The Implementation of a Distributed Framework to support 'Follow Me' Applications

Pete Steggles, Paul Webster, Andy Harter
The Olivetti and Oracle Research Laboratory,
24a Trumpington Street, Cambridge, England.
{psteggles,pwebster,aharter}@orl.co.uk

Abstract

Follow me applications support user mobility by adapting their behaviour to follow a user as he moves around a building. In order to adapt sensibly to their current environment, such applications need to know about the location and status of people and computing resources. This information needs to be readily accessible and must be up-to-date and accurate. As such a system involves human interaction, this presents some real-time constraints.

This paper describes a framework for supporting such applications. It is based around a 3-tier architecture comprising a centralised database, a middle layer of parallel servers and a collection of distributed clients. Some clients gather information from the environment, and are long running. Other clients are mobile applications, which must respond rapidly to environmental changes. As all these applications are in daily use, robustness and availability are key additional requirements.

The framework uses on-demand loading of CORBA objects and is sufficiently flexible and scalable to be applied to a wide variety of other application areas.

The paper also describes some experiences of deploying a substantial CORBA based system. In particular, it shows how a design evolves due to changing requirements and problems encountered, and illustrates the importance of being able to rapidly prototype systems prior to deployment in order to adapt to these changes and to explore alternative solutions.

Keywords: CORBA, Oracle, distributed computing, 3-tier architecture, mobility

1 Introduction

One of ORL's primary research interests is supporting the mobile user in a home or office environment. The ultimate goal of this kind of work is to allow users to move freely around a building without undue degradation in the computing and communications resources available to them. As the user moves around the building his applications, multimedia streams and telephone calls will all be able to come with him, specialising themselves to their current environment as they follow him.

Such mobile applications require rich information about their environment, gathered from a range of environmental sensors and resource monitors. Typically they need to know about the location of people and resources, and a range of machine properties such as input/output facilities, processing capability and network connectivity. Up to now this information has been gathered and stored by applications using ad-hoc techniques. Though this is temptingly easy to implement for small examples, it is hard to maintain and becomes increasingly problematic as new kinds of sensor and resource are introduced.

The goal of ORL's SPIRIT system [1] is to gather information about the environment and present it in a form suitable for mobile applications. Data is gathered from a range of sensor sources, including the active badge system [2], a fine-grained location system using ultrasound [3], monitors of terminal activity, and monitors of CPU, disk and network activity. These latter monitors run on each machine at ORL and so represent multiple data collection sources. The resulting data is combined with static data about the building, people and equipment, and is presented in an easily accessible form suitable for mobile applications.

This system must fulfil the following requirements:

- The precise mix of information required by new mobile applications is hard to predict, so it should be easy to extend both the data modelled by the system and the queries supported.

- Mobile applications have long periods of inactivity (while users are stationary) punctuated by short periods of high activity (when users move). So, while total query throughput is quite low, individual queries must be performed quickly.

- The data processed by the system ranges from valuable, static data about people and resources, through periodic machine status updates, to ultrasound location data streams producing tens of updates per second. This must all be collated and presented to applications in an easy-to-use framework.

- Applications must be able to run indefinitely, even while the system undergoes continued development work. Applications should not be harmed if the machine(s) running the system go down, for example while data is backed up or the software is upgraded. Some temporary loss of service is acceptable if machines go down.

- The system must scale up to hundreds of heterogeneous machines and mobile users. It should be easy to write client applications; the client-specific code required should be minimal and collected data should be presented in a comprehensible way.

This paper begins with a general overview of the chosen architecture for the implemented

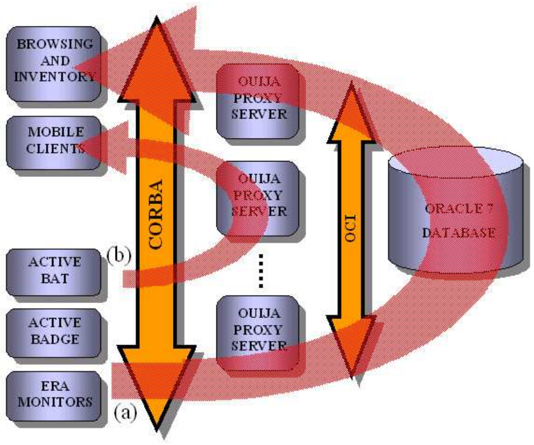

Figure 1: System overview

system. It then discusses the way in which CORBA has been used to allow users and services to be distributed. In particular, the mechanism for providing high reliability is presented. To support large numbers of clients, some filtering of the data is required. The different techniques adopted are discussed, along with some preliminary figures. Finally, some experiences in building the system and deploying it are presented.

2 System Overview

The chosen solution was a three-tier architecture as shown in Figure 1. The bottom layer uses the Oracle 7 RDBMS to provide information persistence. The middle tier features a set of distributed CORBA objects corresponding to real objects in the environment. The top tier of CORBA clients represents the sensor systems feeding the database as well as the mobile applications utilising the stored information. This architecture provides persistence and the ability to make arbitrary queries on the database, yet also includes support for a fast data path that bypasses the bottom tier.

Oracle 7 is a relational database but includes the key features needed to build an object-oriented layer over it. These features include table clustering and stored procedures written in PL/SQL [4]. A simple object oriented modelling language was designed along with a code

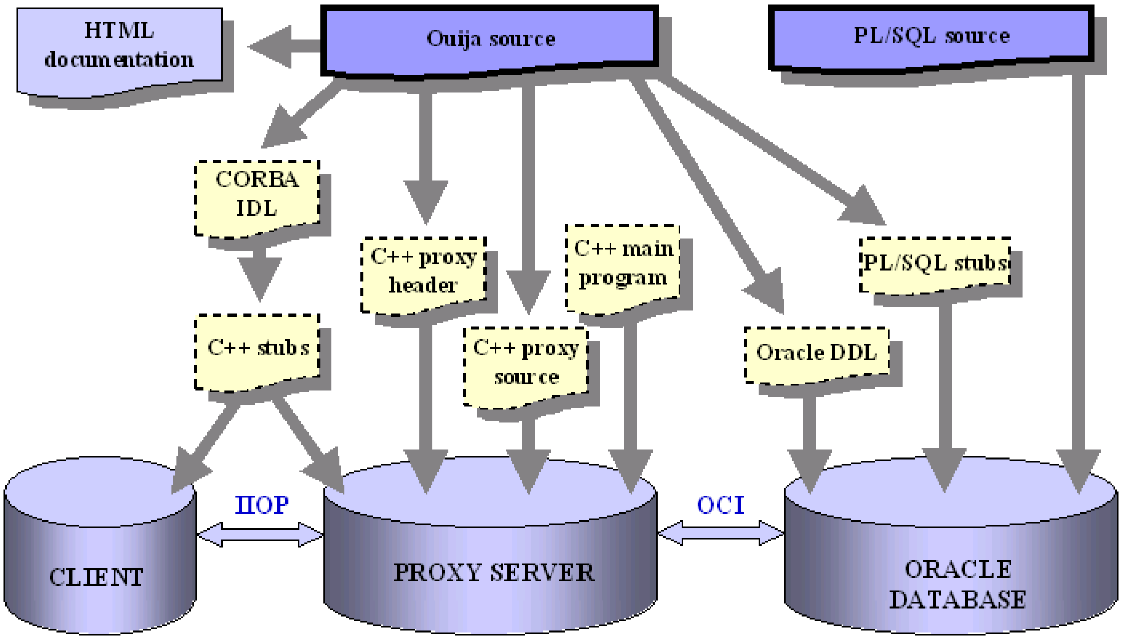

Figure 2: Ouija code generation

generation utility known as Ouija [5]. This automatically generates the Oracle, CORBA IDL and C++ code necessary to create a remotely-accessible object model on the database from a textual description of the data model. The code generation process is illustrated in Figure 2.

Effectively, Ouija provides an object modelling language plus a CORBA mapping for PL/SQL. Objects are represented on the server by sets of database rows; objects' operations are represented by packages of PL/SQL operations. Each object may have a counterpart in the middle tier which acts as a proxy, receiving CORBA calls and forwarding them to an appropriate PL/SQL operation using Oracle's proprietary API [6]. The Oracle DDL for the object model, the PL/SQL package signatures and the middle tier proxy servers are all generated automatically. The basic CORBA objects generated by Ouija have no state apart from a unique object identifier. They can be extended using any combination of C++ inheritance, for example to implement object caching strategies, or by using IDL inheritance, to provide facilities irrelevant to the database.

3 Architecture

3.1 On-demand Loading

Inside the database a persistent object is referred to by a unique identifier. This is made up of a combination of a type code, identifying the objects most derived type, and a unique instance number. This identifier is a primary key for the database table(s) containing the data representing that object.

When a persistent object is made available externally using CORBA, a special object reference is manufactured for it. A standard CORBA reference has the following main components:

1. the repository ID of the most derived type
2. a key uniquely identifying the object within the ORB
3. a networking component; for IIOP this is made up of the IP address and port number on which the ORB is listening

The object reference manufactured for a persistent object sets these components to these values:

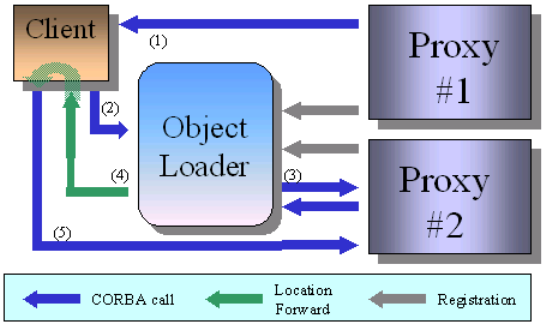

Figure 3: On-demand object loading

1. the repository ID of the persistent object's most derived type (as defined by its type code)

2. the persistent object's unique instance number, as used within the database

3. a well known IP address and port number for a special object loader process which provides the object loading mechanism

Consider Figure 3. When a proxy server starts up it creates a factory for each of the object types which it implements. These factories are registered with the object loader, which maintains a mapping from the type code to the appropriate factory.

Any method which returns an object reference will return a manufactured object reference (*1*). When a method is first invoked on one of these references (*2*), it would normally cause an OBJECT_NOT_EXIST exception in the ORB of the loader as the reference is for a non-existent object. The object loader uses our own ORB, omniORB2 [7], that has been specially enhanced so that any invalid object keys can be passed to a user-defined handler. This routine decodes the key into its type code and instance number components. The type code is used to identify the proxy factory handling the class of objects, which is then sent a load request that includes the unique object instance number (*3*).

The CORBA object corresponding to the identifier is looked up in the address space of the proxy and a new object is created if none is found. The resulting object reference is returned to the loader which sends this new object back to the client (*4*) using a GIOP LOCATION_FORWARD message [8]. On receipt of this message, the client's ORB transparently retries the operation with the new address it has been forwarded (*5*). In addition, if at any time the forwarded reference should become invalid, the client's ORB transparently resorts back to the original object reference. This causes the object to be reloaded.

Objects within the database are therefore only allocated an object implementation in one of the proxies if and when they are first used. This allows the system to support potentially large numbers of objects without committing resources in advance. The proxy is simply a cache of persistent objects; any state which is to be made persistent must be stored within the database. This is because the object may be unloaded from the proxy at any time.

The object loader may appear to be a bottleneck. This is not the case, even though all object references need to be sent to it upon creation. Once the object is created, no further access to the loader is required. In the resource monitoring system the majority of the object references are periodically re-used for database updates by extremely long running clients; the number of object method invocations that use the object loader therefore represent a negligible percentage.

Providing this on-demand loading is the key for achieving the goals set down in the system requirements.

3.2 Robustness

Consider what happens when a proxy crashes. The object reference of the previously loaded object is invalid, so client operations on the object will fail. In this case, the client needs to resort back to the original persistent identifier. This re-contacts the object loader and attempts to re-load the object. As the proxy is down, this will also fail. At this stage, the clients ORB enters a polling loop whereby it sleeps and then retries the operation. Eventually, the proxy will be restarted and the object loader will cause the object to be loaded in the new proxy. While this causes a delay in service availability, all clients continue to run once the proxy service is restored.

3.3 Maintenance

A similar approach provides the ability for system upgrades to take place without disruption to existing clients. A new proxy is started in parallel with the existing one. This registers with the object loader so that any new objects that are loaded will be created using the new proxy. Any existing clients will continue to use the old proxy. At some stage, the old proxy can be killed, at which point any existing object references will become invalid. At the next method invocation, the client will resort back to the original persistent ID and cause the object to be reloaded from the new proxy without any perceived interruption to availability.

3.4 Scalability

The proxies are designed to be long running and the database may contain thousands of objects. Over time, these will be loaded into the proxy and would cause the proxy to consume resources even though only a subset of the objects may ever be in active use. The problem is knowing when it is safe to dispose of an object in one of the proxies. CORBA object references may be stored indefinitely and in a variety of forms including textual off-line references. This makes distributed garbage collection impossible without complex client negotiation. By using on-demand loading, any object may simply be disposed of as and when system resources need reclaiming. Should the object be later referenced, it will be reloaded on demand from the database.

In the implementation, each object loaded by a proxy has an associated LRU counter. This is updated whenever an operation is called on the object. Limits can be set on the maximum number of loaded objects in any given proxy, and the LRU counter will be automatically used to dispose of an existing object prior to loading the new one.

4 Performance

4.1 Proxy Filtering

When dealing with real time updates, there is insufficient storage bandwidth for every transaction to be passed via the database. In these cases, a fast-path for real-time data is provided as shown by arrow (b) in Figure 1. This is used, for example, by a spatial indexing service which is sent the raw sensor streams directly for subsequent processing. The results of this processing may then be stored in the database and used in subsequent database queries.

Real time data may also be stored and retrieved directly from the proxy servers. This ensures clients gain the best response time and the most up to date information. Periodically, the information stored in the proxy can be stored into the database to allow its use in more complex resource queries. In this way, the proxy acts like a write through cache. Should the proxy crash, any cached values will be lost. However, the nature of the data means that once the proxy is restarted, the values that were lost would be out of date anyway.

4.2 Client Filtering

Domain-specific knowledge can be employed to reduce the number of updates sent to the proxies. When dealing with machine monitoring, the types of information to be sent to the database can be broadly classified as state information and resource level information.

State information includes aspects such as the name of the CD-ROM currently loaded in a given drive, and changes very infrequently. The database needs only to be notified of the changes. Similarly, when dealing with a scalar value which represents some resource level, one can define bands which correspond to notional states. The database is only notified when the band of the resource level changes. For example, we could define CPU load bands of 0-30%, 30-70%, and 70-100%.

Client filtering has the disadvantage that the monitored values available will not represent current state. Using CORBA, a feedback chan-

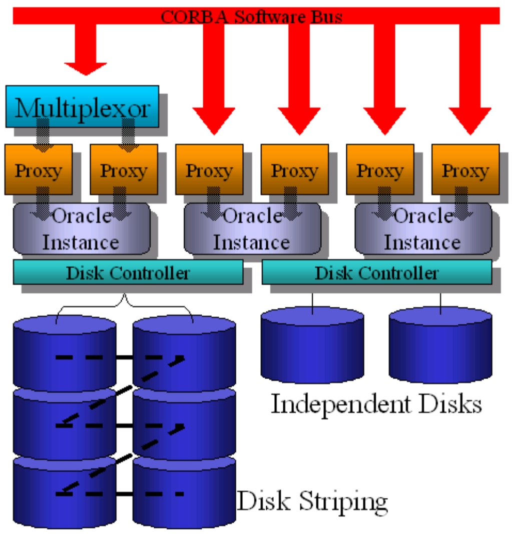

Figure 4: Potential areas for parallelism

nel could easily be added to allow the database to throttle back the number of updates when it is heavily loaded and increase the rate of update when it is less loaded.

4.3 Parallelism

In the cases where filtering is not appropriate, the architecture supports a number of levels of parallelism to allow performance to be improved. Consider Figure 4.

At the most basic level, more disks may be used for the database and they can be organised into a RAID structure to allow data to be striped to improve storage bandwidth. Similarly, multiple disk controllers may be used to provide concurrent disk accesses.

As each proxy is focused on a particular object type, each generally only refers to a subset of the overall data. The Oracle database used in this work supports a parallel server mode, whereby a number of database servers refer to the same database. Proxies can be partitioned amongst the multiple database instances and requests may then be processed in parallel. The on-demand loading mechanism also means that if one database is being particularly heavily utilised, some of the proxies may be moved to one of the other databases without disrupting client access.

The architecture supports one more level of parallelism. A special redirection proxy may register with the object loader to receive proxy object load requests. A number of copies of the same proxy then register with the redirector. When a request to load an object is received, the redirector selects one of the proxies and forwards the request to it. This selection may be on the basis of machine load or take other factors into account such as client geographical location.

4.4 Results

The system is still in the testing stage an has not been fully deployed on a large population of machines and users, but initial testing indicates that performance targets should be achieved.

Currently the database and proxies run on a single processor Sun Ultra-1 workstation (167MHz) with 4 standard 2GB disks.

Using omniORB2 on this machine, the round trip time to echo a null string is around $500\mu s$ for intra-machine communication and around $700\mu s$ for inter-machine communications (using 10 MBit/s Ethernet). A typical Ouija round trip time (client-proxy-database-proxy-client) is 3ms for intra-machine invocations and about 3.5ms for inter-machine invocations, again under the same conditions.

The real time location system generates events at about 25Hz. Necessary computation lags these events by about 40ms. Compared to this figure, the extra lag of about 1ms introduced by distribution using CORBA is not significant.

When attempting to support large numbers of machines, the client side filtering in the monitors manages to generally remove between 80-99% of the updates (about 97% on average). The current system monitors the CPU, motherboard, disks, network cards and processes of lifetime greater than 30 seconds on each machine. This produces, on average, a figure of about 6 fairly small update transactions per minute per machine. The other users of database resources are updates of data about spatial relationships between objects and users, and the integrated spatial resource queries themselves. These loads are quite small in comparison to the update load caused by resource monitoring.

The above figures indicate that it is feasible to monitor at least two hundred machines and users while maintaining good query response times using the current server hardware.

It should be noted that operations are automatically queued in the proxies. This can smooth out the occasional transaction peaks such as under 'start of day' conditions, e.g. after a power cut, but will result in poor query response times on these rare occasions.

Finally, it should be noted that the parallel nature of the overall architecture provides much scope to leverage additional CPUs, disks and network connections on the server side.

5 Experiences

Using the modelling language, an initial data model representing the complete ORL system resources database was written in around 6 months. This was implemented into a working database in 2 months. Since then it has undergone two revisions. The first involved major changes to the complete data model, but took only around 2 months to change the model and implementation. The second involved reorganisation of the member functions and took about 1 month. Clearly, these system changes could not have been made without the rapid prototyping capability provided by the Ouija tool. The current deployed system features 33 entities, 16 relationships and 680 interface methods, all organised as 17 modules. This collection is set to grow as other projects add their own modules to the database.

5.1 Positive Experiences

From this work, it has shown that middleware does work and can be used as the basis for high performance, high reliability applications. The system has been deployed on a number of machine architectures, including Solaris 2.4, Linux, DEC OSF/1, Windows 95 and Windows NT. Long running monitoring clients on each of these machine types have been shown to be robust to network faults and machine crashes. These clients are also unaffected by routine system maintenance, which often involves shutting down the proxies and/or database and restarting them. Since the system was first started, some clients have run uninterrupted for months at a time.

Performance is good, though actual figures are hard to measure as there are so many aspects of the system which are still undergoing tuning. With proxy and client filtering in place, the system can cope with all the monitoring clients and still maintain sufficient response to service mobile clients.

In terms of using the information, the use of CORBA has meant it is very easy to write new clients. Writing the proxy servers is harder, but most of the code is automatically generated. Use of C++ and STL has allowed the source code size to remain small and allows easy extension of the proxies by overriding the generated code.

5.2 Negative Experiences

Not everything went to plan. The original strategy was for user querying and maintenance applications to be written using Java. In practice, this highlighted that none of the currently available Java ORBs fully implement the IIOP protocol [9]. Many did not support heterogeneous architectures by having broken marshaling code that assumed that the endian of values was fixed. None of them correctly supported the location forwarding messages used to support the on-demand loading. Finally, the Java security feature of only being able to contact the network address of the machine which served the application was served from precludes the use of any of the proxy distribution techniques.

Despite considerable investment in trying to circumvent these problems, the Java approach had to be abandoned and CGI was chosen to provide a portable user-interface to the information in the database.

The main disadvantage found with using

CORBA was the size of the stub code generated. The stubs generated by omniORB2 are amongst the smallest of any available ORB, but when a typical monitor refers to the stubs from three or four modules, the stub code still represents around 97% of the total lines of code. Much of this code implements the interfaces for querying the database. With Java no longer being used, most of this is now redundant, so it is hoped the next revision of the model will reduce the code size somewhat.

6 Conclusions

From our experiences, it is clear that using high quality CORBA middleware can provide many important benefits to developers of distributed systems with very few performance costs or other limitations. Performance measurements on omniORB2 [10, 11] indicate that it has a very small call overhead and there is little performance gain by adopting custom communications protocols implemented using low level socket calls. The slight performance hit is more than offset by the much richer RPC abstraction provided, which hides many of the implementation details from client application authors. This includes the problems encountered when dealing with heterogeneous computer architectures and also the provision of support for robustness to network or server failures.

While clients are easy to write using CORBA, the servers are more complex. In this work, this has been addressed by the use of an automated code generation tool which allows much of the process of server construction to be automated. Using this tool, it is easy to prototype new data models and to adapt existing ones to reflect changes in system requirements.

Directions where middleware support could be improved include the provision of fully CORBA compliant Java ORBs and the problem of the size of the CORBA stub code generated. In this latter case, the problem can easily be reduced with the provision of better linkers which can recognise and ignore functions that are never called.

Through the use of CORBA middleware a robust and scalable framework has been produced. This scalability applies to both the number of objects which it can support and to the amount of performance which can gained through optimising the clients and tuning the database. Future work will focus on how to best approach this process in order to meet the performance requirements of current and future applications.

References

[1] SPIRIT home pages. Online. http://www.orl.co.uk/spirit.

[2] Andy Harter and Andy Hopper. A distributed location system for the active office. Technical Report 94–1, Olivetti and Oracle Research Limited, 24a Trumpington Street, Cambridge, England, 1994.

[3] Andy Ward, Alan Jones, and Andy Hopper. A New Location Technique for the Active Office. *IEEE Personnel Communications*, 4(5):42–47, October 1997.

[4] Oracle Corporation. *PL/SQL User's Guide and Reference*, Release 2.3, 1996.

[5] Ouija overview. Online. http://www.orl.co.uk/spirit/ouija.html.

[6] Oracle Corporation. *Programmer's Guide to the Oracle Call Interface*, Release 7.3, 1996.

[7] omniORB 2.5.0 user documentation. Online. http://www.orl.co.uk/omniORB/omniORB.html.

[8] OMG. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, Updated July 1996.

[9] Interoperability between omniORB2 and Java ORBs. Online. http://www.orl.co.uk/omniORB/javaORBs.html.

[10] omniORB performance comparisons. Online. http://www.orl.co.uk/omniORB/omniORB.html.

[11] CORBA comparison project: Throughput test. Online. http://www.kav.cas.cz/~buble/corba/comp/.