabilities and offered broad categories: condition testing, online algorithms, parameterised algorithms, algorithm selection and evolutionary programming. The first four of these are easily incorporated within the DPROJ framework while evolutionary programming remains a special case.

Condition testing maps to alternate sub-elements within a Set.

Online algorithms generally apply to resource management and hence may be implemented in the specific resource handlers of the resource management system.

Parameterised algorithms may be mapped as discretely parameterised algorithms with duplicated algorithm elements as alternates within a Set, each with a different parameter set. Additionally, as each block receives a 'requested performance' indicator, it may be parameterised in one dimension, allowing continuous parameterisation within that element.

Algorithm selection maps to alternate sub-elements within a Set.

Evolutionary programming is not directly supported and does not map cleanly onto the DPROJ framework. Moreover, evolutionary programming targets a different problem; the type of adaptation it provides is geared more towards generating solutions beyond those originally coded by the developer.

6.7.2 Characterisation

The classification of the adaptation process based on the characterisations of section 3.2.2 serve to classify applications according to their perceived performance under different resource conditions. The character of an application is hence a characterisation of its cost-benefit function.

In the DPROJ framework, the key adaptation mechanism is achieved through switching between alternate sub-elements of a set. This switching is necessarily discrete and so the fundamental character of an application adapting through the DPROJ framework is also discrete. However, as each element may also report a linear cost-benefit function, the overall adaptation profile may also be piecewise linear. Such a profile may then approximate non-discrete characterisations such as weak or strong adaptation. As the structure of an application within the framework allows decomposition of an application into elements at an arbitrary granularity, the resulting cost-benefit function may also achieve an arbitrarily close piece-wise linear approximation of a continuous function.

6.7.3 Application Scope

When designing an application for operation within the DPROJ framework, it is important to consider the scope of an application to encompass the complete end to end path of its operating data. Contrary to established distinctions in which a client-server system would be composed of a client application and a server application (with perhaps a proxy application in between), a DPROJ developer should consider this entire chain as a single application to ensure the largest scope for adaptation.

For example, a audio or video streaming application should consider the media source to be part of the mobile application. The third party service provider feature of design principle 5.3 states that while the server component should be considered to be part of the application design, its actual implementation may be externally sourced.

6.7.4 Working Around System Shortcomings

The DPROJ adaptation framework in its current incarnation takes several shortcuts to simplify its implementation and operation. In particular the mandated asynchronous dataflow between elements and the stateless requirement of 'relocatable' elements are constraints which do not fit well with traditional application design.

Asynchronous Dataflow

To recap, the dataflow between functional elements of an application operating within the DPROJ framework is defined to be asynchronous. Asynchronous interaction is better suited for distributed situations, particularly for dealing with latency[94]. As in this case the application developer does not know in advance which elements will be co-located and which will be separated, all interfaces between elements are defined to be asynchronous. Asynchronous dataflow presents two key obstacles for developers: firstly, how to replace synchronous calls, and secondly, how to synchronise parallel dataflows.

In a synchronous call, data is exchanged in both directions during a method invocation: parameters are passed to the receiver which, after some processing, returns some data; the caller is blocked until the call returns. With a location-transparent middleware such as CORBA, this call may be either local or remote if the caller and receiver are on separate hosts. In the case of remote calls, the blocking time will be significantly greater than in the local case. Using asynchronous calls allows the caller to perform useful work instead of blocking.

To simulate a synchronous call under DPROJ, the call and return must be decoupled into separate exports and imports. When the parameters are exported, the caller is free to continue processing. When the receiver is ready to return data, the appropriate import of the caller is invoked. This requires extra work on behalf of the developer to coordinate the actions which may occur in separate threads, although the concept is simple and could be provided in some utility library. However, in the case where the caller may issue more than one call before completing the first, some mechanism (such as call numbering) is required to differentiate the call return values. This leads to second obstacle concerning the synchronising of parallel data flows.

As there is no system-wide clock or timing control, an element which expects to receive two (or more) separate imports together needs to take special actions to ensure that the data arriving at both imports are the expected pair. For example, consider an element that imports two integers (say, A and B), adds them and then exports the result.

A simple case that is complicated by the fact that A and B will arrive at different times with no guarantee of order. A might receive two integers before the first B arrives (perhaps B has a longer dataflow pipeline). Trigger conditions (see section 6.2) can help with synchronising arrival times but the developer must also implement some sort of buffering scheme to support overlapping arrival times. To avoid this, related data should be paired together whenever possible.

Stateless Elements

The DPROJ adaptation process involves the dynamic substitution of functional element implementations which may differ in terms of structure, location or algorithm. When a substitution occurs, the current element is deactivated and then the substitute is activated; if the current element possessed some logical operational state, then it would be necessary to ensure that its substitute echoed this state.

The current implementation does not provide the mechanism for state transfer as it is considered more a middleware issue. The command to transfer state may be issued by the application monitor but the actual transfer must be performed by the middleware or performed by the elements themselves[35].

However, the transfer of state through any mechanism raises some issues for the adaptation framework, in particular:

Differing Implementations — The exact implementation of the substitute element may differ from that of the original. In this case, a simple 'snapshot' of the element's process state is not suitable but a more fundamental logical operational state transfer is required. Hence, automated state transfer is not possible but must be developer provided.

One to Many Mappings — The substitute element may differ not only in implementation but also in structure. This is the case where a Basic element may be replaced by a Container element that provides the same functionality but through several sub-elements operating concurrently. In this situation, there is no one to one, like for like correspondence between the elements and the state of the deactivated element must be somehow spread to all sub-elements of the substitute Container. While this should be possible with a well structured, formalised state transfer mechanism, it will be complex and require an agreed and understood state datatype for each functional element type.

One-Off Costs — As the application monitor performs adaptation based on steady state operational costs, the *one-off* cost of a state transfer is difficult to integrate into the model. Instead the cost must be amortised over some time period.

The issues raised by the problem of state mean that there is no capacity for state transfer in the current implementation. It may be added in the future with substantial middleware and application modification; the changes required to the application monitor should be less. With the current implementation, application developers should try to minimise the number of elements holding or requiring state. Any need for state may be tackled in two ways:

1. Restrict any element with state to have only one implementation — it will never be deactivated and there will be no need to transfer state.

2. Ensure that any state required by an element is tied to the data that it imports.

Latency

One of the most troublesome aspects of distributed applications is the aspect of latency. The latency introduced by cross host communications is often several orders of magnitude greater than that for communications between processes on the one host and greater again than that of communication between components within the same process. As DPROJ achieves adaptation partially through the exploitation and manipulation of component locality, it is possible for the communications "break" to occur at almost any place in the application, depending on conditions. For some latency sensitive applications, such as control systems or interactive applications, control of end to end latency is critical to achieving performance goals.

In designing such applications, it is important to view latency as a performance indicator and not as a resource. While some resource management systems consider it to be a monitorable resource (*i.e.* Odyssey), DPROJ takes the view that high latency should be reported as poor performance and that to explicitly treat it as a resource is incorrect in this framework.

However, this approach also raises the problem of attribution of blame for high latency. Communications latency occurs between two functional elements whereas the DPROJ application monitor models the application by associating resource use and performance with individual elements. While use of communications resource can be attributed to both sender and receiver, it is not possible for two elements as a pair to report high latency or poor performance. This raises two possible methods for feeding back high latency alerts into the model:

1. Receiver reports poor performance: in this approach, the element receiving the data notices that the data is arriving late and indicates poor performance; eventually its performance appears so bad that the monitor transfers operation to a different implementation — hopefully this will be closer to the data source.

   Unfortunately this simple approach must assume that the receiver can measure the latency but at the same time cannot guarantee a correct result: there may be no alternative implementation for the receiver, it may be better to move the sender closer to the receiver and there is no method to appropriate blame for high latency in long chain of functional elements.

2. High level performance reporting: as latency is an end to end performance indicator it is more appropriate that it is reflected as encompassing the complete chain. The notion of such high level reporting is described in section 6.7.5 below.

In the current implementation, option 1 offers basic support for latency control in a limited number of applications.

6.7.5 Feature Wish List

Already it is clear that the current implementation of DPROJ lacks many features which would be necessary if this framework were to be used in an end user environment. In addition to cosmetic details and usability issues, the following lists some functional advances to increase the flexibility and efficency of the system:

Weighting of Container sub-elements — applying weights to the cost-benefit functions reported by sub-elements of a Container would allow accurate control of how the overall Container cost-benefit function is calculated from those of its sub-elements. A weighted aggregation system allows two key features:

1. accurate reflection on the relative contributions of the individual sub-elements to the overall performance of the Container. The developer may set these at compile time and also adjust them dynamically to reflect the changing nature of the application's operation.
2. prioritisation amongst the sub-elements. Weighting the perceived performance benefits of each sub-element will have the effect of reducing their resource allocation during reconfiguration correspondingly — those with a greater weighting contribute more to the overall performance of the Container and hence the reconfiguration process will naturally lend itself to allocating greater resources in these areas. This is especially important for allowing prioritisation between concurrently executing applications.

Increased reflection of the element hierarchy — detailed exposure of the application model would allow an application greater capabilities of inspecting and even modifying its own structure. Currently, the application structure is very fixed: while alternative element implementations may be added into a Set element dynamically, the actual structure of the application cannot easily be altered by the application.

For example, a video monitoring application may wish to have an arbitrary number of video streams being displayed simultaneously. The current framework allows modification of the implementation of streaming elements but does not allow dynamic addition or deletion of additional streams. The video application is unable to show additional streams as it is structured to show only one. While it is possible to add additional video applications, the single application cannot add an additional stream. This could be easily overcome if the application could alter its own structure and model dynamically just as the addition of a new application alters the top level system model.

Higher level performance reporting — a more flexible notion of performance would involve reporting application performance for the entire application (or large part of) rather than at the individual Basic element level. Reporting at Basic element level can be useful for isolating the cause of isolated poor performance but it is not always necessarily true that the performance of the whole derives from the sum of the parts: individual Basic elements may be reporting acceptable performance but the actual performance of the Container (or application) as a whole may not concur.

It would be useful to be able to report performance at a Container level and then use this to scale the sum of the sub-element contributions. This would enable an accurate depiction of performance but still retain the breakdown of the sub-element contributions. This is perhaps the only correct way to work latency considerations into cost-benefit analysis.

Improved element handling — a more sophisticated system might provide a more dynamic approach where elements are instantiated and/or migrated when needed as opposed to the current system employing redundant instances of the same implementation code. Code pools could approach such a level of dynamism through simple extensions incorporating more flexible loaders. For example, a loader could fetch element implementation libraries from a remote *implementation repository* server (perhaps over HTTP) which would deliver an appropriate binary for that host machine. Alternative implementations might use a cross-platform executable format such as Java byte codes to simplify this procedure and increase security at the cost of execution performance.

Similarly, a component level migration service such as CORBA's LifeCycle service could allow element instances to *hop* from host to host. A truly migratory system would be able to transfer the state of an element along with its functionality. Such an advance would remove the 'stateless' constraint from application development.

These advances are not yet implemented due to the small scale of the current application set but would integrate into the existing framework with minimal disruption. For most of these, the major engineering task would be in designing the programming and user interfaces in such a manner that the functionality is exposed in a simple and useful manner.

6.8 Summary

This chapter has outlined the details of the experimental DProj implementation which follows the design given in chapter 5. Building on a CORBA middleware platform, the implementation represents a minimal set of tools and services to create and deploy adaptations with automatic adaptation. The implementation comprises:

- The Element Composition language and development tools
- Application runtime support and dynamic implementation loader
- Application modeling adaptation services in the form of the Application Monitor
- Basic peripheral services implemented in a minimal form; these include the resource manager and policy unit with user interfaces.

The simplistic nature of the peripheral services means that the usability and scope of operation of the system as it stands is limited, but remain sufficient to run simple applications in a controlled environment for testing purposes. Analysis shows that the adaptation grows linearly with application complexity (actually, the number of elements, *i.e.* adaptation granularity) while the rate of adaptation can be set depending on context to balance resource overheads and responsiveness.

The next chapter describes some applications developed and deployed using the DPROJ framework for automatic adaptation.

DProj Operation: Tests and Applications

7

This chapter describes the use of the DPROJ application framework of Chapter 5 to construct and run adaptive mobile applications. This involves two stages: firstly, the evaluation of the system in adapting some simple test applications and secondly the design and implementation of a larger application.

The test applications simulate media streaming from server to client device and execute in a controlled test environment which includes *trace modulation* of resource availability and costs and also enables intensive instrumentation of the running application. The structures of the test applications are simple, allowing for easier understanding of how the system is performing through visualisation of the application configuration over time. The tests characterise the behaviour of an application according to correctness and responsiveness when subjected to variations in resource availability according to set profiles. Analysis of the response highlights the balance between responsiveness and stability, leading to resource filters which implement hysteresis in the resource manager.

Finally the chapter concludes by presenting a mobile application designed and implemented using DPROJ. MobileTRIP is an application that provides real-time visual augmentation of a users environment using visual overlays. Its operation involves both heavy image processing and large data traffic.

7.1 EVALUATION CRITERIA

Evaluation of an adaptive framework focuses on two key criteria. Firstly, evaluation of the correctness of an adaptation decision checks that the adaptation mechanism is making meaningful, sensible decisions to make best use of resources according to the adaptation policy laid down; secondly, evaluation of the responsiveness (agility) of the adaptation mechanism to reconfigure an application upon a change in resource conditions gives a measure of the usefulness of the adaptation mechanism for different environmental conditions.

7.1.1 Correctness

The adaptation process is complex and multi-faceted for even the simplest of useful applications. Determining correctness of adaptation for such an application is difficult and perhaps even subjective. A more tractable and general approach instead is to evaluate the correctness of adaptation applied to smaller applications of limited reconfiguration potential. Using simple test applications which each embody only a single primitive dimension of variability, correctness can be evaluated for simple conditions. Further examples can test combinations of these and assert their correctness. If the test cases cover all primitive dimensions and their combinations, then correctness of the test cases can be extrapolated to assert correctness of any application constructed using such primitives. Evaluation, hence focuses on three primitive structural cases:

Alternate sub-elements — a trivial case in which several discrete Basic elements in a Set providing alternate implementations of the same functionality at different cost-performance points. The cost-benefit function for each sub-element is constant and the combined cost-benefit function for the case is piecewise constant. The adaptation policy acts upon the function and selects a single performance point. This is propagated to the sub-elements resulting in selection and activation of the lowest cost implementation at that point. Evaluation should confirm that the lowest cost sub-element is indeed activated; in the case of more than one sub-element offering different performances at the same cost, the sub-element with greater performance should take precedence.

Serial concurrent elements — in which an application is a Container composed of a single non-branching chain of processing elements. Each of these elements may be a Set element containing alternate sub-elements as described above. This case is represented by the SerialTest application described below.

Parallel concurrent elements — in which an application is a Container composed of several processing elements operating concurrently in parallel (no data interdependencies). Each of these elements may be a Set element containing alternate sub-elements as described above. This case is represented by the ParallelTest application described below.

These test cases are evaluated in *open loop* mode with reference to the resource feedback loop. In this mode, application usage of resources does not affect the resource availability or costing indicators, isolating the behaviour of the adaptation model from the behaviour of the resource handlers. After correct adaptation has been confirmed in open loop mode, the loop may be *closed* to evaluate the system operation under various resource pricing models employed by the individual resource handlers.

7.1.2 Responsiveness

The responsiveness of a system is a measure of its ability to react to change in its condition or environment. Responsiveness is evaluated through measurement of an application's reaction to changes in condition using standard control system approaches.

However, responsiveness requires not only fast reconfiguration of an application in the light of new resource conditions but an ability to reconcile long term configuration and performance goals with what may be transient changes in resource conditions.

The rapidly changing environment of the system often means that a reconfiguration will only just be executed before the application must be reanalysed and a new decision arrived at. Reactiveness encompasses this need to adapt currently executing plans to the new situation. This readiness to alter existing configurations coupled with a large response time can also be the cause of instability in the system that may then lead to oscillations: the feedback nature of the system may lead to successive iterations of the application monitor reconfiguration process decreasing and increasing resource costs and thence reconfiguring the application. In some case, an application may get caught in these oscillations, activating and deactivating the same application elements at alternate iterations of adaptation process.

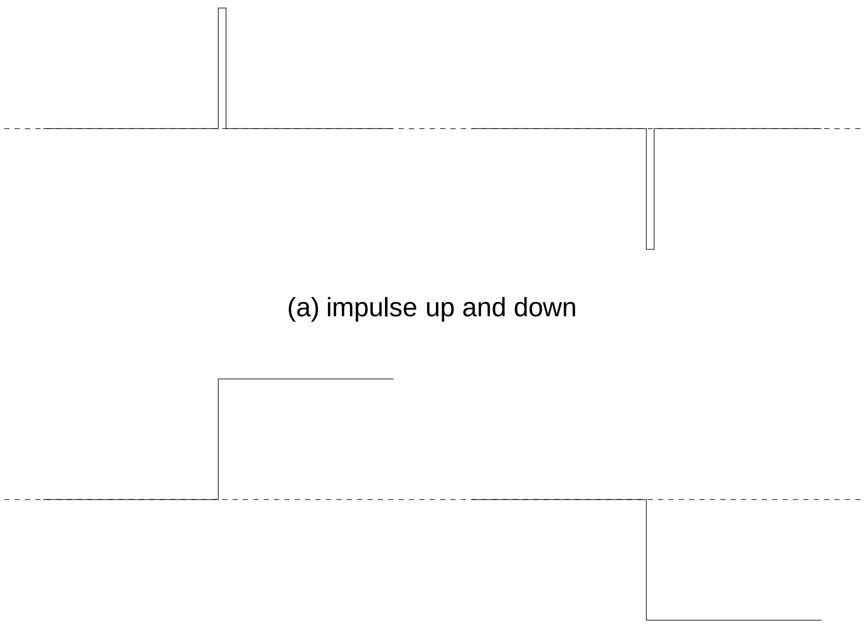
In order to be useful, any reconfiguration decision should aim to save more resources than it consumes. Rapid reconfiguration of an application may be wasteful of resources and so some form of hysteresis will be required to smooth these transitions.

Evaluation of the framework's adaptation responsiveness will both determine the limits on responsiveness imposed by the DPROJ framework structure and also evaluate the utility of the adaptation process to adapt applications to large variations in resource availability. Between them, these two criteria should answer the question: does pervasive adaptation and resource management provide better adaptation decisions than application-specific approaches?

As the adaptation process incorporates both a unified resource metric and a unified performance metric, it can be compared to a single input, single output control system. It can be evaluated in open loop mode to evaluate its reactiveness independent of resource management concerns and then in closed loop to evaluate its overall response. Evaluation will consider both the response to changes in resources and steady state operation of the application in terms of its configuration and performance metric. The resource availability or cost input can be modulated to control to the adaptation process; profiles to be considered include *step response* and *impulse response* in both step/impulse up and down format.

An impulse response gives the response of the system to a momentary spike or dip in input as shown in figure 7.1(a). Theoretically, an impulse has infinite amplitude while being infinitesimally narrow; it has the same constant steady state behaviour both before and after the impulse. A practical implementation must hold this impulse for a minimum of one sample period; additionally, the impulse will not be infinite in amplitude but will be very large in comparison to the steady state values. The response to an impulse gives an indication of the tolerance of a system with regards to momentary changes in conditions in an otherwise steady environment.

A step response gives the response of the system to a rise or fall in input as shown in figure 7.1(b). Theoretically, a step is an instantaneous change in amplitude; it has constant steady state value both before and after the impulse. A practical implementation will perform this change between input samples to approximate an instantaneous step. The response to a step gives an indiction of how fast the system can respond to a genuine



(a) impulse up and down

(b) step up and down

Figure 7.1: Resource cost input profiles.

change in conditions. This will be characterised by: a) how fast the system can acquire its optimal configuration for the new condition, and b) any overshoot in approaching the new configuration.

7.2 Evaluation Environment

A controlled execution environment is required to provide precise experimental control and reproducible conditions. A real wireless mobile environment will not be able to reproduce arbitrary conditions for repeatable testing. Instead, a simulated mobile environment operating in a fixed wired network allows greater control over resource availability through the process of *trace modulation*[65, 10].

Trace modulation involves replacing a resource with an equivalent emulation whose availability is modulated by a predefined trace. The trace provides a time-varying profile of critical performance or availability parameters for a particular resource which may be applied, through emulation of the resource, many times over. Noble applies trace modulation to communications resources but it may equally well be applied to other resources in either open or closed loop modes of operation.

For the purposes of evaluation, all resources are held to be at constant availability and cost excepting only that of network availability which is modulated according to predefined traces. Applications are configured to run in two host mode (single client and single server) with the controlled resource being the communications medium between the two. The medium is treated as a half-duplex carrier in which the communications capacity is shared between the two-way traffic and hence the cost of traffic flowing either way is calculated on the same basis. This is representative of a shared medium communications channel such as Ethernet or peer to peer waveLAN but not full-duplex connections such as GSM dial-up.

7.2.1 Test Configuration

The tests are conducted in a fixed network environment between two hosts running the Linux operating system. Host Tetleys acts as the server while host Rigwelter acts as the client device; they are connected by a switched 100Mbit Ethernet. Their specifications are as follows:

	Tetleys (server)	**Rigwelter (client)**
OS	Linux 2.2.14	Linux 2.2.5
CPU	dual Intel Pentium-III @ 500MHz	dual Intel Pentium Pro @ 200MHz
Memory	256MB	128MB
Comms	100Mbit ethernet	100Mbit ethernet

The hosts are, currently, regarded as fairly recent and high specification machines; clearly the client host and the communications channel are not representative of a mobile system. It is instructive to observe the adaptation in such a free environment where the only constraints are those imposed artificially. The application performance can be observed to remain within some theoretical resource limit. This theoretical limit is used by a customised resource manager and policy unit to determine resource costs.

The server hosts the server code pool while the client hosts the client code pool and also the application monitor, resource manager and policy unit. The application monitor operates at 1Hz, performing an update of the entire application model tree and issuing reconfiguration instructions every second. The CORBA naming service is located on a separate server but as this is only used to locate application and system components at startup its location has no impact on the on-going operation of the application.

7.2.2 Customised Resource Manager and Policy Unit

For the following tests, the resource manager and policy unit described in chapter 6 are replaced by customised versions which allow:

1. Explicit manipulation of resource availability/cost values which may be modulated by some given trace.

2. Instrumentation to extract runtime configuration data from the application model, including resource costs, element activation, element locality and the cost-benefit function of an application or any Container element within it.

Trace Modulation of Resources

The new policy unit can manipulate the resource availability and costs of the resource manager according to set traces provided as Python code. These traces may be open loop, setting the cost irrespective of resource use, or may be closed loop, calculating the resource cost based on current usage and theoretical availability.

Current profiles fix the cost of client and server CPU and battery, manipulating the communications cost as the independent variable. As adaptation is based on minimising total cost for performance, it is the ratio of costs that is important and hence adjusting a single variable is sufficient.[1] This follows the trend of other adaptive systems in focusing on the communications bottleneck.

For these tests, two modulation schemes are required: open loop and closed loop. In open loop mode, trace modulation is applied to the communications cost which is passed to the application model on each update.

In closed loop mode, trace modulation is applied to the communications availability; communications cost may then be calculated by taking into account the link cost and the communications traffic as described in section 5.4.3. In this case, the traffic is modelled as having a number of channels which may be bonded up to some limit. The "charge" is per channel and hence the communications capacity is effectively purchased in fixed quanta. The costing scheme for this should try to encourage use of any open channels but discourage opening a channel for only a little traffic. At the same time, it should try to ensure that the total traffic does not exceed the capacity provided by the maximum number of channels. Section 5.4.3 proposed a scheme where the total cost of the open channels is spread over all of the current users of communications but recommended that some buffer zones be implemented so that nearing full capacity of the currently open channels should raise costs to try to slow overflow into an additional channel.

The scheme used here is given in figure 7.2 with normalised costs and capacities for a system where there may be up to four bonded channels. In this scheme, the penalty cutoff (k) is defined as 15% and gives the lower and upper limits of unity cost behaviour within a channel. If a channel's usage is below 15% or above 85% then the cost is artificially inflated to penalise operation in this vicinity, increasing linearly to reach a maximum where each communications unit is effectively being charged the cost of the entire channel. This has the same form as the scheme in section 5.4.3 with a 15% left shift but differs in that the cost penalty peaks are the same magnitude no matter which channel boundary is being approached, the penalties are linear instead of exponential and the cost function has no discontinuities against traffic. The first two of these changes make the costing fairer and more manageable (through a more compressed scaling of costs) while the removal of discontinuities better supports the incremental approach of the adaptation process.

[1] For example, in the Serial Test, it is the ratio between the CPU cost *difference* (between client and server) and the communications cost that determines application configuration.

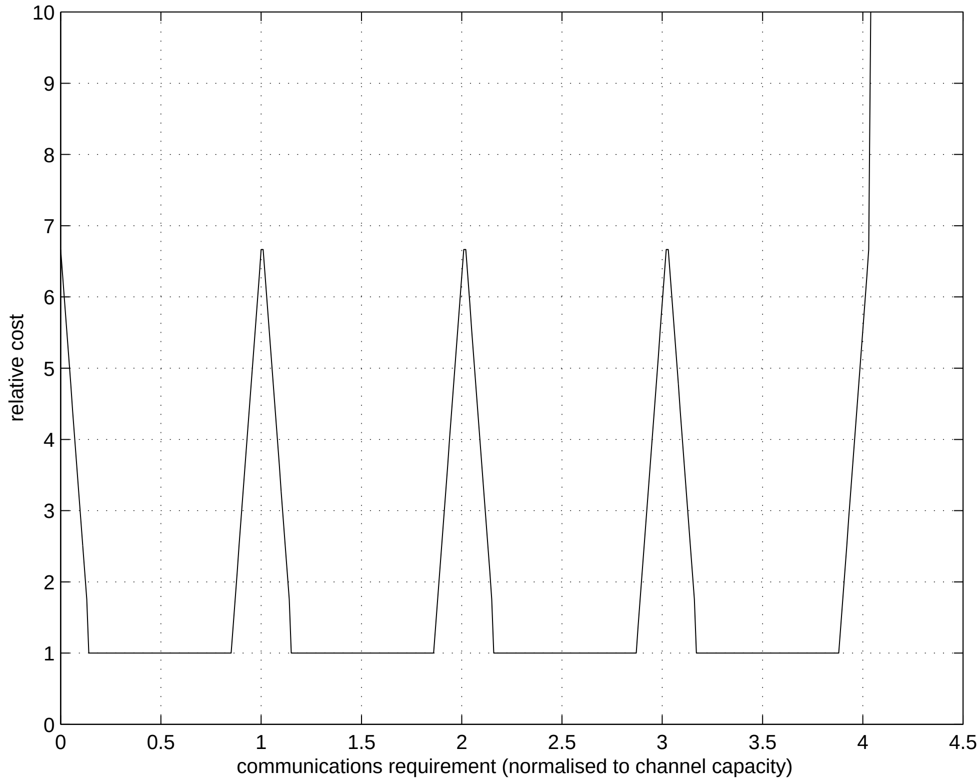


Figure 7.2: Example traffic costing profile.

Visualisation of Application Configuration

The policy unit can extract runtime instrumentation data from the application monitor, including element activation status, gateway traffic, CPU requirements and cost-benefit functions. This allows inspection of the performance of the adaptation mechanism in several ways. Currently, it will only extract individual element information at the first level in the hierarchy. This is sufficient for the simple Serial Test and Parallel Test applications.

For simple applications, the extracted information can be post-processed to visualise the state of the application over time. This can be in two forms:

1. Element locality pattern — a two dimensional raster showing active element location over time such as given in figure 7.4. This shows the first layer Set or Basic elements on the Y-axis and time on the X-axis. The location of each element at any point in time is represented by the shade of the point: white for server side and grey for client side. Superimposed on the raster is a line plot of the performance set-point and the operating cost at that point.

2. Element cost pattern — a three dimensional plot showing attribution of costs to individual elements (at this level in the hierarchy) over time as given in figure 7.5. Superimposed is a line plot of the total operating cost.

As the elements are arranged in a single dimension, practical visualisation is limited to simple applications that have a linear or near linear structure. This is suitable for the test applications described below but more complex applications may produce too much data to visualise.

7.3 Serial Test

Media streaming is a common test case for adaptive mobile systems. The Serial Test application simulates an application that streams video or audio data from a remote host to be presented on the mobile device. Media streaming is a good test case especially for Quality of Service demonstrations where the quality (*i.e.* fidelity) of a media stream can be varied to best fit conditions. This example demonstrates how the locality of processing for media stream decoding can be varied to control resource usage.

The Serial Test application is designed to measure adaptation performance for an application exhibiting linearly structured dataflow. This application is a chain of dummy data processing elements, each of which increases the magnitude of the dataflow along the chain. The two ends of the chain are anchored to the server and the client to enforce a split across two hosts. The Serial Test application can be thought of as a basic model of any media streaming type application; the data originates in compressed format and is decompressed through various stages resulting in increasing dataflow traffic before final presentation at the user's mobile terminal. The exact location of the decompression is unimportant but the origin of the data and the final presentation point of the data are.

7.3.1 Aim

The basic aim of this test is to check the dynamic characteristics of the adaptation process under variation in resource cost (availability) using a simple linear dataflow case. The ratios between server CPU cost (C_s), client CPU cost (C_c) and communications cost (C_n) are varied to control the balance of resource availability. Initially, $C_s = 10$, $C_c = 100$ and $0 <= C_n <= 1000$ is the independent variable. The application affords no variability in performance so adaptation focuses on cost minimisation.

7.3.2 Description

The N-Block SerialTest application comprises a TestSrc data source and a TestSink data sink linked via N TestBlock data processing elements. The table below shows the basic characteristics of each element type. The TestSrc element generates t_0-byte data frames at 5 frames per second. Each TestBlock element exports a data frame twice the size of the one it imports. The TestSink element receive the final frame and discards it. The TestSrc is constrained to operate solely on the server while the TestSink is constrained to the client. Each TestBlock may operate on either the server or the client. The location of each TestBlock is unconstrained by the location of any other. The CPU requirements of each element type is the same at 100TIPS.

		traffic		locality	
type	CPU	in	out	client	server
Source	100	0	$t_0 = 256$B	–	•
Block$_n$	100	t_n	$t_{n+1} = 2 \times t_n$	•	•
Sink	100	t_N	0	•	–

Table 7.1: Serial Test component characteristics.

For the purposes of these tests, a 10-Block SerialTest application was used with $t_0 = 256$bytes at B_0 up to $t_n = 256$kB at breakpoint B_{10}. Operation of the application is enforced across two hosts; ideally, the application will only be partitioned at one point. The application may be partitioned at any of the breakpoints B_0 to B_{10}. Hence, the total cost function $C(b)$ for operation of the application may be devised in terms of the breakpoint B_b for $0 <= b <= 10$, CPU requirements per element $T = 10$ and the cost parameters C_s, C_c given previously:

$$
\begin{aligned}
cost &= \text{server CPU} + \text{client CPU} + \text{network traffic} && (7.1)\\
C(b) &= C_s T(b+1) + C_c T(11-b) + 5 \times \frac{t_0}{1024} C_n 2^b && (7.2)\\
&= 11100 - 900b + 1.25 C_n 2^b && (7.3)
\end{aligned}
$$

Figure 7.3 shows total cost versus breakpoint over a range of communications costs. The framework will partition the application to minimise the cost, hence when:

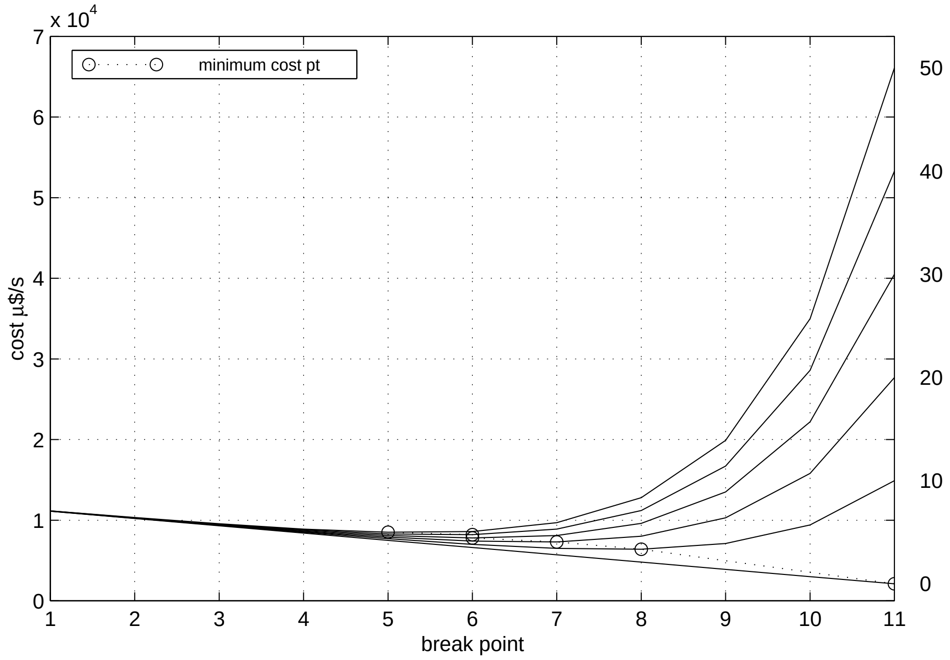


Figure 7.3: Serial Test cost versus breakpoint for different communications costs (from free to 50μ\$/kB).

$$\frac{dC(b)}{db} = -900 + 1.25C_n 2^b = 0 \tag{7.4}$$

Note that the data path of the application exhibits exponential increases in traffic (doubling at each block). While an application with a linearly increasing traffic will also gain adaptation from DPROJ, the results will be much less interesting: except at an exact balance point where the cost is the same no matter where the breakpoint is, open loop adaptation will always result in a break point at either extreme. A closed loop system will be similar but constrained by the communications limit.

7.3.3 Observations

Startup

In this case, the application begins partitioned between client and server with all blocks executing on the server except for the sink. This is a very high bandwidth configuration as the network traversal occurs after block9 with the 256kB frames. During execution, the application monitor calculates the application cost-benefit curve every second and performs any required reconfigurations.

The application gradually approaches its minimum cost steady state configuration. In this case, this takes approximately seven seconds. Some fluctuations occur due to variations in the measured communications use. The approach to the steady state is

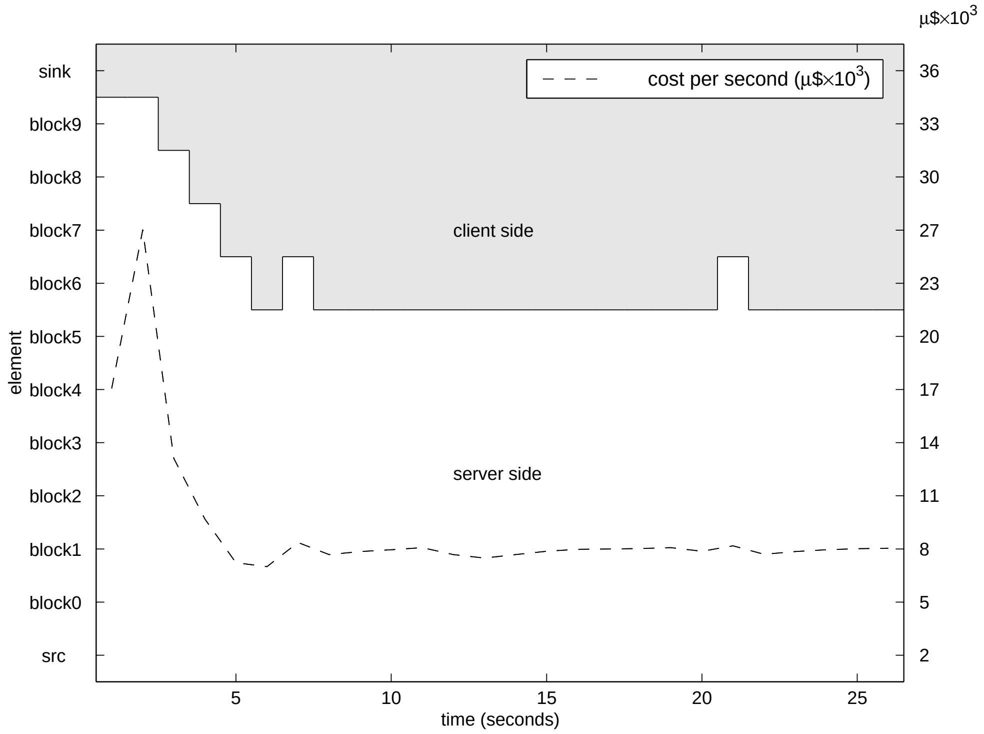


Figure 7.4: Serial Test locality pattern — startup.

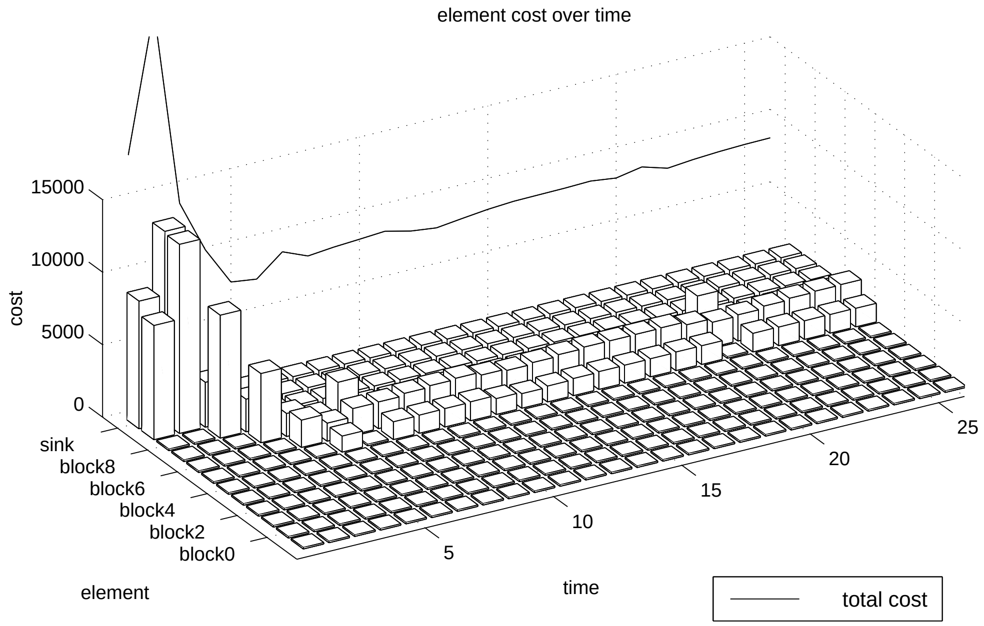


Figure 7.5: Serial Test cost distribution — startup.

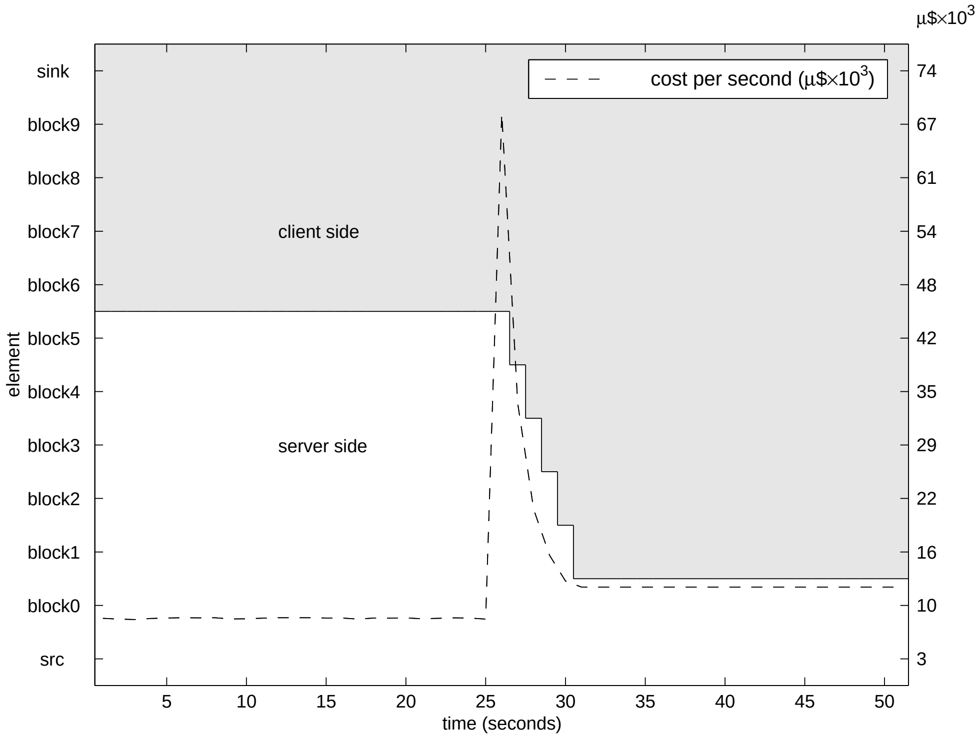


Figure 7.6: Serial Test open loop locality pattern — step up.

gradual as the framework is unable to estimate the possible cost-benefit curves of far removed configurations. This is due to the manner in which the application apportions cost of execution of a particular configuration; CPU and power costs are directly attributable to the element under execution, while communications costs are attributed to both the receiver and the sender. Hence, only an element that is directly communicating via the network incurs the communications costs which penalise it in the decision process.

Figure 7.4 shows the configuration at startup, operating with $C_n = 28$. Equation 7.4 predicts a break at point $b = 6$ for this cost.

Open Loop Response

Figures 7.6 and 7.7 show the Serial Test response to a step in resource cost, up and down respectively. For both tests, $C_s = 10$ and $C_c = 100$ as before; the step up steps C_n from 30 to 800 at time $t = 25$ (the value changing between sample at $t = 25$ and $t = 26$); step down steps C_n from 800 to 30 at the same point.

The step up response shows the same behaviour as for start up shown in figure 7.4. The peak cost is high but is quickly brought back down to reasonable levels. It reaches a steady state after 5 reconfiguration iterations. This shows that the incremental approach to the optimal configuration is proportional to the divergence from the current configuration. As the applications have a hierarchical structure and reconfiguration occurs at any level in the structure, rapid convergence on a stable configuration after a large

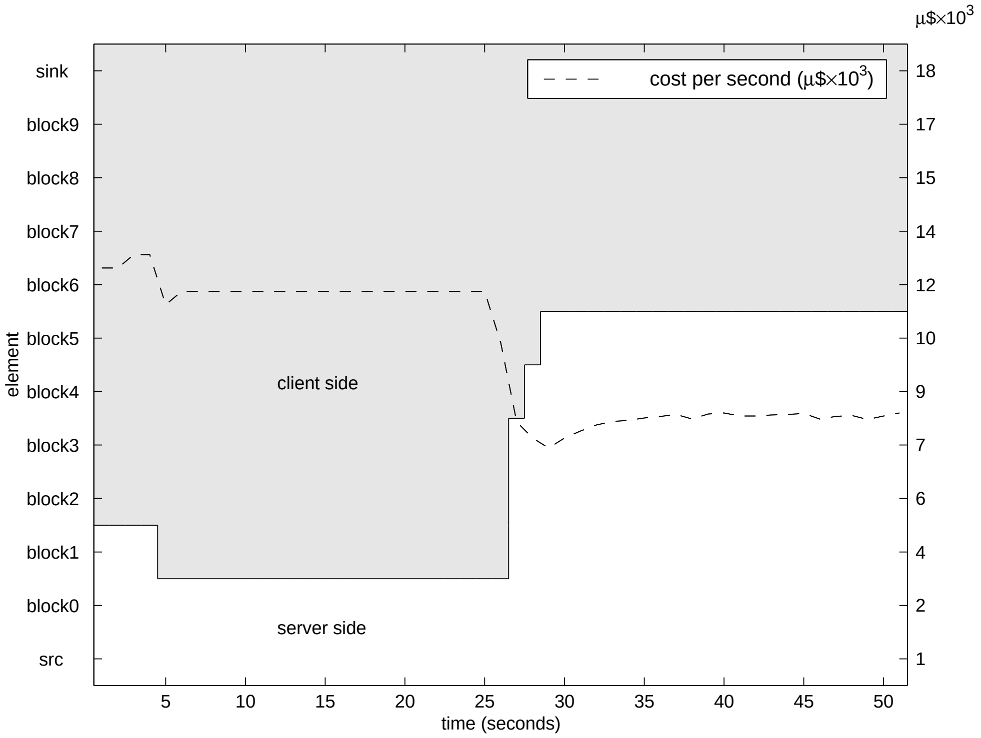


Figure 7.7: Serial Test open loop locality pattern — step down.

change in resource conditions is possible even in more complex applications.

The step down response differs slightly in profile from the step up but takes the same number of iterations to achieve its stable configuration. With the step down in communications cost, the dominant cost in the system becomes the CPU cost. Thus the incremental reconfiguration does not result in a staircase of gradual shifting but can achieve the reconfiguration with large shifts. This can be seen where the first reconfiguration involves moving all of blocks 1, 2 and 3 to the server simultaneously. Such large movement can occur because the system is responding to a reduced communications cost (*i.e.* a higher relative CPU cost) and hence CPU cost becomes the dominant factor in adaptation decisions. In this case, blocks 2 and 3 are moved as the relocation would result in an overall cost saving, even if its communicating neighbours did not relocate. For blocks 4 and 5, the CPU saving is not sufficient to warrant a move. However, once it is apparent that block 3 will be on the server, blocks 4 and 5 follow it incrementally.

Figures 7.8 and 7.9 show the Serial Test response to positive and negative impulses in resource cost. Note that impulse responses are instructive in determining the responsiveness of a system but are not representative of actual resource cost traces that may be expected in a real mobile environment — costs are a controlled value and are unlikely to fluctuate wildly. The resource manager and handlers which determine these costs are resource specific and can produce smoother costs which may us low pass filtering to reduce anomalous noise, but can still pass mode changes (*e.g.* changing communications network) as step changes. In these tests, CPU costs are maintained as before; the

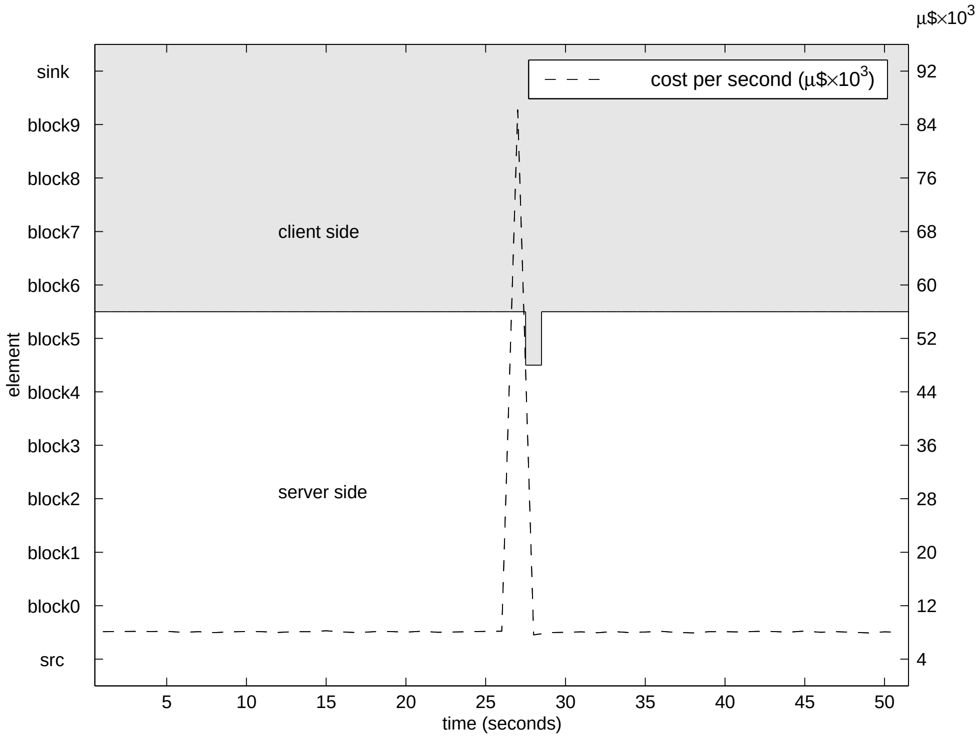


Figure 7.8: Serial Test open loop locality pattern — impulse up.

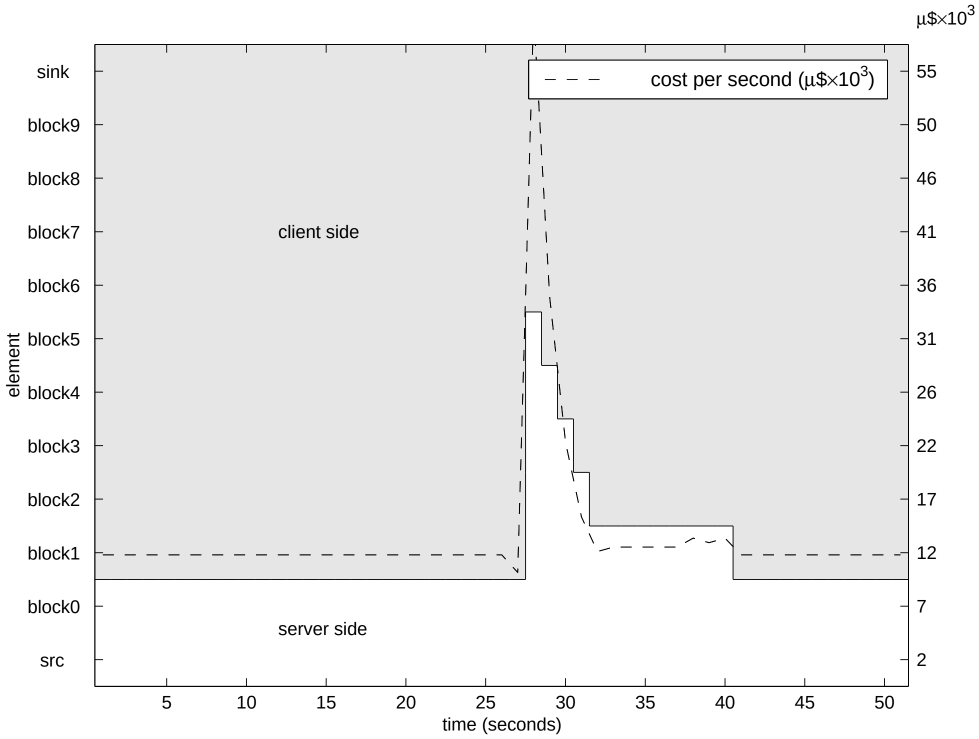


Figure 7.9: Serial Test open loop locality pattern — impulse down.

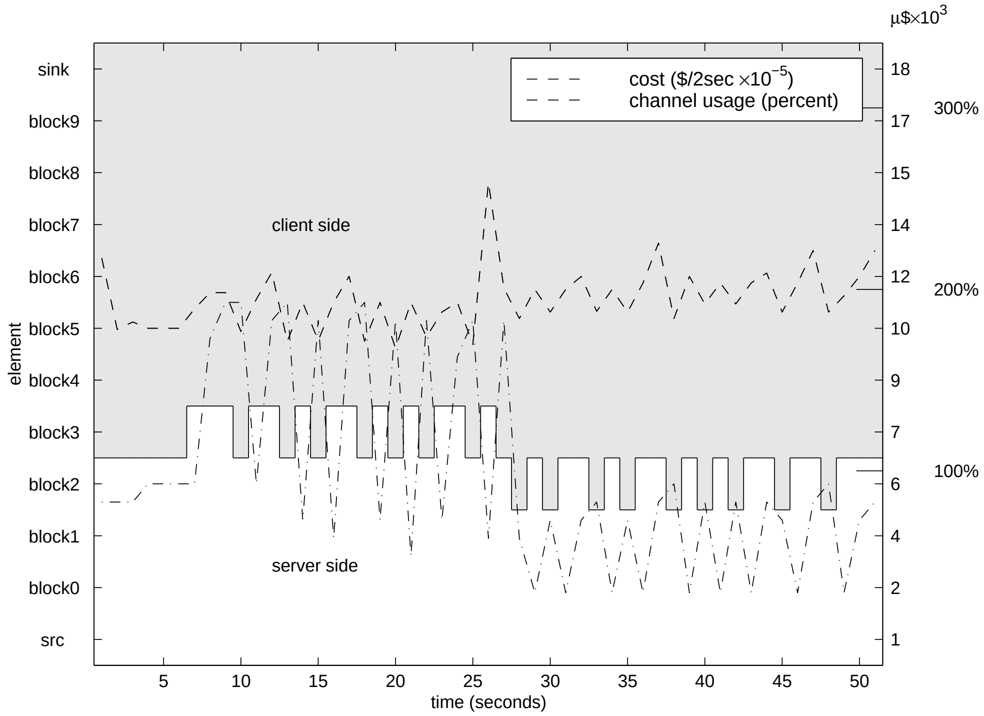


Figure 7.10: Serial Test closed loop locality pattern — step up.

impulse up spikes from 30 to 100,000 at time $t = 25$ for one sample while the impulse down drops from 800 to 10 also at $t = 25$ for one sample.

The impulse up is a momentary rise in communications cost. The adaptation response is initially the same as for a step up but this soon reverts to its original configuration once the spike has passed. The impulse down, on the other hand, provides a much sharper change in configuration in the same manner of the step down response, only the difference in communications cost is greater here and so too the effect. The configuration then returns to its original condition through several iterations.

Closed Loop Response

For the closed loop response, the communications usage is fed back into the resource manager to determine resource costing as described previously in section 7.2.2. Here, the resource manager implements a communications medium consisting of up to two bonded 10kB per second channels. This is equivalent to having two channels of exclusive availability (*i.e.* one or the other may be used but not both concurrently) where one channel has twice the capacity of the other. For simplicity, it is assumed that the channels can be opened and closed with zero start up time or cost.

Figures 7.10 and 7.11 show the step up and down responses of the Serial Test application under closed loop operation. In stepping up, the channel cost is stepped up from μ\$2000 per second per channel to μ\$4000 per second per channel at time $t = 25$; the step down is similar but steps from μ\$4000 down to μ\$2000 per second per channel. In

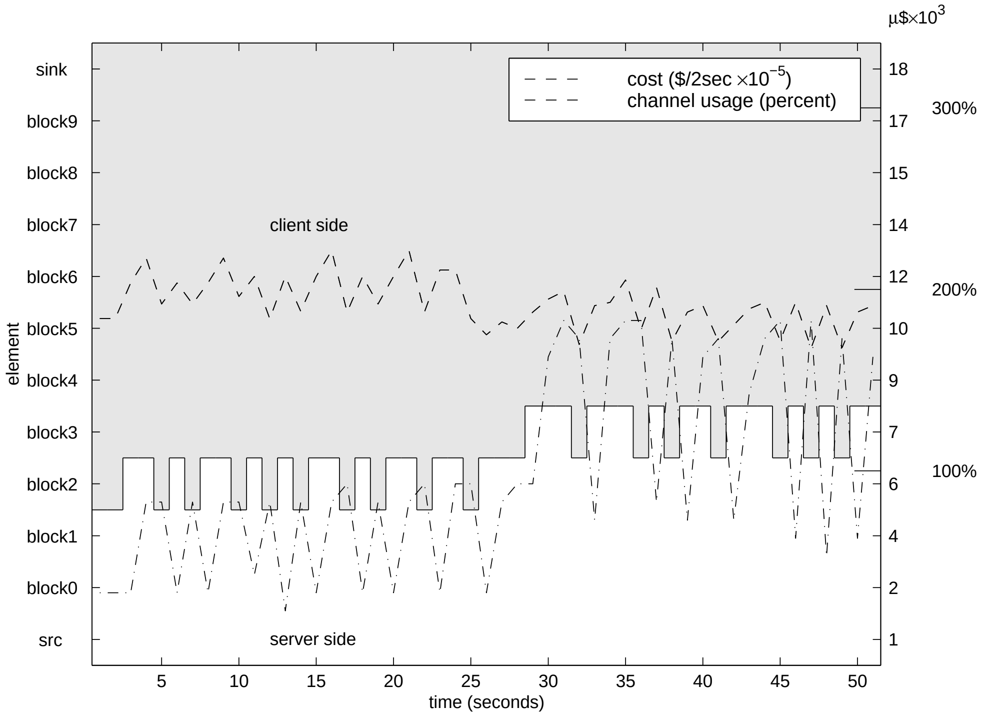


Figure 7.11: Serial Test closed loop locality pattern — step down.

addition to the locality and cost plots as before, these figures include communications usage normalised to single channel capacity (expressed as percent).

These responses show that the application operates using two channels when communications is cheaper and only a single channel when it is more expensive. The locality, cost and communications usage responses all have some oscillatory behaviour. This can be attributed to two factors:

1. The communications cost are artificially inflated when a channel is either less than 15% or greater than 85% used. If the communications usage is in these ranges, the costs will inflate markedly, resulting in sharp peaks in the total cost.

2. The above is compounded by the specific characteristics of the Serial Test application. The data rates between block 1 and block 2 is 5kBps, between blocks 2 and 3 it is 10kBps, and between blocks 4 and 5 it is 20kBps. Unfortunately, these are the only options available with this application (within the range of the communications available) and the 10kBps and 20kBps fall exactly on the channel limits and hence also where the communications cost is greatest. Further, small variation in the measured bandwidth means that the application may at some points attempt to exceed the total channel availability by a small amount, triggering an even greater cost.

The system reacts to the increased cost through reconfiguration to lower communications usage and hence oscillates.

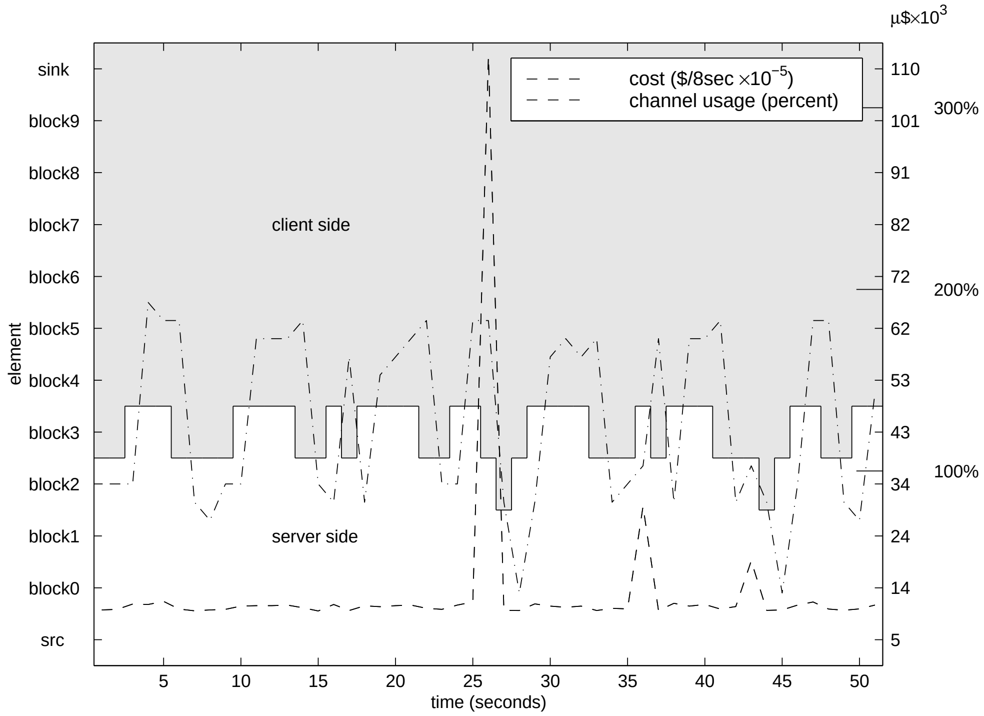


Figure 7.12: Serial Test closed loop locality pattern — impulse up.

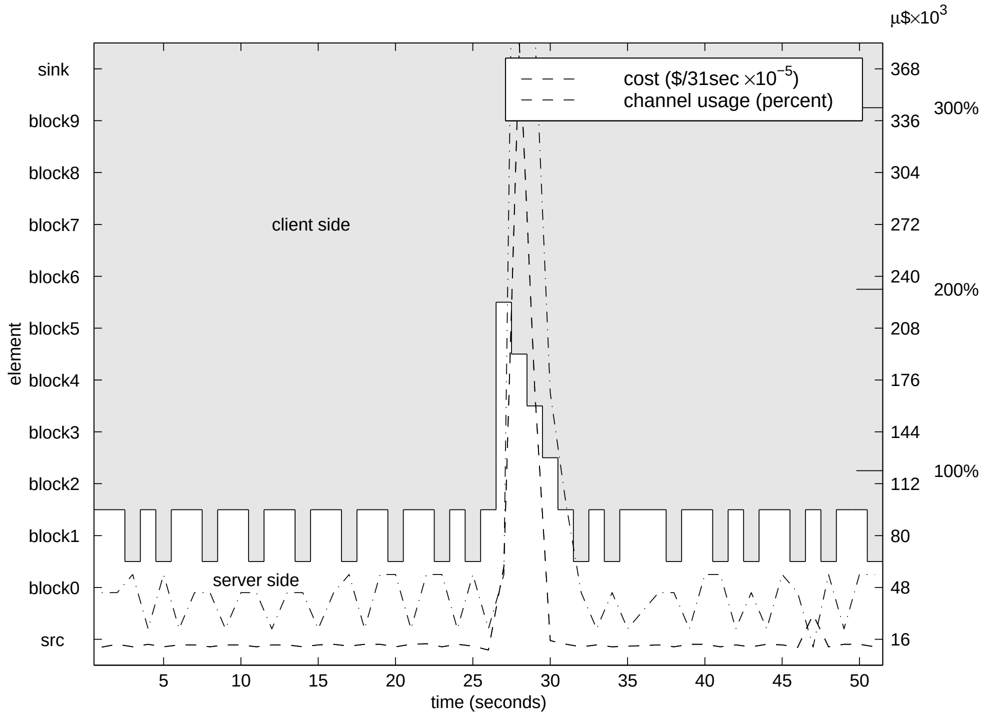


Figure 7.13: Serial Test closed loop locality pattern — impulse down.

Figures 7.12 and 7.13 give the response of the Serial Test configuration to impulse up and down variations in channel cost. The impulse up changes the cost from μ\$2000 per second per channel to μ\$100000 at time $t = 25$ for a period of one second (one sample). The impulse down pulses from μ\$8000 to μ\$100. The responses of the system show characteristics similar to those for the open loop impulse tests with the impulse up producing negligible effect while the impulse down causes a sudden shift which is correct in the same manner as a step back up. The main difference is the presence of some jitter (small oscillations) as also found for the closed loop operation in the step responses.

7.3.4 Summary

In summary, the system can respond to changes in resource cost quickly and achieve a suitable new configuration in a time period proportional to the divergence from the original configuration. An application with a deeper hierarchical structure may be able to reconfigure more rapidly than this in some cases.

The response of the system to changes in resource costs is incremental in one direction (a rise in communications cost or a drop in client CPU cost) but sudden in the other direction. This emphasises the need for resource managers which can control the rate of change of resource costs: these should low pass filter the monitored usage and availability to minimise the impulses passed on to the adaptation system but should still allow mode changes to be accurately reflected. Additionally, some form of hysteresis is required to minimise unwanted oscillations. This can take two forms to counter two forms of oscillation:

1. A refined costing scheme for bonded communications channels that reflects the currently open channels and the rate at which they may be opened or closed in its costs. This will counter oscillations between communications modes and hence also oscillations between communications costs.

2. Introduce overhead costs for switching between alternate implementations of an application element. This can be as simple as adding the overhead cost to the cost-benefit function of each inactive element.

7.4 Parallel Test

The DProj framework performs adaptation across multiple applications, aiming to minimise the total cost of the system while maximising total performance. In this test, two identical instances of the above media streaming test are run in parallel and are subjected to the same step up and down tests.

7.4.1 Aim

The aim of this test is to confirm the operation of the adaptation system in adapting parallel applications (equally well, parallel operations within a single application) in terms of resource cost minimisation and fairness. Observe the step response. In the open loop case, the two applications should be partitioned in the same way as for a single instance of the Serial Test application. In the closed loop case, the adaptation should partition the applications to keep the total communications traffic within the available bandwidth. The impulse response tests are not described here as the transient response is not substantially different.

7.4.2 Description

This test involves two instances of the 10-block Serial Test application operating in parallel, both streaming from a source on the server to a sink on the client. The application parameters remain the same at $C_s = 10$, $C_c = 100$ and with C_n being the independent variable.

7.4.3 Observations

Startup

At startup, the applications are launched and configured for operation between client and server with all application elements on the client except for the data source which is locked to the server. Figure 7.14 shows the locality summary of the application elements for 25 seconds from startup time with an open loop resource manager. DPROJ has divided both application instances in the same manner, minimising overall costs. Figure 7.15 shows the individual element costs for the same period. This shows that the main cost of the application operation is the communications cost. The difference in processing cost is evident in the step between elements executing on the client and those on the server.

Open Loop

Figures 7.16 and 7.17 show the Parallel Test response to a step in resource cost, up and down respectively. For both tests, $C_s = 10$ and $C_c = 100$ as before; the step up steps C_n from 30 to 800 at time $t = 25$ (the value changing between sample at $t = 25$ and $t = 26$); step down steps C_n from 800 to 30 at the same point. These values are the same as those used in the Serial Test open loop tests.

Because the resource prices are fixed, their availability is effectively unlimited, hence two Serial Tests operating in parallel suffer no greater bandwidth restrictions than a single instance. This is evident in the open loop step responses in which each instance of Serial Test shows the same response as for the single instance operating alone. Both instances adapt in the same way, indicating a fair distribution of resources.

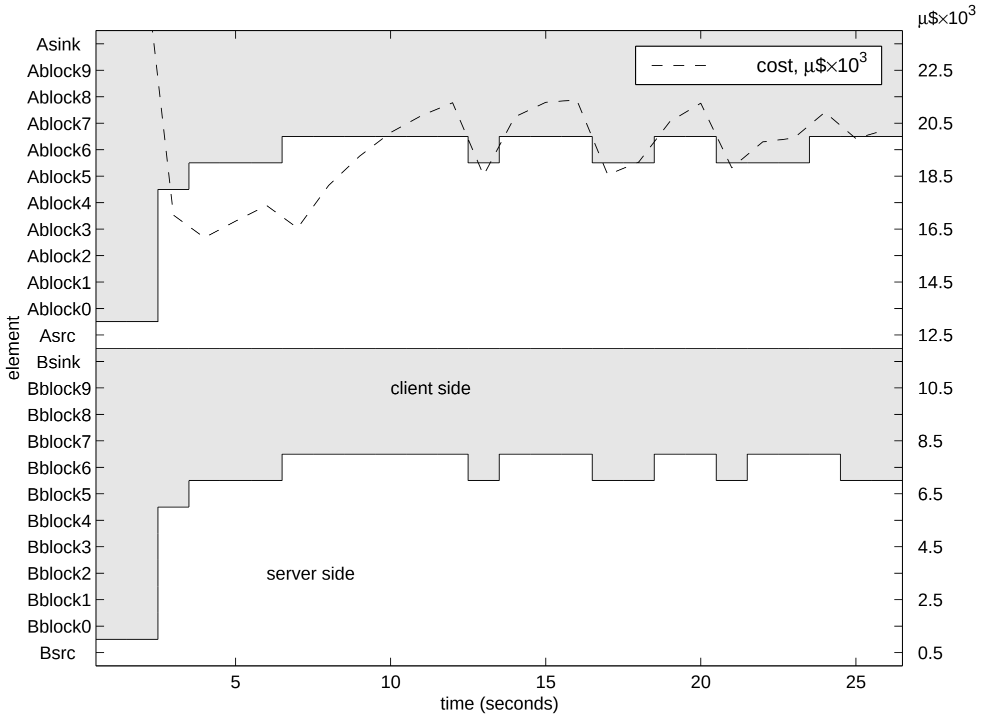


Figure 7.14: Parallel Test locality pattern.

Closed Loop

For the closed loop response, the resource manager uses a feedback loop to determine communications costs using the same scheme as for the Serial Test.

Figures 7.18 and 7.19 show the Parallel Test response for step up and down inputs respectively using a communications manager operating with 4 channels each of 10kBps capacity. This is exactly double that used for the Serial Test closed loop step responses and hence the responses here show the same characteristics.

However, if the available communications bandwidth is reduced to that available for the Serial Test step responses (2 channels at 10kBps each), then the Parallel Test should operate at a lower bandwidth configuration at both sides of the step. Focusing on the step up response, figure 7.20 shows this to be largely true but also features significant oscillations, particularly at the lower channel cost (pre-step). A simple counter to this oscillatory behaviour in the feedback system is to introduce a low pass filter into the loop. Figure 7.21 shows the effect using a first order low pass filter on the communications cost. The filter acts upon the calculated (C_n^c) and current (C_n^t) communications cost to determine the cost for the next time period (C_n^{t+1}):

$$C_n^{t+1} = \frac{1}{2}C_n^t + \frac{1}{2}C_n^c \tag{7.5}$$

The filtering removes the major pre-step oscillations although the minor post-step oscillations remain as they are produced more by the coincidence of the channel capacity and the traffic levels at block boundaries.

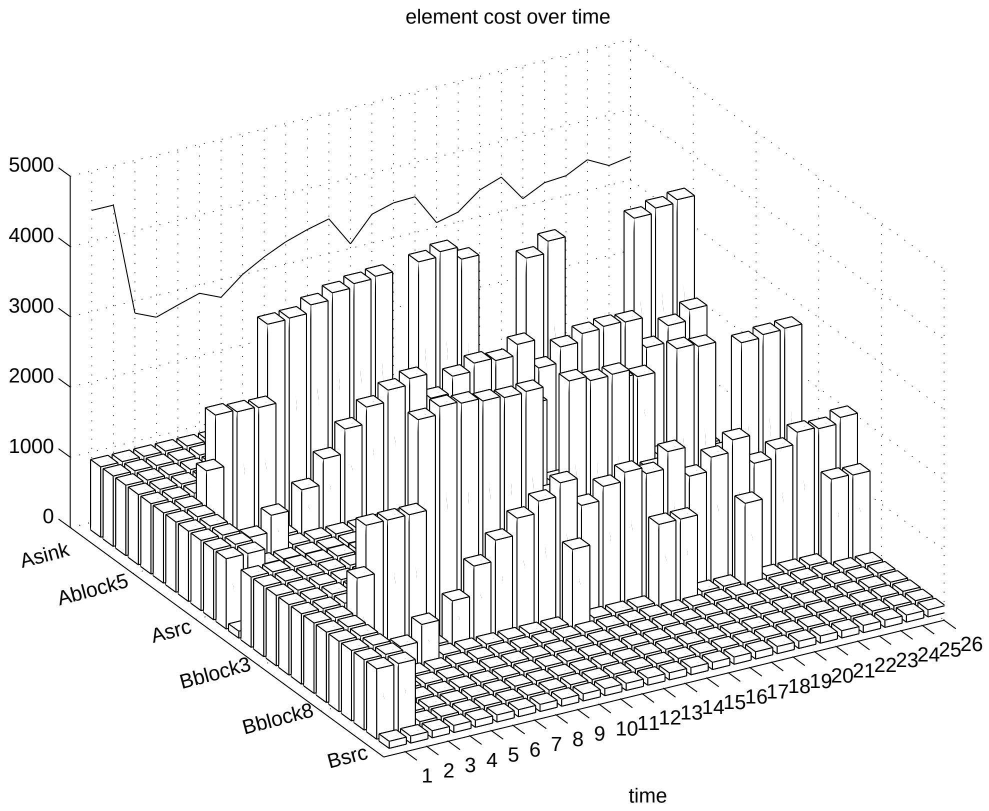


Figure 7.15: Parallel Test individual elemental costs.

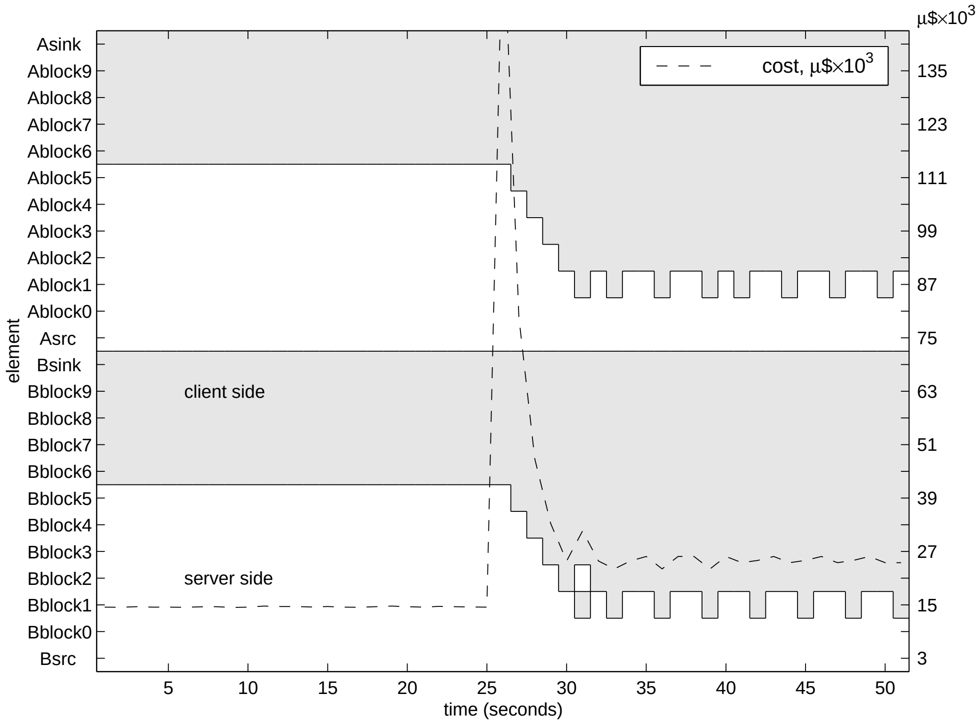


Figure 7.16: Parallel Test open loop locality pattern — step up.

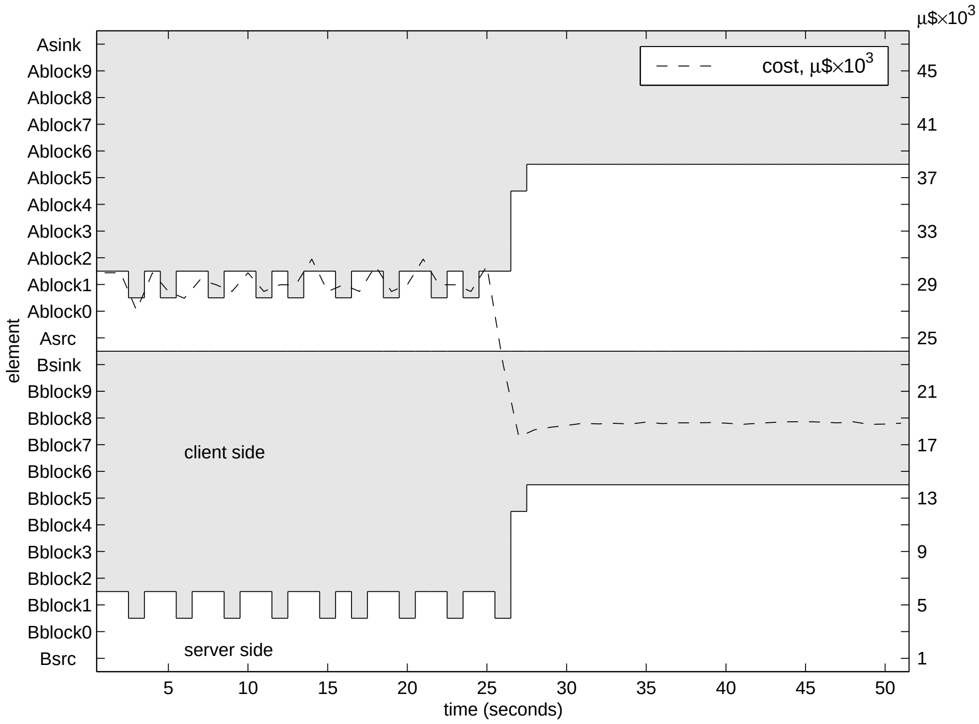


Figure 7.17: Parallel Test open loop locality pattern — step down.

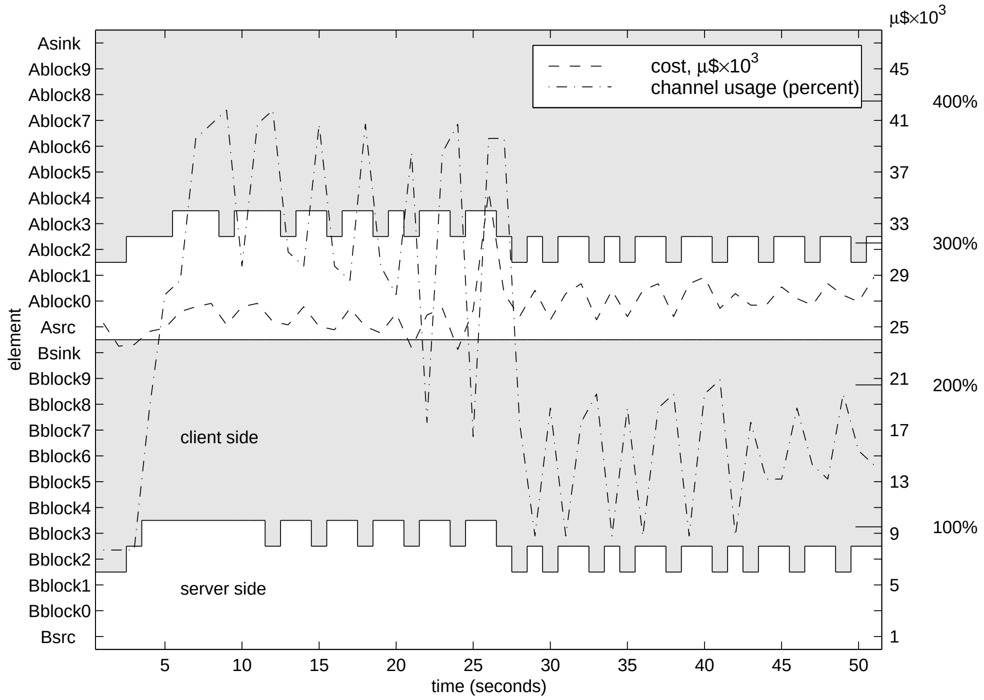


Figure 7.18: Parallel Test closed loop locality pattern — step up.

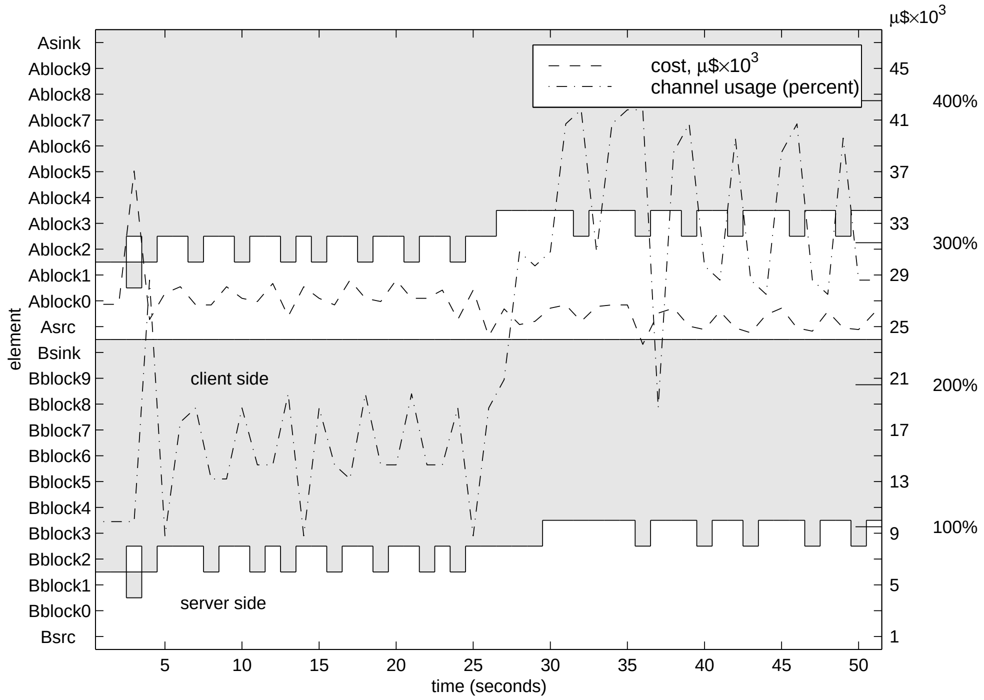


Figure 7.19: Parallel Test closed loop locality pattern — step down.

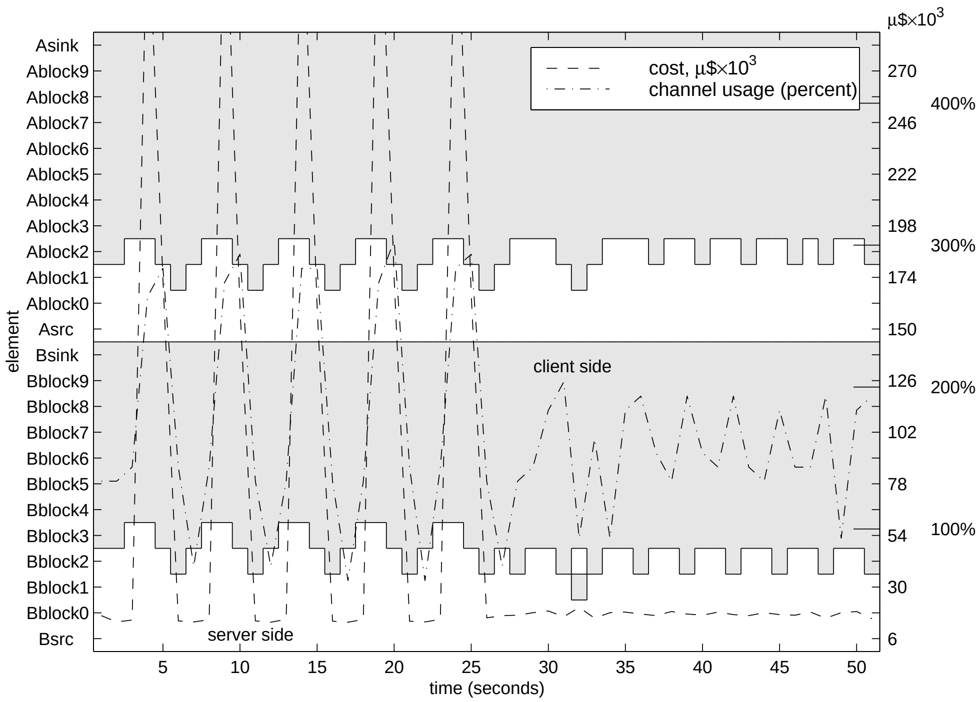


Figure 7.20: Parallel Test closed loop locality pattern — step down with reduced bandwidth.

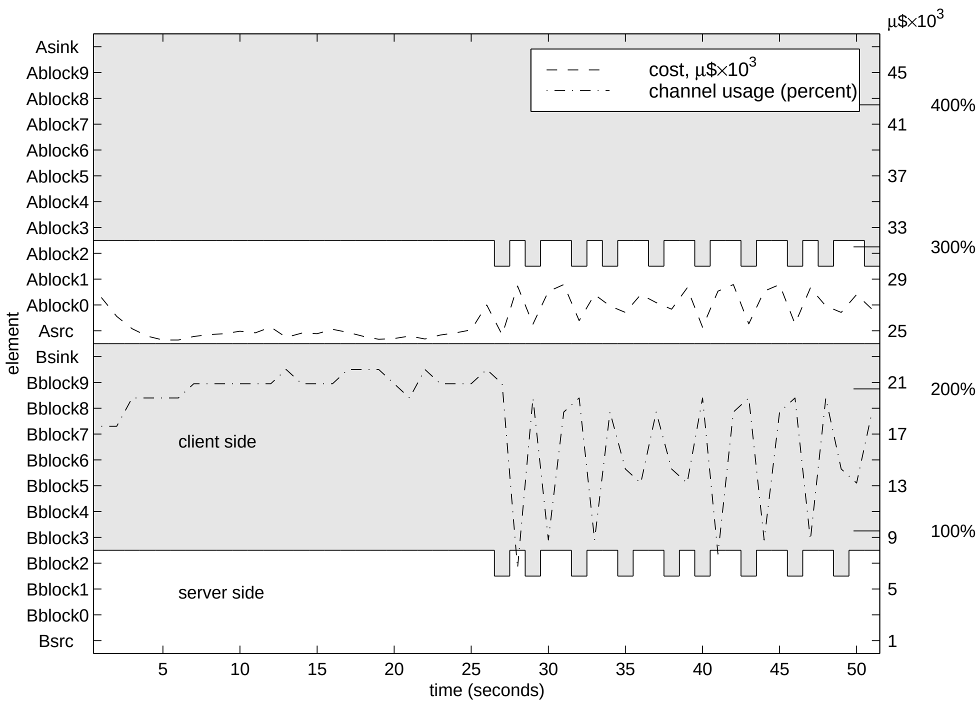


Figure 7.21: Parallel Test closed loop locality pattern — step down with reduced bandwidth and cost filtering.

7.4.4 Summary

The Parallel Test shows that the DPROJ application monitor can achieve fair distribution of resources amongst parallel dataflow paths whether they be parts of the one application or separate applications. The rate of adaptation remains the same for the Parallel Test as for the Serial Test.

The closed loop tests demonstrated the fine line between responsiveness and oscillatory behaviour. In this case, low pass filtering of the control loop was required to damp oscillations — such functionality may be implemented in resource handlers without adversely affecting reactiveness to step changes as the resource handler is specific enough to be able to distinguish between regular fluctuations in availability and major mode changes.

7.5 MOBILETRIP

MobileTRIP is a re-implementation of the Target Recognition using Image Processing (TRIP) project software developed at the Cambridge University Engineering Department Laboratory for Communications Engineering and at AT&T Laboratories Cambridge[53]. The TRIP project was instigated to develop a cheap tagging and location system to facilitate the development of *context-aware* or *sentient* computing applications.

7.5.1 Target Recognition using Image Processing

TRIP is designed to provide 3D location information for tagged objects with minimal specialised hardware. Hence, TRIP requires only off the shelf computer hardware and a webcam and relies on a computationally expensive software implementation for its operation.

Many location systems such as the GPS, the Active Badge[34] or Active Bat [96] require extensive infrastructure for both mobile and fixed components. While this is understandable for global systems such as GPS, indoor systems such as the Active Badge and Active Bat operate in a much more constrained and limited environment which could be monitored in a more economic manner.

The Active Badge provides room granularity location information through infra-red emitting badges and sensors fixed in each room of the building. While the hardware is relatively simple, it is custom made and fairly expensive. The Active Bat provides location information to within several centimetres in three dimensions and an approximation of orientation using an ultrasound emitter and ceiling mounted receivers. The emitter tags are battery operated and emit ultrasound bursts when triggered by a radio beacon. The tag therefore requires both ultrasound and radio hardware plus some controlling logic. The receivers each contain a digital signal processor to analyse the ultrasound signal and are connected back to a central server. As each tagged object requires a tag and each room requires approximately one receiver for each square metre of ceiling area, the roll-out cost of this system can be very expensive.

The TRIP system however uses commodity computers and image capture devices to locate the tags, both of which are relatively inexpensive. As web-cameras and video conferencing cameras proliferate, they become even cheaper still. The average camera peripheral is a single chip solution in which the most expensive part is often the lens. An average office room could be reasonably covered with just four or five well placed cameras. The TRIP tags are cheaper still as they can be easily printed and attached to various items. Because TRIP tags are so inexpensive, we can afford to tag liberally, not just people and computers but also everyday "boring" objects such as books and chairs and shelves etc. One of the first applications using TRIP is the TRIP Librarian, described further in section 7.5.1.

Today, many computer users already own a very powerful computer as well as a webcam and a printer. With just these tools, and a download of TRIP software, a location system for everyday use becomes a reality. Basic context-aware, sentient computing systems can be deployed rapidly and at low cost.

TRIP Tags

In the TRIP world, objects are marked using TRIP tags, 2-D circular trinary ring codes that can be easily printed and attached to objects or locations of interest. The design of the TRIP ring code follows from the requirement that these targets should be locatable at the furthest possible distance from the digital image source. This requires the selection of a target design whose features would be invariant under rotation and foreshortening and provided, at the same time, high contrast. An example tag with TRIP code is shown as Figure 7.22. The ring code is divided into two rings, each of 16 sectors. Reserving one sector for synchronisation and two sectors for parity marking leaves 13 sectors each of which corresponds to a trinary digit. This allows almost 1.6 million individual codes which is sufficient for a research system. Further codes could be obtained by using slimmer segments and cleverer synchronisation and error checking markers at the cost of some recognisability at low resolutions or longer distance.

The tags currently used have overall diameters of 5cm or 2.5cm. The dimensions of the tag, along with the resolution of the camera, determine the maximum distance from the camera at which the tag may be identified. The high contrast, rotationally invariant design of the tag enables a high identification success rate even when the tag occupies only a small number of pixels in the captured image.

The TRIP Image Processing Pipeline

The fundamental TRIP operation reduces to some heavy image processing backed up with some location service interfaces and object databases. Of most interest here is the TRIP tag recognition and identification process. The location service and object databases can hook into the system through CORBA middleware services.

TRIP performs tag identification and location in seven processing steps between capturing the image and producing the tag code. The following description of the TRIP target recognition process is condensed from that given in [53]. Figure 7.23 shows the related stages of the target recognition process.

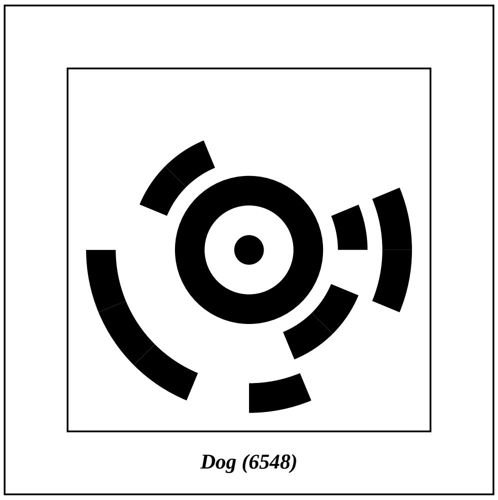


Figure 7.22: An example TRIP tag showing the full locator ring plus two segmented code rings.

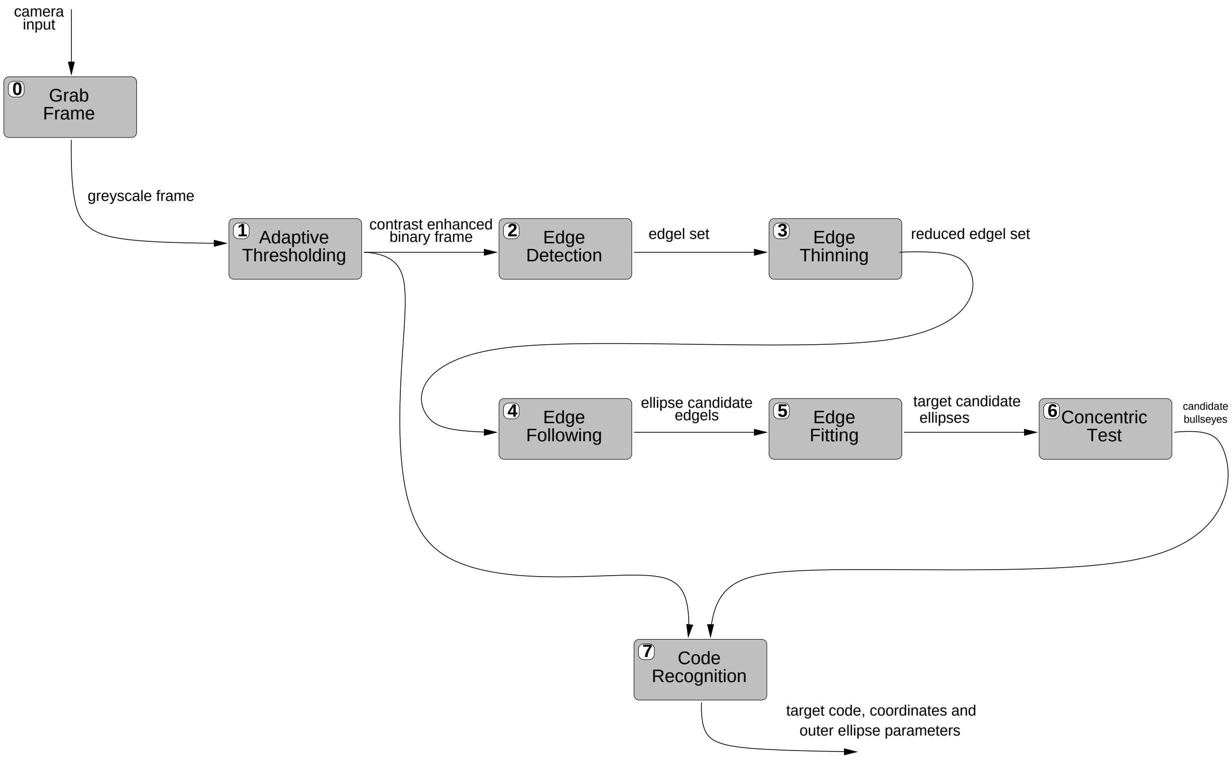


Figure 7.23: The TRIP target recognition pipeline.

1. **Image Contrast Enhancement** or **Thresholding** transforms the greyscale image to a monochrome black and white image. As the tags are high contrast black on white, any adequately lit tag in the image will persist in the new image. The current TRIP implementation uses Wellner's adaptive rolling-average thresholder[98] which considers the background illumination of each pixel. The adaptive thresholder removes the effects of shadowing and enhances the contrast. This stage makes TRIP software very robust under variable lighting conditions.

2. **Edge Detection** uses an Edge Passing Filter that identifies the pixels at or around which the image values undergo a sharp variation, i.e. its edges. The algorithm uses a five pixel weighted mask in both the vertical and horizontal directions around a pixel to calculate the x and y gradients from which an edge determination can be made. This stage results in an image indicating the presence of potential edges.

3. **Edge Thinning** localises the edge locations and removes false edges caused by noise. This is performed in in two steps: firstly, the thinning of wide edges to a width of 1 pixel (non-maximum suppression); and secondly, thresholding the results using an established minimum value to categorise true edges. This stage results in a set of *edgels* which list chains of connected edge points.

4. **Edge Following** traces the path of each of the edgels and retains only those edgels that could plausibly be part of an ellipse. Practically, this stage removes straight edgels by comparing the edgel's perimeter length to the edgel's maximum absolute dimension. This stage results in a pruned set of edgels.

5. **Ellipse Fitting** uses the Direct Least Squares Fitting of Ellipses algorithm[24] to determine for each edgel the parameters of the ellipse that it best fits. This stage results in a list of potential ellipses (specified by centre coordinates, dimensions and orientation).

6. **Concentric Ellipse Matching** filters the ellipse candidates from the previous stage to identify concentric ellipses that are likely to be part of a tag. This stage results in a list of candidate tag *bull's-eyes* and outer ellipse parameters.

7. **Code Recognition** uses the identified tag candidates and ellipse parameters to search the thresholded image from stage 1 for the ring code features. It first identifies the synchronisation sector and then follows the ellipse to determine the code value. If the synchronisation sector is not found or the code does not pass the parity error check, then the code and tag candidate is rejected. This stage results in a set of positively identified tags, each located within the image frame by x and y coordinates, a and b ellipse abscise parameters and the ellipse rotation with respect to the frame. If the position, orientation and lens characteristics of the camera is known, these values are sufficient to determine the 3D location and orientation of the tag.

An example run of the image processing pipeline is shown stage by stage in figure 7.24. The identified tags are marked on the final image by the placement of cross-hairs.

Application: TRIP Librarian

The TRIP Librarian is one of the first demonstration applications to use the TRIP service. It is a simple example application that uses the TRIP infrastructure to implement a laboratory-wide book tracking and indexing service[53].

Each book in the laboratory is entered into a database and a TRIP tag is printed and affixed to the book's spine. Similarly, bookshelves and other plausible book locations are tagged and entered into the database. Figure 7.25 shows a snapshot of some books which have been tagged.

The application uses TRIP location services to track and locate the books on demand. To do this, it utilises video input from cameras distributed around the laboratory which are aimed at the more populous bookshelves. Additional video input may be provided from a manually operated hand held camera which the human librarian operates. This allows the librarian to wander the lab and record the presence of books in places not monitored by fixed cameras. The recording is then downloaded from the video-camera and processed offline. The shelf tags provide localisation hints to the TRIP processor which associates temporal proximity of tags in the recording to a close spatial proximity.

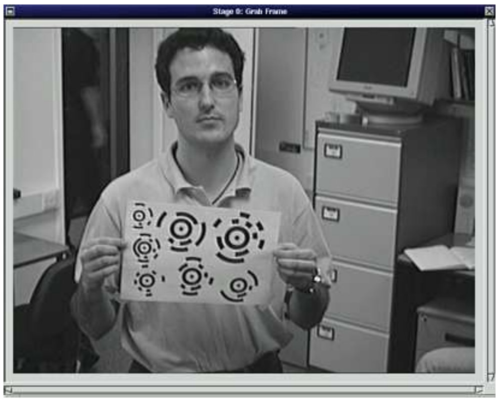
7.5.2 MobileTRIP Application Design

The MobileTRIP project aims to "invert" the TRIP location system, using it on a mobile host with a camera to identify tagged objects within view. This involves image capture, image processing, tag identification, information overlay and finally presentation of the information back to the user

The features of this application make it suitable for implementation using DPROJ framework, in particular:

- the ultimate source and sink for the data are colocated on the mobile device,
- the application processes large amounts of data,
- the data is analogesque in nature,
- it requires substantial processing capability, and
- the processing may be performed at widely ranging variable qualities and rates.

Figure 7.26 shows the top-level application elemental composition. The structure is not linear and has two parallel data paths at this level. The FrameGrabber and TRIP elements are Container elements and hence expand to include a further graph of functional elements. These both serve to reduce the complexity at the top-level and also pose as reusable components suitable for use in different applications. Single level expansion of the element hierarchy is shown in Figure 7.27. In this view it can be seen that the



Stage 0: Grab Frame

Stage 1: Adaptive Thresholding

Stage 2: Edge Detection

Stage 3: Edge Thinning

Stage 4: Edge Following

Stages 5-7: Target Recognition

Figure 7.24: Images from the TRIP image processing stages.

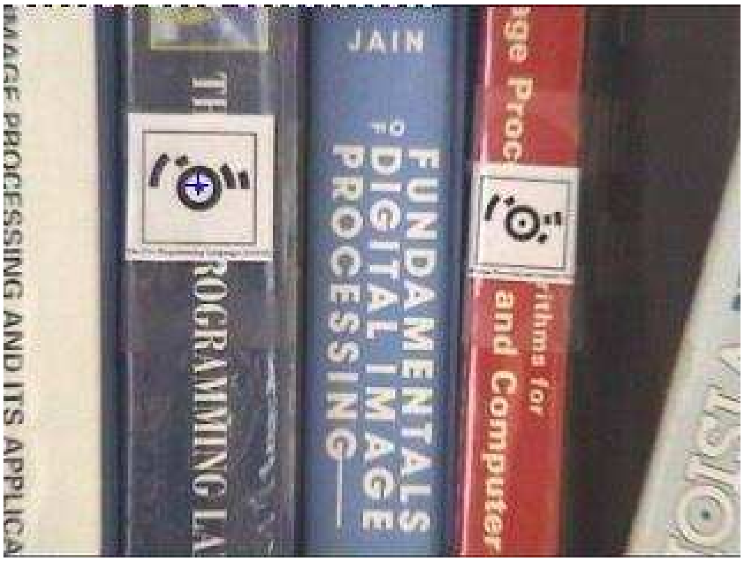


Figure 7.25: TRIP Librarian input image.

overall data path is more complicated as the internal data path of the TRIP element is exposed. The application data path is not strictly linear with up to three parallel paths at one point.

The FrameGrabberMulti element incorporates a FrameBuffer multiplex which allows the user to select the input source from either an active camera or from a file (for testing or offline analysis). It is constrained to operation on the client only as is the FrameDisplay.

The TRIP element provides the core image processing and target recognition functionality for the application. It imports a FrameBuffer and exports a trip identity code. The TRIP element and its contained sub-elements are allowed to operate on either the client or the server.

In operation, the FrameGrabber pushes captured frames into a FrameBranch which duplicates it, sending one copy to the FrameTagOverlay and the other to the TRIP element. The TRIP element proceeds to locate and identify any targets within the image via its multiple stage pipeline. The resulting tag identifier is then translated into a real world description by TRIPtaglookup and this too is passed to the FrameTagOverlay where it is combined with the original frame. This is then presented to the display device by FrameDisplay. Figure 7.28 shows the final overlayed image as presented to the user. The tag in the image is overlayed with a short label describing the tagged object.

7.5.3 TRIP Pipeline Implementation

The design allows for alternate implementations for some elements. These alternates use different algorithms and have a different cost-performance function. The following table lists the implementations currently available for elements in the TRIP pipeline. For each element class and implementation variant it lists a description, an indication of its

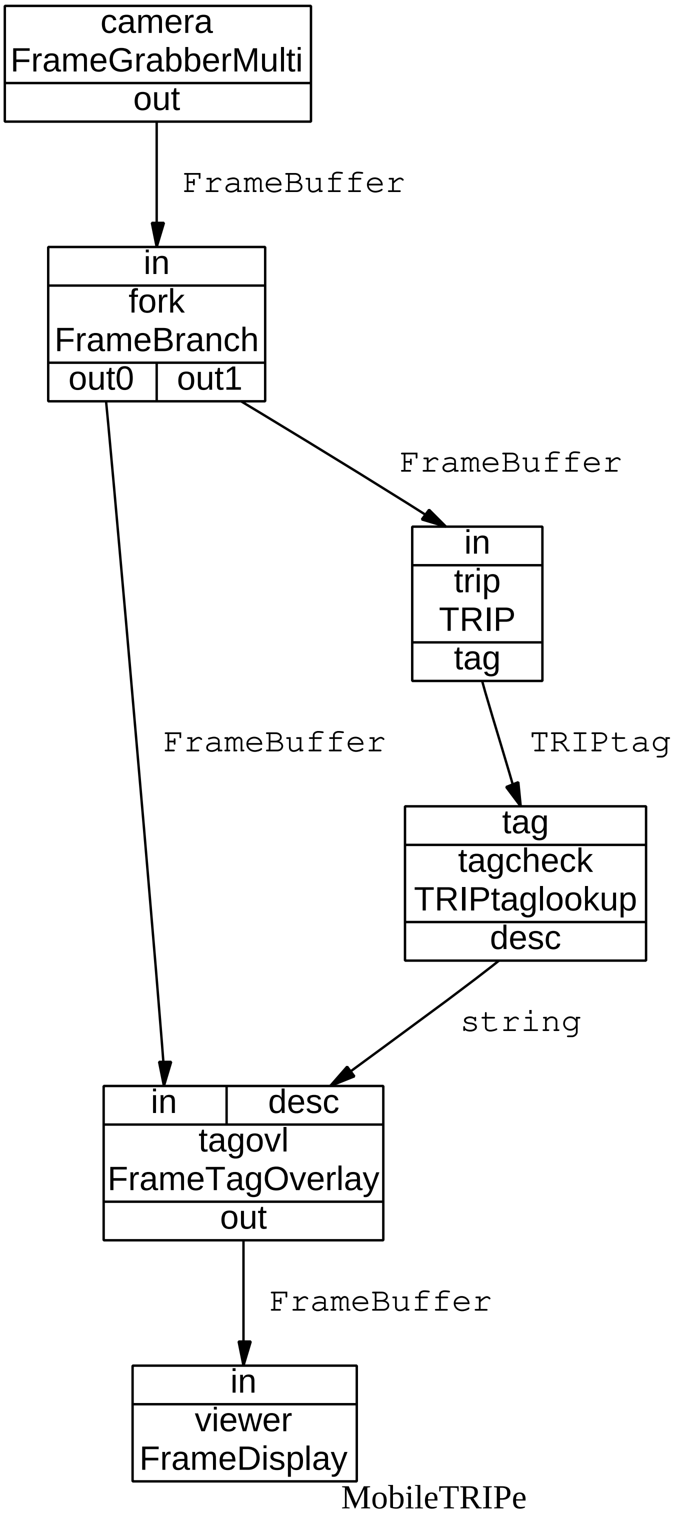


Figure 7.26: MobileTRIP application inter-module dataflow (toplevel view).

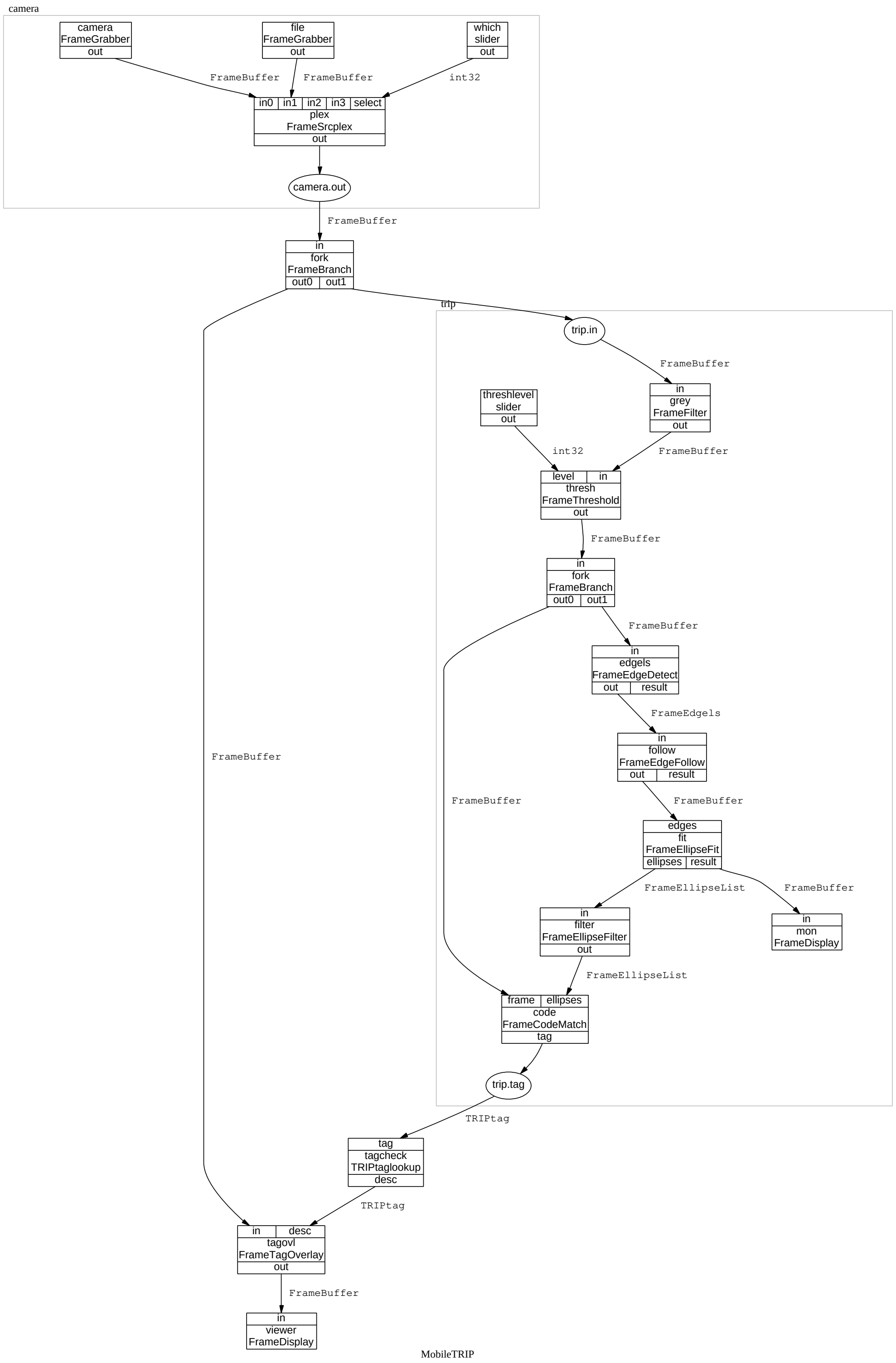


Figure 7.27: MobileTRIP application inter-module dataflow (+1 level expansion).

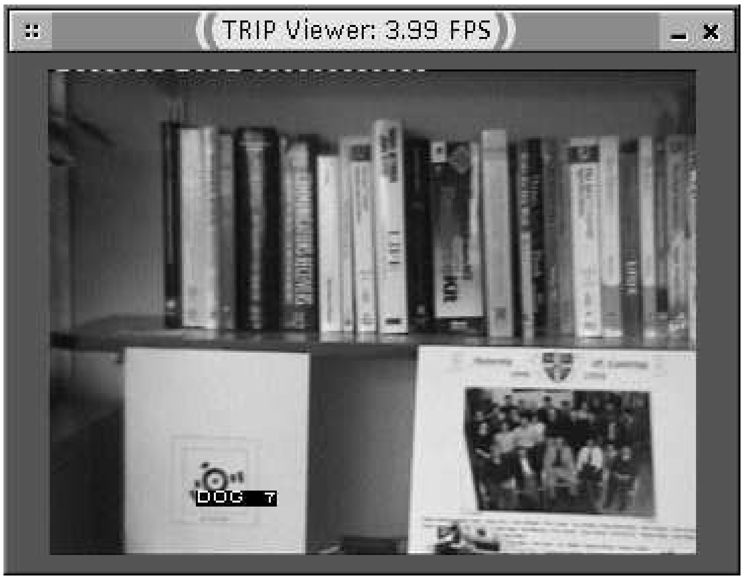


Figure 7.28: MobileTRIP screen capture of tag based information overlay.

relative performance, an approximation of its CPU requirements in terms of the number of pixels in the image (N) or the number of edges found (E), and its location constraints (Server or Client).

element	variant	description	perf.	O_{CPU}	loc.
Greyscale	Basic	convert colour frames to greyscale	500	3N	SC
Thresholding	Basic	fixed contrast enhancement	400	100N	SC
	Adaptive	adaptive contrast enhancement	800	1000N	S
EdgeDetect	Basic	basic edge detection	500	50N	SC
EdgeFollow	Basic	basic edge following	500	50N	SC
EllipseFit	Basic	basic ellipse fitting	500	100E	SC
EllipseFilter	Basic	basic ellipse filter	500	50E	SC
CodeMatch	Basic	basic code matching	500	100	SC

Additionally, the TRIP pipeline incorporates several ancillary elements which do not directly process the data:

FrameBranch	Basic	bifurcation of the frame data path	100	N	SC
Slider	Basic	provide an integer source range 0-255	100	10	SC
FrameDisplay	Basic	displays stage progress for debugging	100	N	SC

The Camera generates frames at up to 10 frames per second and is limited by the throughput of the application. Each exported FrameBuffer is in 24 bit RGB format at a resolution of 320x240 (225kB). CPU requirement is $O(N)$ with a low coefficient where N is the number of pixels.

Greyscale reduces the FrameBuffer to 8 bit greyscale (75kB). CPU requirement is low and $O(N)$.

Thresholding limits the frame pixel values to 0 or 1 values but the exported FrameBuffer is still 8 bits deep (75kB). The Basic implementation does hard limiting based on the value provided by the user via the slider and operates in $O(N)$. The Adaptive thresholder uses a sliding window to threshold each pixel value according to the average intensity in that region. This provides a result with greater detail but requires

The export from this stage is this list of edges and hence the amount of data exported is dependent on the image contents. For reference, the image in figure 7.28 showing a single tag and several other high contrast edges typically results in approximately 10 lines, each of up to 20 points, hence using 16 bit words for coordinates: approximately $10 \times 20 \times 4 = 800$bytes $= 0.8$kB per edge set.

Ellipse Fitting fits an ellipse to each of the edges. This results in a list of ellipses, each a 20 byte record.

Ellipse Filtering searches for concentric ellipses amongst the list provided. Any ellipse which is represented by at least two lines is passed through the filter to export.

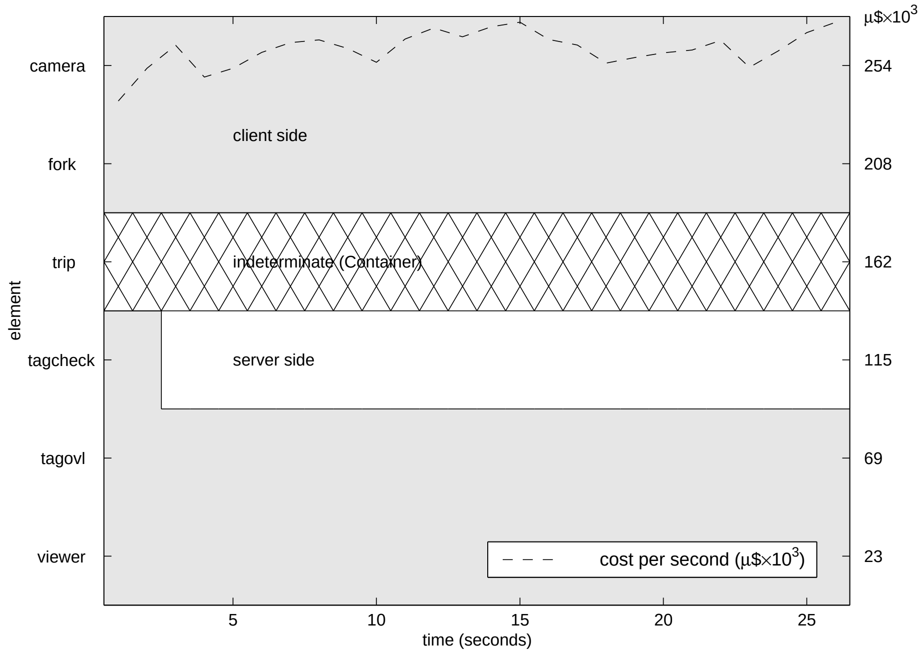
Code Matching takes the list of ellipses and the thresholded frame to determine the code for each ellipse. If it is valid, it is exported as an x,y coordinate pair and an identification number.

7.5.4 MobileTRIP Operation

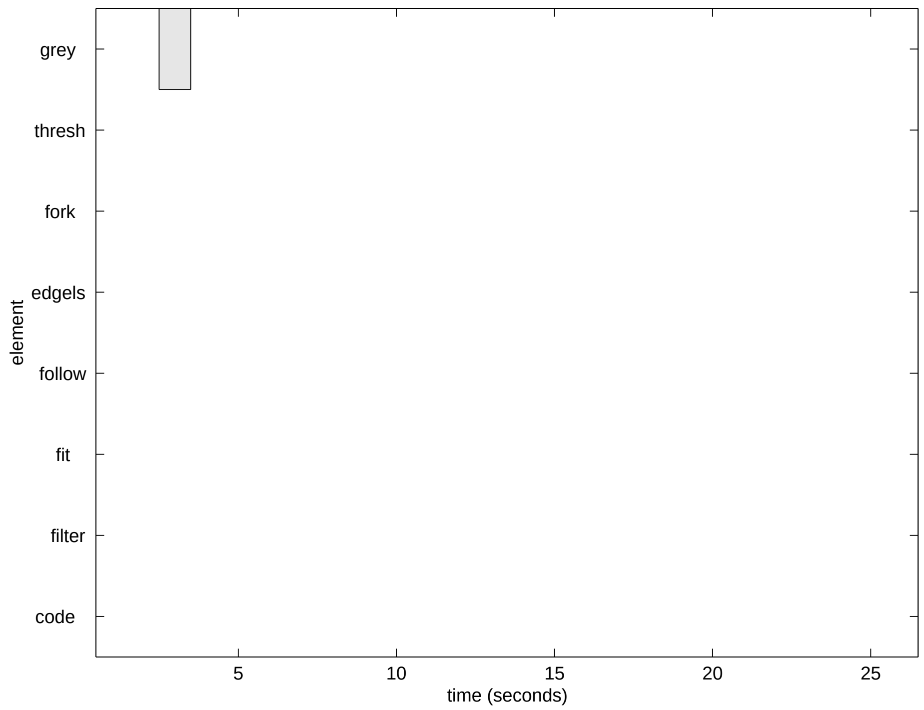
The application is run in a single client, single server configuration in a similar environment to that used for the Serial and Parallel Tests previously. Initially, the resource manager uses fixed pricing of $C_s = 10$, $C_c = 100$ and $C_n = 38$.

Figure 7.29 gives the locality plots showing the location of operation of the application's major components at startup. It is divided into two plots: figure 7.29(a) gives the locations of the elements at the top level while figure 7.29(b) gives the locations of the elements comprising the TRIP pipeline Container. The TRIP element in the top-level view is therefore hashed indicating that it may be distributed and doesn't necessarily operate at a single location. The FrameGrabber Container is not shown separately as it is a trivial element that is always located on the client. These plots show that for basic operation (performance set-point is at 50 — *i.e.* using Basic thresholding) where communication is relatively cheap, the bulk of the processing is allocated to the server. The partitioning is such that the FrameBuffer path only crosses the communication boundary once.

Figure 7.30 gives the locality plots for the application when the communication cost is subjected to a ramp up from $C_n = 10$ at $t = 10$ to $C_n = 10000$ at $t = 40$. As the communication cost increases, reconfiguration of the application gradually moves elements of the TRIP pipeline to the client. First, the Greyscale filter is moved, reducing the bandwidth by 66%. Following this, the adaptive system is can then relocate the Thresholder but this does not provide a saving in bandwidth as its export bandwidth requirements are approximately the same as it imports. The location of the Thresholder continues to alternate with fluctuations in measured data rates. The introduction of hysteresis to the relocation evaluation would prevent such rapid oscillations. At this point, MobileTRIP presents a challenge to incremental adaptation. Further relocation of elements which 'border' the communications break can only increase the overall bandwidth from this point as another FrameBuffer data path must be cut. The adaptation mechanism hence is unable to proceed past this point. This shows the limitations of the incremental approach where the adaptation mechanism settles on a local minimum when a better configuration is obvious but is inaccessible as the system must first suffer a temporary increase in cost before gaining the benefits.

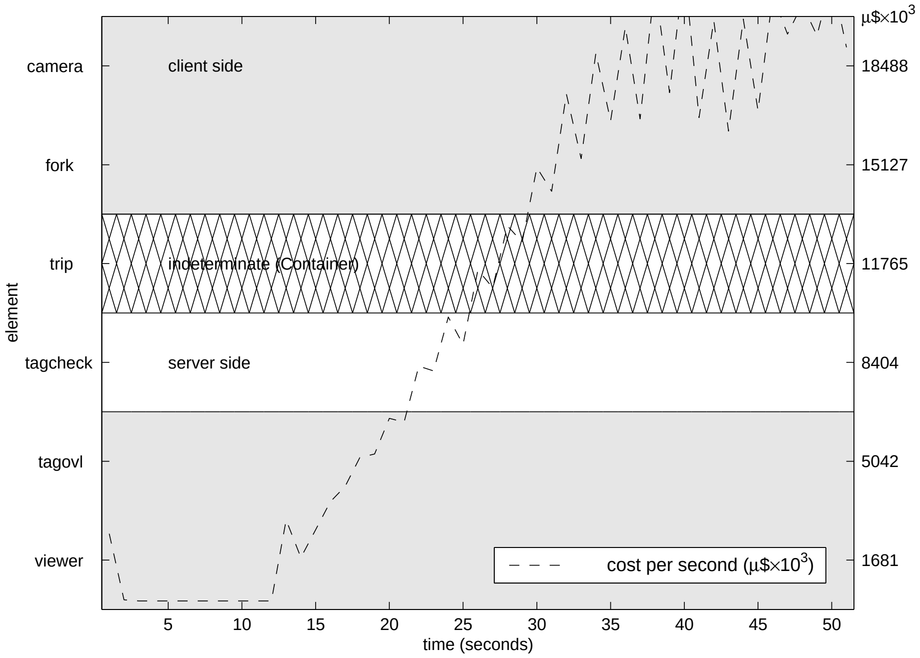


(a) MobileTRIP top level locality plot.

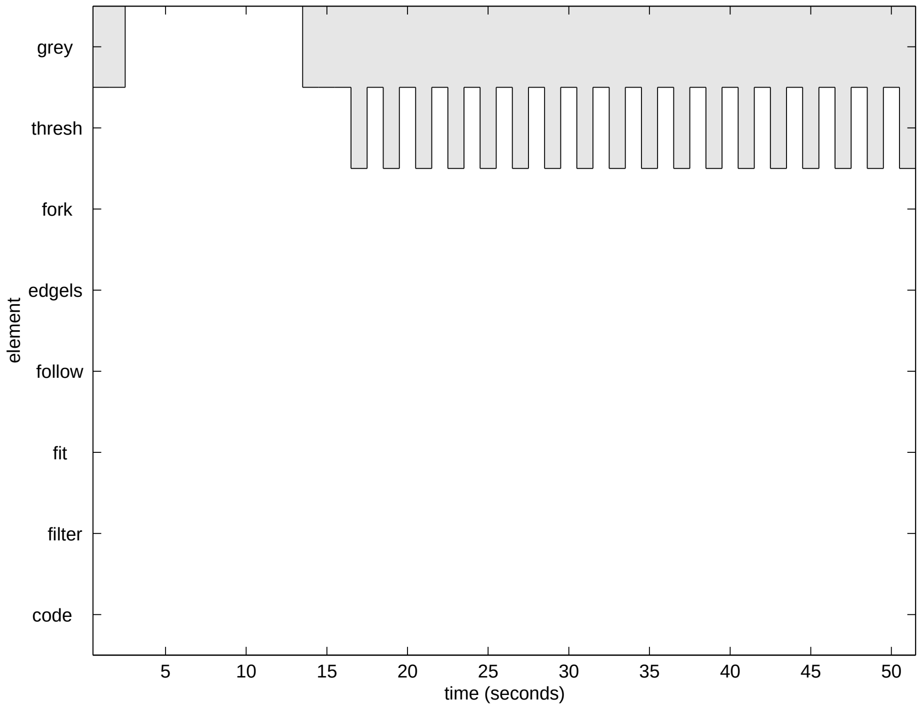


(b) MobileTRIP.TRIP component locality plot.

Figure 7.29: MobileTRIP component locality plots — startup.



(a) MobileTRIP toplevel locality plot — ramp up communications cost over $10 < t < 40$.



(b) MobileTRIP.TRIP component locality plot — ramp up communications cost over $10 < t < 40$.

Figure 7.30: MobileTRIP component locality plots — ramp up.

While the ramp-up example shows that the system cannot optimally adapt to gradual but large increases in communications cost, such a situation is unlikely to be so severe in a real operational environment. In a practical system, such a large change in communication cost is likely to derive from a change in communication mode, for example in the case of channel overload (exceeding the channel capacity) or disconnection.

Figure 7.31 gives the response of the application when subjected to a simulated disconnection. In this scenario, the resource handler for the communications channel notices that the channel is or will soon be disconnected. Hence, it raises the cost of the channel and also of the CPU for hosts which communicate with the client via the channel. For the trace used to simulate disconnection in figure 7.31, disconnection is anticipated at $t = 25$ and both C_n and C_s are raised to 10000 (the costs are adjusted after the configuration sampling at $t = 25$ but before the instrumenting and reconfiguration at $t = 26$). After this increase, the cost of operating various elements on the server become infeasibly expensive and hence all elements but one are immediately relocated to the client. The last element is relocated in the following iteration. The application is reconfigured to operate through disconnection within two sample periods.

7.5.5 Analysis of Operation

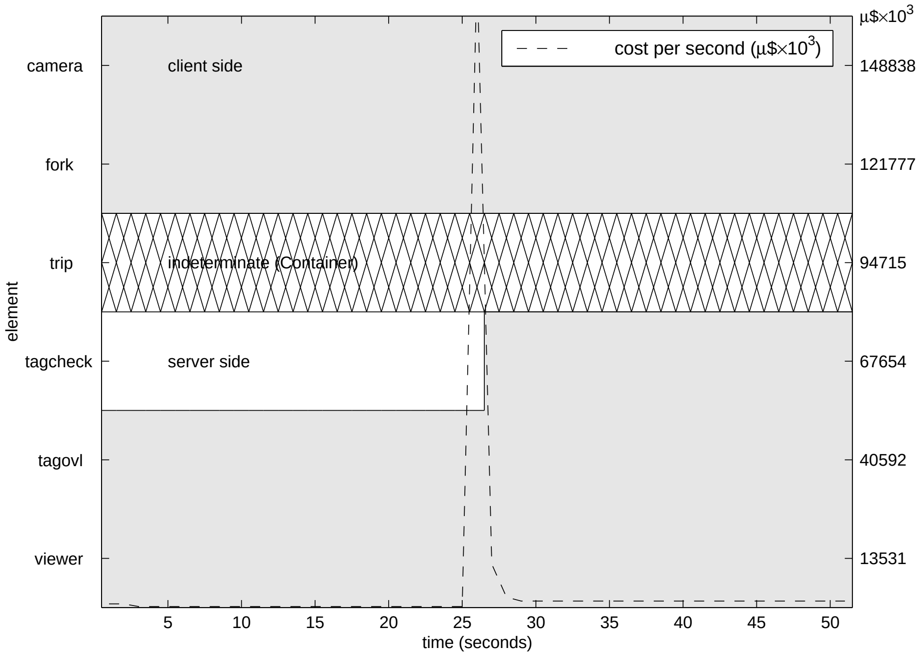
The MobileTRIP application serves as an example of a useful application that can benefit from the automatic adaptation service provided by the DPROJ framework. Adaptation is achieved primarily via manipulation of process locality and secondarily via manipulation of fidelity. It demonstrates two key capabilities of the framework's automatic adaptation mechanism beyond those demonstrated by the Serial and Parallel tests. Firstly, it shows that the framework can arrive at sensible and efficient configurations even for applications that exhibit a non-linear data path. Secondly, it does so for a multi-level hierarchically structured application.

However, the example also shows that the pervasive adaptation service provided by DPROJ may result in a configuration which is less than optimal in some situations. In general terms, this shortcoming arises from the incremental approach used by the adaptation process in minimising resource costs. This approaches a local minimum which may not be the global minimum. The ability of the incremental approach to pass local minima is limited by the exploration range of the approach.

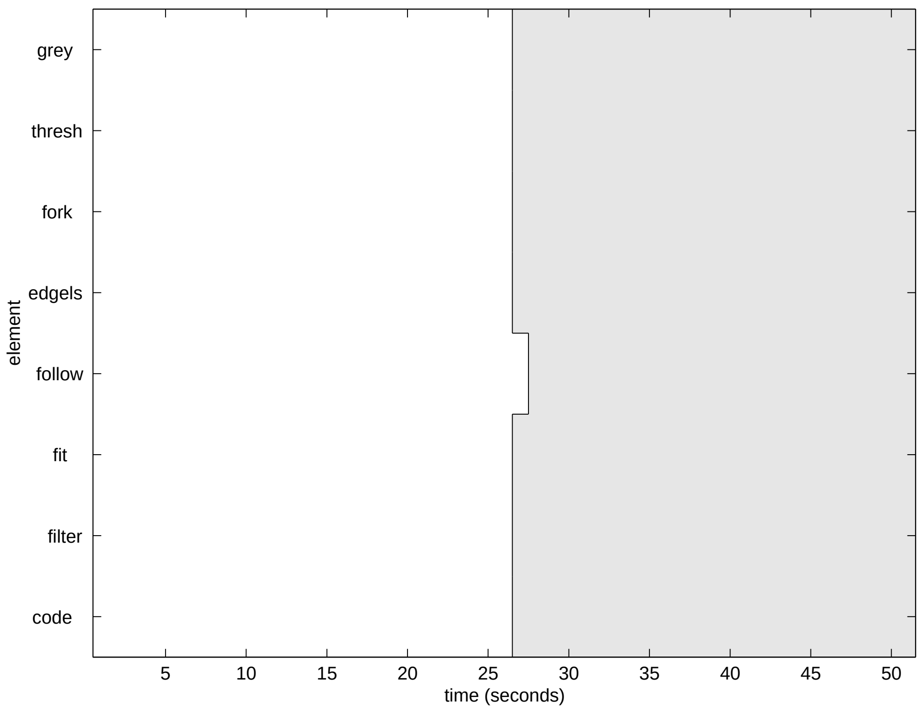
Alterations to the construction of the model may be able to alleviate some situations similar to this one if they enable wholesale shifting of an entire Container element from one location to another. To achieve this, the merge operation for Containers would be altered as follows for merging two Containers e_i and e_j where there is a one to one correspondence between sub-elements::

$$e_i^c + e_j^c \rightarrow e^s\{e_i^c, e_j^c, e^c\{e_{i0} + e_{j0}, e_{i1} + e_{j1}, \ldots, e_{in} + e_{jn}\}\}$$

Hence, instead of merging to produce a Container in which each sub-element is a Set of the corresponding sub-elements of the original Containers, the original Containers, with location constrained to a single host, could be offered as equivalent alternatives. This results in a Set offering distinct selections between a) all of Container i on host A,



(a) MobileTRIP toplevel locality plot — disconnection at time $t = 25$.



(b) MobileTRIP.TRIP component locality plot — disconnection at time $t = 25$.

Figure 7.31: MobileTRIP component locality plots — disconnection.

b) all of Container j on host B, or c) a mix of Containers i and j operating on both hosts concurrently (as before). The adaptation process can hence swap from a partitioned Container (case c) to a Container fully operational on a single host (cases a or b) in a single iteration.

7.6 SUMMARY

The tests results of the Serial and Parallel Test applications detailed in this chapter verify the operation of the DPROJ framework in performing application adaptation through control of the locality and fidelity of application components. These tests were performed in open loop mode to verify correct adaptation given set cost patterns and then in closed loop mode, verifying the operation of the feedback loop in regulating the use of resources. This verification of basic adaptation operations and application structures forms the basis for tests of larger applications built through compounding of these basic structures.

MobileTRIP is an example of such an application which possesses multiple parallel data paths and a multi-tiered hierarchical structure, but which at the same time is simple enough to observe and analyse through the course of adaptation. It's implementation serves as a guide to the use of the framework to construct mobile applications, in this case derived from a non-mobile version, and demonstrates the utility of adaptation for real applications.

The execution of these tests leads to characterisation of the DPROJ adaptation process in three areas: Firstly, it has been shown that the process facilitates rapid adaptation in response to resource changes; the response is asymmetrical for step up and step down inputs, but in the worst case, reconfiguration of the application is achieved in a time period proportional to the divergence from the original configuration. Secondly, the tests show that the DPROJ adaptation process can achieve fair distribution of resources amongst parallel dataflows or applications. Thirdly, it is demonstrated the incremental approach to adaptation as a computationally tractable solution for online dynamic adaptation, although it is limited in that it is unable to provide a globally optimum result that might be achieved using a computationally expensive offline algorithm such as Min-Cut Max-Flow graph cutting.

8 Conclusions and Further Work

The thesis of this dissertation, stated in chapter 1 and discussed, expanded and demonstrated through the body of this dissertation, is that it is both feasible and beneficial to provide automatic dynamic adaptation of applications in mobile computing systems. Furthermore, it has been shown that Pervasive Adaptation, as embodied in the DPROJ framework, is a plausible and useful approach to constructing such applications and such a system.

To the best of the author's knowledge, DPROJ represents the first system to bring automatic dynamic adaptation to mobile applications. It is a plausible solution as it has been demonstrated to successfully adapt running applications against changing resource availability, using an adaptation decision mechanism that has an operational complexity proportional to the structural complexity of the application and can achieve an optimal configuration in response to environmental changes within a time period which is proportional to the difference in configuration of the application before and after the change.

8.1 CONTRIBUTIONS

This thesis and the resulting DPROJ framework has made several important contributions to the field of adaptation for mobile computing. In summary, this thesis has:

1. **proposed metric unification as a means of reducing adaptation complexity** — The adaptation system uses metric unification to reduce resource costs and performance metrics to simpler single dimensional values. Resource costs are unified to a financial cost metric (dollars) while performance is held as an application specific notion and hence is reported by application components as a single unit-less value. This approach is both natural and powerful, enabling reduction of complexity in the adaptation process by reducing the adaptation space to a two dimensional cost-benefit function.

2. **developed hierarchical cost-benefit based adaptation as a low complexity adaptation process** — The application monitor models the application as a simple cost-benefit function which presents the application's profile from a "user" viewpoint: the application can perform at a range of performance settings, each of which has an associated cost. This profile is a simple piecewise-linear function that can be accumulated hierarchically over the structure of the application and acted upon by an automated policy at the highest level.

3. **demonstrated pervasive adaptation as a fair, efficient and meaningful form of integrated adaptation** — Integrated adaptation promises the benefits of both application-transparent and application-specific adaptation. From application-transparent approaches, it inherits centralised resource management which lends it fairness and efficiency in handling resources; from application-specific adaptation, it inherits meaningful adaptation mechanics. Using its centralised resource management and non-specific notions of performance, DPROJ has been shown through operation with several test cases to provide fair and efficient use of resources and also to provide meaningful adaptation.

4. **demonstrated pervasive adaptation as a means of automatic adaptation** — Other integrated adaptation systems such as Odyssey can offer only resource availability hints to applications, which applications must respond to by adapting themselves. DPROJ is designed to perform dynamic manipulation of an application's configuration directly and hence provides automatic adaptation to suitably constructed applications. The application developer is relieved of explicitly specifying the adaptation procedure.

5. **provided separation of adaptation concerns from functionality for application developers** — DPROJ encapsulates the adaptation logic and control in the application monitor which offers adaptation as a service to suitably constructed applications. As the adaptation process is based on cost-benefit analysis, it is sufficiently general to operate on an application without specific knowledge of the application's purpose. The adaptation operates using application structure meta-data provided by the developer when designing the application. An application developer can concentrate on application functionality without needing to integrate adaptation code into the application's function.

8.2 IMPLICATIONS

The result of this thesis has broad implications for mobile applications. Applications operating within the DPROJ framework can be automatically reconfigured to account for environmental or resource conditions not originally conceived by the developer. Resources are managed centrally, providing fair and non-competitive allocation yet adaptation remains meaningful as it is based on application-specific notions of performance.

The DProj framework also implies a new approach to application development with a heavy emphasis on modularised design. The cost of automatic adaptation comes in restrictions on developer freedom in application design. In particular, the developer must first provide complete application structural meta-data via the EC element composition language, and secondly, ensure that the application abides by several guidelines for simple distribution which emphasise encapsulated application components, asynchronous call use, and awareness of state and latency limitations.

8.3 Future Work

The work described in this dissertation constitutes a first step into the open field of automatic adaptation, and as such is open to improvements to increase its functionality and ease of use and also to reduce its operating overhead.

The DProj framework is an implementation of the concepts outlined in this dissertation and focuses on the application monitor which provides the adaptation "core" but also provides primitive implementation of peripheral services such as element migration, service discovery, adaptation policy and resource management. The adaptation core is sufficient for adaptation of simple applications such as the examples of chapter 7 but requires further refinement and features to increase its flexibility and make it suitable for more sophisticated applications. Many possible advances are described in the feature wish-list of section 6.7.5. These focus on improving the flexibility of the system, adding introspection capabilities to allow an application to dynamically alter its own structure and high level performance reporting for an improved notion of Container level performance which can then consider such aspects as end to end latency.

Also suggested are several obvious improvements to the supplied peripheral services, including the underlying element "migration" scheme and state management which are limited or non-existent. DProj relies on redundant instances of application elements to perform naïve "migration" of application elements during reconfiguration. This approach is simple and fast but cannot handle elements which have some state. The cost-performance evaluation of application operation tries to minimise its dependence on element state by placing strict limitations on the operation of elements and through making assumptions about the nature of a particular implementation. Perhaps the most important advance would be the replacement of the element instantiation and management system with another available CORBA based object control framework. Several of these exist including the CORBA standard LifeCycle service (as used to implement migratable GUI components in DAWS[31]) or the LocALE Location Aware LifeCycle Environment which provides further management extensions to the LifeCycle service[52]. These services provide true object migration, fault-tolerance and state transfer for CORBA objects and would make for a more flexible and capable middleware basis for implementation of the DProjframework.

EC Definitions for the Serial Test Application

A

The N-Block SerialTest application comprises a TestSrc data source and a TestSink data sink linked via N TestBlock data processing elements. Figure A.1 shows the application data flow for a 10-block application. Previously, table 7.1 showed the basic characteristics of each element type. The 'frame' data passed between processing blocks is of data type `TestData` as defined in figure A.2. It is a structure constituting a variable sized block of arbitrary data and a size measure.

The `TestSrc` element generates t_0-byte data frames at 5 frames per second and is defined by the code in figure A.3. It takes no imports but has a single `TestData` export named `out`. It posesses the standard properties `type`, `serverside` and `clientside`. Their values indicate that the module has implemention variant `Basic` available for operation on both client and server. This is generally true for the module, however specific applications may wish to constrain its locality at a higher level. In this case, it is done using the `modproperty` pragma in the SerialTest application module definition described below.

Each `TestBlock` element exports a data frame twice the size of the one it imports. It is defined by the description in figure A.4 and has a single import, `in`, and a single export, `out`, both of data type `TestData`. It has `Basic` implementation variants for both client and server side in the general case.

The `TestSink` element receives the completed frame and displays it and is defined by the description in figure A.4. It has a single import of type `TestData` and no exports. It has `Basic` implementation variants for both client and server side in the general case.

Figure A.6 defines the actual 10-block SerialTest application. Logically, it comprises only a single module, a container module, that contains the component modules comprising the application (its children). The definition lists the modules comprising the container as type, name pairs. Additionally, it lists the logical connections between the components as *wires*. Note that the container itself does not have `serverside` and `clientside` properties as the container does not have a logical presence outside of its child modules. The location of the container is the location of its children which may be spread across more than one hosts. The `modproperty` pragmas can change named properties of the named child. In this application, the `TestSrc` (`src`) is constrained to operate solely on the server while the `TestSink` (`sink`) is constrained to the client.

Each `TestBlock` may operate on either the server or the client.

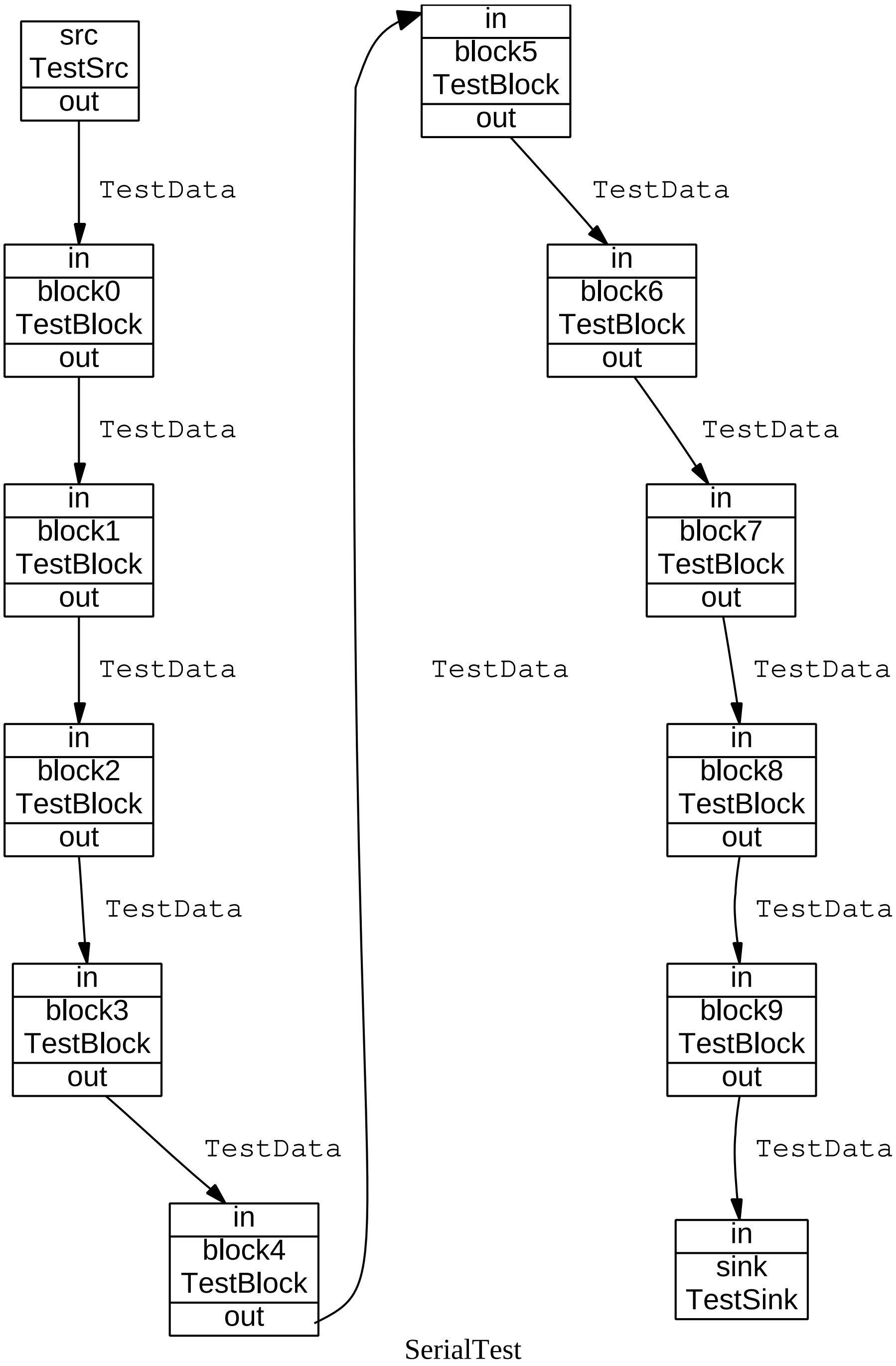


Figure A.1: 10-block SerialTest dataflow.

```
# CodeGenerator [DataType definition file]
#    - automatic code for DataType TestData
#    - Tue Nov 14 15:45:45 GMT 2000

DataType TestData {

    type struct
        int32    size
        buffer   data

}
```

Figure A.2: TestData.typ data type definition.

```
# CodeGenerator [Module definition file]
#    - automatic code for module TestSrc
#    - Tue Nov 14 15:45:45 GMT 2000

Module TestSrc {

    type normal
    language cpp
    property string      serverside    Basic
    property dummyType   type     normal
    property int32       blocksize     10240
    property string      clientside    Basic
        export  TestData       out

}
```

Figure A.3: TestSrc.mod definition.

```
# CodeGenerator [Module definition file]
#   - automatic code for module TestBlock
#   - Tue Nov 14 15:45:45 GMT 2000

Module TestBlock {

    type normal
    language cpp
    property string     serverside    Basic
    property dummyType  type    normal
    property int32      growsize      10240
    property string     clientside    Basic
        import  TestData      in
        export  TestData      out

}
```

Figure A.4: TestBlock.mod definition.

```
# CodeGenerator [Module definition file]
#   - automatic code for module TestSink
#   - Tue Nov 14 15:45:45 GMT 2000

Module TestSink {

    type normal
    language cpp
    property string     serverside    Basic
    property dummyType  type    normal
    property string     clientside    Basic
        import  TestData      in

}
```

Figure A.5: TestSink.mod definition.

```
Module SerialTest {

    type container
    language cpp
    property dummyType   type      container
    property int32       application      1
    container

        module  TestSrc       src      (30,200)
        module  TestBlock     block0   (150,200)
        module  TestBlock     block1   (250,200)
        module  TestBlock     block2   (350,200)
        module  TestBlock     block3   (450,200)
        module  TestBlock     block4   (550,200)
        module  TestBlock     block5   (150,300)
        module  TestBlock     block6   (250,300)
        module  TestBlock     block7   (350,300)
        module  TestBlock     block8   (450,300)
        module  TestBlock     block9   (550,300)
        module  TestSink      sink     (670,300)

# the data path
        wire    src.out - block0.in
        wire    block0.out - block1.in
        wire    block1.out - block2.in
        wire    block2.out - block3.in
        wire    block3.out - block4.in
        wire    block4.out - block5.in
        wire    block5.out - block6.in
        wire    block6.out - block7.in
        wire    block7.out - block8.in
        wire    block8.out - block9.in
        wire    block9.out - sink.in

# anchor the ends
        modproperty src.clientside 0
        modproperty sink.serverside 0

}
```

Figure A.6: SerialTest.mod definition.

Bibliography

[1] R.S. Allen. *A Graphical System for Parallel Software Development*. PhD thesis, University of Newcastle upon Tyne, July 1998. 4.2.1, 4.2.1

[2] Y. Amir, B. Awerbuch, and R.S. Borgstrom. The java market: Transforming the internet into a metacomputer. Technical report, Department of Computer Science, John Hopkins University, 1998. 5.1.2

[3] O. Angin, A. T. Campbell, M. E. Kounavis, and R. R-F. Liao. The mobiware toolkit: Programmable support for adaptive mobile networking. *IEEE Personal Communications — Special Edition on Adaptive Mobile Networking*, 5(4):32–34, August 1998. 4.1

[4] *An Introduction to ANSAware 4.0*. APM Ltd., Poseiden House, Castle Park, Cambridge, UK, February 1992. 2.4.3

[5] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal*, 6(3):138–151, May 1998. ACM/Springer-Verlag. 4.1

[6] Babelfish. Babelfish web site. `http://babelfish.altavista.com`, 2001. 5.1.2

[7] A Baggio. Design and early implementation of the cadmium mobile and disconnectable middleware support. Rapport de Recherche RR-3515, INRIA, April 1998. http://www-sor.inria.fr/publi/DEICMDMS_rr3515.html. 3.5.3

[8] A.G. Barto, R.S. Sutton, and C.W.Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13(5):834–846, Sept 1983. 4.3.5, 4.3.5

[9] A.D. Birrell and B.J. Nelson. Implementing remote procedure call. *ACM Transactions on Comuting Systems*, 2(1):39–59, February 1984. 3.4.2

[10] J. Bolliger, T. Gross, and U. Hengartner. Bandwidth modelling for network-aware applications. In *Proceedings of IEEE Infocomm'99, New York, USA*, March 1999. 7.2

[11] R. A. Brooks. A robust layered control system for a mobile robot. Technical Report AIM-864, AI Lab, MIT, September 1985. 4.2.1

[12] S. Buckingham. An introduction to the general packet radio service. Technical report, Mobile Lifestreams Limited, 9 The Broadway, Berkshire, UK, January 2000. `http://www.gsmworld.com/technology/yes2gprs.html`. 3

[13] V. Bush. As we may think. *The Atlantic Monthly*, July, 1945. 2.1

[14] L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000. 1.4

[15] Luca Cardelli. Mobility and security. In *Foundations of Secure Computation, Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation, Marktoberdorf, Germany*, NATO Science Series, pages 3–37, August 1999. 2.6.1

[16] Luca Cardelli. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603, chapter Abstractions for Mobile Computation, pages 51–94. Springer Lecture Notes in Computer Science, 1999. 2.6.1

[17] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model*. Redmond, WA, USA, October 1995. 2.5

[18] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concept and Design*. Adison Wesley, 2001. 2.4

[19] DARPA Internet Program. *Transmission Control Protocol*, September 1981. 2.3

[20] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. Network Working Group, December 1998. 2.3.2

[21] T. Edmonds, S. Hodges, and A. Hopper. Pervasive adaptation for mobile computing. In *to appear in Proceedings of the 15th International Conference on Information Networking*. IEEE, IEEE Computer Society Press, February 2001. (document)

[22] T.M. Edmonds. A decision engine for robot football. Technical report, Laboratory for Communications Engineering, Engineering Department, University of Cambridge, Trumpington St, Cambridge, UK, CB2 1PZ, July 1998. 4.1

[23] T.M. Edmonds, S.E. Hodges, and A. Hopper. An adaptive thin-client robot control architecture. In *Proceedings Real-Time Computing Systems and Applications*, Hong Kong, December 1999. IEEE. (document), 2.1, 4

[24] A. Fitzgibbon, M. Pilu, and R. Fisher. Direct least squares fitting of ellipses. In *Proceedings of International Conference on Pattern Recognition*, Vienna, Austria, August 1996. 5

[25] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM, Dec 1999. 3.6

[26] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1964. 3.3.1

[27] A. Friday, N. Davies, G.S. Blair, and K.W.J. Cheverst. Developing adaptive applications: The MOST experience. *Journal of Integrated Computer-Aided Engineering*, 6(2):143–157, 1999. 5.1.3

[28] Globalstar. Globalstar website. `http://www.globalstar.com`, 2001. 3

[29] J Gosling, B Joy, and G Steele. *The Java Language Specification*. Adison-Wesley, 1996. 2.4.2

[30] S. Green, L. Hurst, B. Nangle, and R. Evans P. Cunningham, F. Somers. Software agents: a review. May 1997. 3.4.2

[31] D.P. Grisby. *A Distributed Adaptive Window System*. PhD thesis, University of Cambridge, March 1999. 8.3

[32] GSM. Gsm web site. `http://www.gsm.org`, 2001. 2

[33] G. Hamlin and J.D. Foley. Configurable applications for graphics employing satellites (CAGES). In *Proceedings of the Second Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'75)*, pages 9–19, Ohio,USA, June 1975. ACM. 3.3.1

[34] A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Network*, 8(1), January 1994. 7.5.1

[35] M. Herlihy and B. Liskov. A value transmission method for abstract data types. 4(4):527–551, October 1982. 6.7.4

[36] S. G. Hild. *Managing Mobile Connections*. PhD thesis, University of Cambridge, Sept 1997. 1.1, 3.4.1, 3.5.1

[37] S.G. Hild and P. Robinson. Mobilizing applications. *IEEE Personal Communications*, pages 26–34, Oct 1997. 3.4.1

[38] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Operating Systems Review, SIGOPS*, 21(5):1–2, 1987. 3.5.1

[39] G.C. Hunt. *Automatic Distributed Partitioning of Component Based Applications*. PhD thesis, University of Rochester, Rochester, New York, 1998. 3.3.3, 4.2.2

[40] G.C. Hunt and M.L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI'99)*, pages 187–200, New Orleans, LA, Feb 1999. 3.3.3

[41] IBM. Visualage generator. `http://www.ibm.com/software/ad/visgen/`. 3.3.2

[42] ISO. *ODP Reference Model: Descriptive Model*, January 1995. ISO/IEC Recommendation X.902, International Standard 01746-2. 2.4.3

[43] ISO. *ODP Reference Model: Overview*, January 1995. ISO/IEC Draft Recommendation X.902, International Standard 01746-1. 2.4.3

[44] ISO. *ODP Reference Model: Prescriptive Model*, January 1995. ISO/IEC Recommendation X.903, International Standard 01746-3. 2.4.3

[45] T.T. Jervis and F. Fallside. Pole balancing on a real rig using a reinforcement learning controller. Technical Report CUED/F-INFENG/TR-115, SVR Group, Cambridge University Engineering Department, December 1992. 4.3.5, 4.3.5, 4.3.5, 4.3.5

[46] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings, Fifteenth Symposium on Operating System Principles*, Cambridge MA 02139 USA, December 1995. MIT Laboratory for Computer Science. 3.4.2, 3.4.2, 3.4.2

[47] R.H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994. 3, 2.2

[48] R.H. Katz and E.A. Brewer. The case for wireless overlay networks. In *SPIE Multimedia and Networking Conference*, January 1996. 1.1, 2.2

[49] D. Kimelman, T.Roth, H.Lindsey, and S. Thomas. A tool for paritioning distributed object applications based on communication dynamics and visual feedback. In *Proceedings of the Advanced Technology Workshop, Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, OR, June 1997. 3.3.2

[50] Doug Kimelman, V.T. Ragan, Tova Roth, and Mark Wegman. Partitioning and assignment of distributed object applications incorporating object replication and caching. In *Proceedings of 3rd Workshop on Mobility and Replication, Brussels, Belgium*, July 1998. 3.3.2

[51] E. Koutsofios and S.C. North. *Drawing Graphs with Dot*. AT&T Bell Laboratories, Murray Hill, NJ, USA, November 1996. 7

[52] D. López de Ipiña and S-L. Lo. LocALE: a location-aware lifecycle environment for ubiquitous computing. In *Proceedings of the 15th International Conference on Information Networking (ICOIN-15)*, January 2001. 8.3

[53] D. Lopez de Ipina. Trip for sentient computing. Technical report, Laboratory for Communications Engineering, Engineering Department, University of Cambridge, August 1999. 7.5, 7.5.1, 7.5.1

[54] J.R. Lorch and A.J. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications*, pages 60–72, June 1998. 5.1.3

[55] D.A. Maltz and P.Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of The Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1037–1045, 1998. 2.3.3, 2.3.3

[56] S. Mann. Wearable computing: A first step toward personal imaging. *IEEE Computer*, 30(2):25–32, February 1997. 2.1, 2.1

[57] C. Mayers. *Writing Distributed Applications using ANSA and ANSAware 4.1*. APM Ltd., Poseiden House, Castle Park, Cambridge, UK, June 1994. 2.4.3

[58] D. Michie and R.A. Chambers. Boxes: An experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence*, volume 2, pages 137–152. Oliver and Boyd, 1968. 4.3.5

[59] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applicatons*, San Diego, CA, USA, December 2000. 3.6

[60] P. Naur. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960. 6.2

[61] Network Working Group. *Internet Protocol*, September 1981. 2.3

[62] Network Working Group. *IP Mobility Support*, October 1996. 2.3.1, 2.3.1

[63] J. Nilsson. *Real-Time Control Systems with Delays*. PhD thesis, Lund Institute of Technology, Box 118, S-221 00, Lund, Sweden, February 1998. 4.3

[64] B. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM, Oct 1997. 3.4.3, 5.1.4

[65] B. Noble, M. Satyanarayanan, G. Nguyen, and R. Katz. Trace-based mobile network emulation. In *Proceedings of ACM SIGCOMM'97, Cannes, France*, September 1997. 7.2

[66] B. D. Noble. *Mobile Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1998. 1.1, 2.2, 3, 3.4.3, 3.5, 3.5.1

[67] B. D. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Computing Systems*, 8(4), 1995. Also as technical report CMU-CS-95-119, School of Computer Science, Carnegie Mellon University. 3.4.3

[68] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. The Object Management Group, Framingham, MA, 1996. 2.5, 2.5.3

[69] Object Management Group. The corba services, omg technical document formal/98-0705, 1998. 2.5.3

[70] T. Okoshi, M. Mochizuki, Y. Tobe, and H. Tokuda. MobileSocket: Session layer continuous operation for java applications. *Transactions of Information Processing Society of Japan*, 1999. 2.3.3, 3.5.1

[71] Open Software Foundation. Introduction to OSF DCE. Technical report, Open Software Foundation, Cambridge, Massachussetts, USA, 1992. 2.4.3

[72] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbinger, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999. 3.2.1

[73] Parabon. web site. `http://www.parabon.com`, 2001. 5.1.2

[74] L. E. Parker. Local versus global control laws for cooperative agent teams. MIT AI memo no. 1357, March 1992. 4.1

[75] Popular Power. web site. `http://www.popularpower.com`, 2001. 5.1.2

[76] Real Networks. Real audio player. `http://www.real.com`, 2001. 3.5.2

[77] Jun Rekimoto and Katashi Nagao. The world through the computer: Computer augmented interaction with real world environments. In *proceedings of User Interface Software and Technology Conference (UIST '95)*, Pittsburgh, Pennsylvania, USA, November 1995. 2.1

[78] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, January 1998. 2.6.1

[79] D. Riecken. Intelligent agents. *Communications of the ACM*, 37(7):18–21, July 1994. 3.4.2

[80] M. Satyanarayanan. Scalable, secure and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990. 3.4.3, 3.5.1

[81] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, pages 26–33, Feb 1996. 1.1, 2.6.2, 3.4.3, 3.5.1

[82] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990. 3.4.1, 3.5.1

[83] M. Satyanarayanan and D. Narayanan. Multi-fidelity algorithms for interactive mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM, Dec 1999. 3.6

[84] R.W. Scheifler and J. Gettys. *The X Window System*. Digital Press, Bedford, Massachussetts, USA, 1992. 2.6.1

[85] B.N Schilit, M.M. Theimer, and B.B Welch. Customizing mobile applications. In *Proceedings of USENIX Symposium on Mobile and Location-Independent Computing*, August 1993. 2.6.2

[86] W. Simpson. *IPng Mobility Considerations*. Network Working Group, August 1994. 2.3.1

[87] Sourceforge. Sourceforge web site. `http://www.sourceforge.net`, 2001. 5.1.2

[88] Symbian. Symbian web site. `http://www.symbian.com`, 2001. 2.1

[89] M. Torngren. Fundamentals of implementing real-time control applications in distributed systems. Mechatronics Lab, Dept of Machine Design, Royal Institute of Technology, S-100 44 Stockholm, Sweden, 1997. 4.3

[90] US Robotics Inc. *Pilot Handbook*. Los Altos, CA, USA, 1996. 2.1, 2.6.2

[91] A. van Dam, G.M. Stabler, and R.J.Harrington. Intelligent satellites for interactive graphics. *Proceedings of the IEEE*, 62(4):483–492, April 1974. 3.3.1, 3.3.3

[92] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2):46—55, 1997. 2.5

[93] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999. 6.3

[94] J. Waldo, G Wyant, A Wollrath, and S Kendall. A note on distributed computing. Technical report, Sun Microsystems, 255 Garcia Avenue, Mountain View, CA 94043, USA, November 1994. 2.6.1, 5.1.2, 6.7.4

[95] WAP Forum. *Wireless Application Protocol Architecture*. WAPForum, April 1998. `http://www.wapforum.org`. 2.1, 1, 3

[96] A. Ward, A. Jones, and A. Hopper. A new location technique for the active office. Technical Report TR-97-10, Olivetti and Oracle Research Laboratory, 24a Trumpington Street, Cambridge, UK, 1997. 2.1, 7.5.1

[97] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991. 2.1

[98] P. Wellner. Interacting with paper on the digitaldesk. *Communications of the ACM*, 36(7):87–96, August 1993. 1

[99] WorkSpot. corporate web site. `http://www.workspot.com`, 2000. 5.1.2

[100] H. Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE trans. on commun.*, COM-28:425–432, 1980. 2.4.3