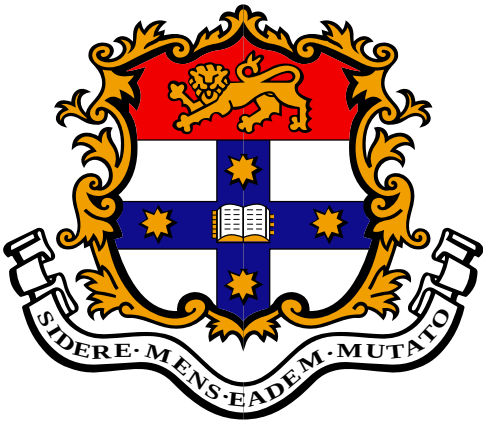
Ph.D. Annual Progress Report for 2003



Daniel Cutting

February 24th, 2004

School of Information Technologies
The University of Sydney

Abstract

This thesis seeks to identify how contextual information available to mobile devices in an ad hoc network can be used to improve the availability of data shared between them.

Contents

1 Problem definition and motivation

1.1 Introduction

This thesis seeks to identify how contextual information available to mobile devices in an ad hoc network can be used to improve the availability of data shared between them.

In the early days of distributed systems, networks consisted of fixed infrastructure. As technology improved, these 'wireline' networks became extremely fast and reliable and distributed systems were able to take advantage of them to make use of many loosely coupled machines in an effective and robust manner [1, 2, 3]. Since they were connected via wireline networks, machines were rarely unavailable and the topology of machines was infrequently modified. Also, since the machines were fixed and had no need to be portable, they were able to be relatively powerful.

When portable machines such as laptops became feasible, distributed systems adapted somewhat to cope with the occasional unavailability of machines [4, 5, 6]. The core of the system however generally remained well connected and reliable.

With the advent of wireless networks, these general assumptions in distributed systems were no longer necessarily valid. With wireless networks, devices were suddenly capable of being truly mobile and problems such as intermittent connectivity were exacerbated. Additionally, since the devices needed to be carried by people, they were necessarily less powerful and feature-rich than their fixed peers. Finally, truly mobile devices enabled the creation of an entirely new class of infrastructure-free networks called mobile ad hoc networks [7, 8, 9, 10].

The proliferation of such devices, in addition to small sensors, advances in human-computer interaction and the general growing ubiquity of computational resources has been collectively termed *pervasive computing*.

It is within pervasive computing environments that this thesis is based, specifically that part related to personal, mobile devices. Much research has gone into low-level routing algorithms for ad hoc networks [11], and some work has also looked at application distribution [12, 13]. Our research concentrates on application distribution, and in particular how *context-awareness* [14, 15] can be used to improve the robustness of applications running in such environments by making the sharing of data between devices more reliable and resilient.

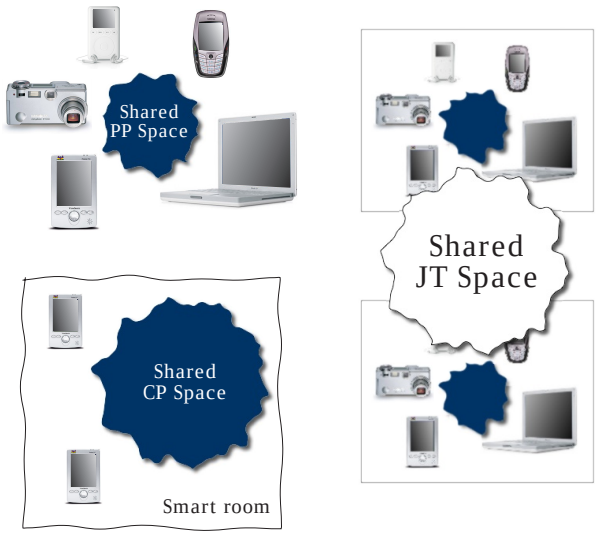


Figure 1: Personal Persistent (PP), Joint Transient (JT) and Communal Persistent (CP) spaces.

The rest of this report is organised as follows: sections 1.2 and 1.3 cover the motivation and objective of the research, section 2 examines the relevant existing literature and research and section 3 describes our initial approach and contributions.

1.2 Motivational scenarios

To scope our research, we define three classes of situations (shown in Figure 1) referred to as Personal Persistent (PP), Joint Transient (JT) and Communal Persistent (CP) and present some motivational scenarios to illuminate them.

The PP class comprises personal devices that are owned by a single user, such as PDAs, music players, laptops, mobile phones, cameras, etc. These devices will generally only be used by the owner and will be a part of the user's

life for relatively long periods of time (weeks, months or years). Hence, the data that are shared between them may be quite long-lived, including music, photographs and documents. In this scenario, the devices involved may be reasonably specialised. For example, a music player is very good at playing music, but not particularly useful for viewing photographs. The shared data should be generally available to whichever devices are able to make use of them. However, since these devices are also generally resource-constrained, it is not necessarily appropriate to store all such instances with the device itself. Additionally, although a laptop may be the most powerful device for playing J2ME Java games, the mobile phone may also have this capability and on occasion it may be expected that the user would prefer to play the games with this device. The game should be accessible to the phone even when the laptop is turned off and secured in a backpack. A calendar application could allow editing and notification of appointments via any of the user's devices with the appropriate input and output modalities. Finally, since this is a personal scenario, we do not wish to openly share data with unknown devices, so a secure way for enabling devices to join and leave the group may be necessary.

The JT class encompasses situations where two or more devices owned by different users come into contact. In such situations, which are typically short-lived (measured in minutes or hours instead of weeks or years), users commonly wish to share data or collaborate on shared resources. An example is a casual meeting in a cafe between a number of colleagues. Three have laptops, two have PDAs and one has a voice recorder. Those with laptops wish to arrange them in such a fashion as to have a large virtual workspace spanning all screens. Those with PDAs jot down various notes and ideas, and the user with the voice recorder begins taping the meeting. Part of the way into the meeting, a late colleague turns up and appends his laptop to the virtual workspace. Towards the end of the meeting the user with the voice recorder is urgently called away, taking her device with her. During the meeting, it should not be possible for people nearby to 'snoop' on the network connections to read sensitive data. Similarly, colleagues should have fine-grained control over what resources and data are made available to the rest of the team. The relatively frequent connection and disconnection of devices should not have a serious impact on the availability of the data; at the end of the meeting, all recordings and notes should be available to all users.

The final class, CP, accounts for typical office situations. When a manager goes to work with her laptop she is able to connect to the distributed file system available throughout the workplace. At midday she attends an offsite meeting but is able to access relevant files from the file system as though she were still connected. While at the meeting she makes some modifications as indicated by the client and these are reintegrated into the file system upon her return to the office. As another example, a user may leave virtual 'sticky' notes around a building [16] with their PDA which other users are able to observe via their PDAs when they are nearby.

1.3 Research objective and approach

In the scenarios enumerated above, the users do not explicitly need to inform the devices of how they wish to share data. Instead, we hypothesise that the distribution of data can be competently achieved via the use of various pieces of low- and high-level contextual information available to the system (described in sections 2.5 and 3). By making use of this context we expect the devices can cooperate to distribute the data in the most applicable way to support the applications.

Specifically, the problem can be defined by the following points:

- the system should maximize the availability of relevant data to applications even in the face of unexpected network or device failures
- the system should minimize the amount of battery usage and network traffic required
- the system should be constrained by codified application semantics and explicit user policies
- the system should make use of relevant contextual information in its distribution algorithms.

We limit the scope of this research to focusing on data distribution in conceptual shared spaces as shown in Figure 1: collections of mobile devices in a personal ad hoc network (PP), temporary shared spaces between possibly untrusted parties (JT), and communication between mobile devices and communal fixed infrastructure (CP).

The objective of the research is to determine how and to what point the availability of data in such situations can be improved by integrating contextual information into the distribution policies, and to identify the types of context and distribution policies that are most relevant to each situational class. Additionally, we aim to find a generalised set of heuristics that is able to cope with all three classes and can be used as a guide in future systems.

Our approach to the problem is to build a generalised data sharing model (aimed at devices shown in Figure 2) that can be used as a testing framework for our ideas. A structure for storing data will be physically distributed over multiple devices and receive contextual information gleaned from several sources including the environment, local sensors, device capabilities, user policies and applications. This information will enable a decentralised distribution algorithm to make decisions on how the structure is actually distributed over the devices. Furthermore, contextual information generated by the structure will be fed back to the applications to allow them to adapt to various situations as they occur (this is known as 'application-aware adaptation' [17]).

With this model, we aim to test low-level aspects such as how various distribution policies (including random and single-node 'server' distribution, full replication and context-aware distribution algorithms) affect battery life and by building sample applications on top of these layers we aim to run high-level experiments that explore general data availability in the face of partial network and device failures. We expect our experiments to run mostly in simulation (possibly using simulation software such as UBIWISE [18]), but it is also hoped we will be able to build a physical prototype of the system that demonstrates the applicability of the model. See section 3 for further discussion of the proposed model.

Figure 2: The model supports devices above the dashed line.

2 Literature review

This section examines the existing research and literature relevant to the problems expressed above. The material is arranged in an order roughly corresponding to section 1.1, covering distributed operating systems and grid computing, distributed file systems, mobile computing (including ad hoc networks and decentralised algorithms), and a survey of different approaches to distributing applications at the application level. The section concludes with a brief look at context and context-awareness.

2.1 Distributed operating systems

The term *distributed operating system (DOS)* can refer to various types of systems. A common definition, known as a *single-system image*, is an operating system executing on a collection of networked machines but appearing to users as a single, unified system. The idea of a DOS is to harness the resources available in many machines to improve general utilisation of resources or make possible applications that are not possible on individual machines.

2.1.1 True distributed operating systems

Amoeba [2, 3] is a distributed operating system which allows multiple computers to act as a 'processor pool' for applications. Running on a tightly coupled multiple computer architecture (i.e. computers linked via a high speed and high reliability network), it provides an environment that is focused on distributing computational load and has a special language called *Orca* to facilitate parallel programming. Other servers in the system provide specialised services such as a shared file system and users connect to the system through graphics terminals.

Sprite [3] takes a more decentralised approach. Instead of users connecting to servers via terminals, Sprite integrates the collection of users' workstations themselves into a complete distributed operating system. When a process is

initiated by a user, it begins execution on their personal workstation. If desired, the process can be migrated to other idle workstations in the system, but this is generally an explicit operation, not initiated by the operating system itself.

Both Amoeba and Sprite simplify the development and execution of applications across multiple machines though both are designed for highly connected and stable networks. In general, such DOSs allow applications to execute without explicitly worrying about the details of their distribution. By using problem-oriented abstractions such as memory, servers, threads, etc. the application can concentrate primarily on the task at hand instead of the details of networking, processor allocation and data distribution. A major drawback of this approach however is that applications do not always have control over the way they are distributed. This means that if the nodes making up the DOS are unreliable, faulty, often unavailable, or malicious, the application may be unable to function correctly. In cases where the DOS is not fault tolerant, it is even possible that the entire system will fail or behave in unexpected ways.

These factors are a consequence of DOS design stemming from the assumed availability of generally reliable, high-speed networks. As the pervasive computing paradigm develops, however, and in particular for the scoped environments pertaining to this thesis, such an assumption is no longer valid. It is now far more common that at least some components in the system will be unreliable or unavailable, due especially to high-latency wireless networks, greatly constrained devices such as mobile phones, and the fact that devices are not necessarily under the authoritative control of any one entity.

Additionally, given that the devices in a pervasive computing environment are likely to be extremely heterogeneous, application distribution using this approach (which generally relies on nodes being more or less equivalent in terms of architecture and power) is not necessarily applicable in these sorts of scenarios.

2.1.2 Distributed virtual machines

A slightly different approach to distributing actual operating systems is to distribute a *virtual machine* running on host operating systems. [19] defines three ways of distributing applications in a virtual machine environment:

- specially compile applications to handle the distribution
- use features of the host operating systems (such as distributed shared memory) to implicitly run the virtual machine over multiple nodes
- create a virtual machine that is explicitly distributed.

Partially in the first category and partially in the second is Jackal [20], which use specialised compilers to turn ordinary source code into code capable of distribution over a shared memory architecture.

The second category also includes projects like Jupiter [21] which executes over distributed shared memory running on the host machines and aims to span 64- or 128-node systems.

Several projects in the third category exist which aim to present a single-system image of a Java Virtual Machine (JVM) across many physical nodes, including JESSICA [22] which provides a shared memory space and preemptive thread migration and Cluster Virtual Machine for Java (formerly cJVM) [19].

Another way to distribute a virtual machine is to have dedicated servers hosting various parts as explored by Sirer et al [23]. As an example, they have implemented a partially distributed JVM which runs on users' local machines but which communicates with a single server acting as the byte code verifier. They claim this design allows for easy maintenance of components of the JVM. However it also means there are bottlenecks in the system, and the system cannot function if servers fail.

Barr et al have developed a single-system image distributed JVM for ad hoc networks called MagnetOS [8]. They have aimed the system primarily at ad hoc and sensor networks, where energy conservation is extremely important. MagnetOS automatically partitions and moves components of monolithic Java applications around a network of nodes with the goal of conserving battery power. To this end, several component placement and migration algorithms have been studied which indicate that the overall omniscience of an operating system with regards to its processes allows more energy-efficient operation in such networks.

The virtual machine approach to distributed operating systems is perhaps more amenable to pervasive computing environments than are true distributed operating systems since it allows heterogeneous devices to run a host operating system that suits them best and still conform to the requirements of a distributed system. Projects such as MagnetOS demonstrate that distributing a complete virtual machine is a feasible approach to distributing applications in some pervasive computing environments. It should be noted however, that the system can still suffer from problems such as network unreliability and unavailability of nodes which can result in overall system failure.

2.1.3 Grid computing

Concurrent with the development of distributed operating systems has been the development of grid computing, known over the years as *"metacomputing, scalable computing, global computing [and] Internet computing"* [24]. Whereas distributed operating systems tend to be designed for relatively localised groups of processors or computers, grid computing encompasses specialised wide-area computing usually tailored for specific scientific requirements.

Some initial grid computing efforts are described in [24] including FAFNER, which aimed to solve RSA encryption challenges by factoring large numbers, and I-WAY which connected several large computing systems with high speed networking for a variety of applications. Globus is a second generation system based on I-WAY, and allows applications to treat a network of distributed heterogeneous machines as a single machine. Globus is composed of many elements that allow resource discovery based on LDAP, a distributed file system based on extensions to FTP, security mechanisms and means of creating and executing applications.

WebOS [25] takes core OS elements such as file systems, security and process management, and distributes them over multiple web servers along with resource discovery facilities which allows for dynamic load sharing of applications. AgentOS [26] is also aimed at wide-area computing, but uses a mobile agent paradigm. Mobile agents are considered a natural fit with wide-area computing since they allow transparent migration of execution units and with correct metrics and migration policies this can result in efficient use of available resources. A relevant feature of AgentOS is its event pool which is a mechanism of event communication between groups of distributed agents. Four permutations of active/passive push/pull communication are identified which allow synchronous or asynchronous notification of events and sharing of data. The mobile agent paradigm has also been applied to localised pervasive computing environments, for example in the one.world [13] and LIME [27] projects (discussed in section 2.4).

Grid computing technologies could be useful in building large-scale infrastructure that supports some elements of pervasive computing environments, such as in CP scenarios and various principles, such as designing systems to handle unreliable components, may also be applicable at the level of mobile devices.

2.1.4 Distributed shared memory

The general idea of *distributed shared memory (DSM)* is to have virtual memory 'chunks' that are accessible from local and remote nodes. It is then possible to run applications using the virtual memory that need have no knowledge of the fact that they are physically distributed. Such systems can be based on several different philosophies including page-based, shared-variable and object-based [1].

Page-based DSM, as implemented in IVY [28], translates the common notion of virtual memory backed by a disk to network-backed virtual memory, where pages of virtual memory are stored throughout the distributed system and 'paged in' when accessed by a process. This approach, while appealing in its simplicity and application-transparency, can result in poor performance when pages straddling several shared variables need to be frequently shipped around the network. This is somewhat intractable when combined with typically dynamic networks in pervasive computing environments.

Shared-variable DSM systems attempt to better serve the needs of the application (and reduce network usage) by breaking up the chunks into more appropriate logical units. For example, Munin [29] makes use of the types of data stored and their access patterns and allows different coherency protocols based on these factors.

Finally, object-based DSM systems rely on highly structured memory chunks that manage access to the memory.

Middleware systems such as Linda, described in section 2.4.1, can fit into this classification.

2.2 Distributed file systems

Strongly related to distributed operating systems are *distributed file systems (DFS)* which allow the long term storage and sharing of files between nodes of a distributed system. As a means of sharing data between applications, file systems can be useful. File-based communication between processes can be indirect, with the producers and consumers not needing to know of each other's existence or even needing to run concurrently. By employing distributed file systems, applications running on separate machines can execute as though all computation were local, through familiar file system operations.

Some distributed file systems focus on exporting remote file systems to local machines over relatively stable networks. Such systems, known as *network file systems*, include NFS [30] and Harp [31]. Client-server distributed file systems such as Andrew [4] use the notion of trusted servers to provide a convenient way to access the same files from many remote clients, appearing as a single file system regardless of the number of nodes involved. However in large installations, availability can suffer when a server or network connection fails. This problem has led to distributed server pools such as Frangipani [32] which use sets of trusted servers to store replicas of files, improving availability. Coda [4, 5, 17] extends the trusted server pool approach by supporting short periods of client-server disconnection, complete with reintegration of client modifications upon reconnection. Odyssey [6, 17], JetFile [33] and MFS [34], although still somewhat client-server in philosophy, are designed to support poor network connectivity as the normal case. Farsite [35], in contrast to the client-server approach, fully distributes files over a set of untrusted workstations.

The original UNIX supported only local file systems. This changed with the introduction of VFS (virtual file system) which defines generic file system and file operations as distinct from specific file system implementations. The NFS protocol [30], built on top of VFS, is a server-oriented file system protocol that allows client machines to mount specific remote hosts' file systems into their local file system. Once mounted, clients do not need to explicitly deal with server names and can access the files as though they were local. The focus on remote file access distinguishes NFS more as a network file system than a distributed file system and it is not intended to improve availability of files but to simply allow the sharing of files between nodes. Remote file systems can be mounted in both 'hard' and 'soft' states where hard mounting means that all operations on the remote file system block until completion (even if the server dies and is restarted) and soft mounting means the operations can eventually time out with an error. Built on RPC, NFS is stateless which means no server needs to maintain any information about its clients as all RPCs contain enough information to service a request (and therefore server crash recovery is trivial). The original NFS did not support file locking and concurrency control was lacking (multiple processes could write to the same remote file and the writes would be intermingled). NFS has been widely deployed and works well in situations of high network connectivity and where file access is not excessive.

Harp [31] makes use of the NFS protocol but provides replication of files on multiple servers so as to improve availability. In this sense it is a combination of a network and distributed file system. As with ordinary NFS configurations, remote file systems are mounted locally. However, in typical NFS configurations, the failure of a server means clients need to wait for it to recover or handle the fact that the files are unavailable. Harp attempts to rectify this by using a primary copy mechanism which uses a primary server and a set of backup servers. When the primary server fails, clients are able to continue to operate by contacting a backup server. After some period, the failed server can be repaired and reintegrated into the server set. One of Harp's novel features is the combination of a volatile modification log and a UPS backup device on each server. The log is applied to the file system in the background which means that the operations are generally fast. The UPS protects against power failures allowing the log to be flushed to secondary storage in such an event.

The Andrew file system [4] is designed as a centralised pool of file servers. It is made up of two major components, Vice (the collection of trusted file servers), and Venus, a process residing on each client that handles the communication with Vice and caches files locally. Andrew has been designed to allow location-transparent file access and trivial data sharing between remote workstations. Due to the relatively small set of centrally administered servers, security and data integrity are fairly easy to implement. Files are stored in the server pool, but are migrated in their entirety to the client upon opening. Subsequent file operations such as read and write are thus as quick as local file operations. When

the file is closed, changes are shipped back to the pool. Andrew went through several iterations: AFS-1 was the initial prototype implementation and suffered from poor performance when heavily loaded partially due to its pessimistic client caching policy which had clients contacting the servers to ensure cache consistency upon every open operation. AFS-2 and AFS-3 use optimistic policies whereby any client that caches a file agrees to notify the server when it is to be modified. Beyond a certain number of nodes however, Andrew suffers from performance problems and since files are not replicated, the failure of a particular node can cause widespread disruption throughout the network due to file unavailability.

Frangipani [32] also uses the notion of a centralised file server pool. This pool is made up of an arbitrary number of servers running the Frangipani file system layered on top of a shared disk abstraction called Petal which manages distribution of data over many physical nodes. Petal is able to robustly replicate data over many servers which greatly improves availability compared to systems like Andrew. Also, it is easy to increase the amount of storage available simply by adding new servers to the pool without explicitly configuring the locations of files and directories. Adjacent to Petal is a distributed lock service which is used by the file system layer to synchronise access to the Petal virtual disk.

Drawbacks with the Andrew file system have led to the development of Coda [4, 5, 17], a file system more able to cope with environments that entail common node and network failures. Although Coda has similar basic client-server design principles to Andrew, it has been designed from scratch to allow users to continue working on files even in the face of node and network failures. When a client becomes disconnected from all Coda servers, it enters a 'disconnected' mode and relies instead on the versions of the files it has cached. With appropriate prefetching and aggressive caching policies, this can actually produce acceptable operating conditions. Modifications and access to cached files are simply logged and affected locally until the client is again able to access the Coda servers (reentering 'connected' mode) at which point the log is transmitted, and the changes committed globally. The logical extension of this means intentionally disconnected client operation is possible. For example, a user can take a laptop home over the weekend, work on cached files and have them automatically reintegrated upon their return to work. This approach to disconnected operation is called *application-transparent adaptation* [17] and means applications do not need to worry about the state of the network.

While the Coda approach is quite successful for reasonably stable networks with relatively infrequent communication interruptions, it is not as successful in more dynamic domains, such as highly mobile pervasive computing environments. To address the difficulties of mobile domains, where the quality of networking is frequently sub-optimal, the Odyssey system [6, 17] introduces the concept of *application-aware adaptation* [17]. Whereas Coda attempts to hide the failure of nodes or network links, Odyssey instead informs the applications of network quality through callback mechanisms. Applications can thus adapt to their connectivity which is appropriate for certain types of applications that rely on timely delivery of potentially lossy data, such as streaming audio and video where the quality of data can be reduced as the network links degrade. Although applications could potentially determine such situations themselves, the ad hoc nature of this approach means applications are harder to create and would not necessarily perform well in competition with one another. Having the system coordinate the applications means they can be coerced into cooperating with regards to resource allocation.

MFS [34] is also aimed at highly variable network connectivity. Like Coda, it uses a centralised server pool to maintain primary copies of files, and caches working versions locally but instead of entering various modes (i.e. connected or disconnected), it maintains a set of priority queues which store operations and data to be asynchronously transmitted back to the server in the background. This technique, called *asynchronous writeback* means bandwidth is generally used more fully in situations where connectivity has only barely triggered a state change in Coda, and overall utilisation is increased. Another novel aspect of MFS is its cache consistency algorithm. In Coda, Andrew, Frangipani and other file systems, there is only a single type of cache consistency. That is, a file follows a caching policy regardless of the type of file or access patterns. MFS, however, allows different files to be cached in different ways. In particular it specifies "private" and "shared" files. Private files, which include home directory files, etc. are cached optimistically (as with Coda) and committed to the server with less priority. Shared files however are updated and committed with high priority. Files can be automatically classified as private or shared based on profiles or access patterns.

JetFile [33] is a file system designed for wide area distribution at various network speeds ranging from dial-up modems to gigabit links. It is not completely decentralised, as it maintains several servers for operations such as file versioning and security management, however the files themselves are cached aggressively around the entire network. A novel

feature of JetFile is the employment of IPv6 multicast as a means of distributing a file's data and metadata. Any servers that are interested in a particular file join that file's multicast group (there can be several such groups for each file corresponding to different update speeds for use in slower networks) and listens for updates and requests. Thus, reading a file can be very efficiently satisfied by any server that has a cached copy of the file and is in the multicast group. Another relevant feature is that prefetching policies can be dictated on a network by network basis. High speed networks could prefetch files and related files, while low speed or expensive networks could avoid prefetching all together, delivering only those parts of the file explicitly requested. Since JetFile relies on some centralised servers it is able to avoid the problem of trusting unknown or malicious entities to some extent. Each file in JetFile can be protected with public key cryptography and validated with a checksum to ensure against tampering.

Farsite [35] moves closer to a fully decentralised file system and is aimed at optimising file availability. It is targeted at making use of free space on desktop machines around an office, with the assumption that nodes are generally available and network connections are reasonably reliable. Farsite employs a hill-climbing algorithm to improve replica placements over time. This implies that the network topology needs to remain relatively stable.

Finally, the cooperative file system (CFS) [36] is a read-only file system built using the distributed hash table, Chord [37]. Like FarSite, CFS does not rely on any centralised servers but instead distributes storage around a network. It is resilient to node failure (or disconnection) as it replicates small segments of files on different nodes, which also helps to distribute load when files are accessed. Being fully decentralised it is adaptive to changes in network topology although since it is based on distributed hash tables, it is not necessarily amenable to mobile ad hoc networks. File integrity is maintained through public key encryption. CFS, however, is not as flexible as a typical file system as it does not support multiple writers to a file. Files are owned by individual nodes and may only be updated by that node. They can however be read by any other node with permission to do so. The absence of multiple writers allows a great many assumptions, particularly regarding cache consistency.

In summary, distributed file systems are generally well understood and are well established in situations where network topologies are relatively stable. There has been some progress in supporting more unstable pervasive computing environments with systems such as Coda, Odyssey, and MFS all capable of supporting disconnected operation in mobile devices to varying degrees. However, most still rely on centralised servers which, while acceptable in CP scenarios, limits their usefulness in supporting applications for PP and JT scenarios. Additionally, file systems are conceptually more persistent entities than is perhaps required for the JT scenario and do not necessarily provide the most appropriate data sharing semantics in all circumstances. However, many of the ideas underlying distributed file systems, such as replication algorithms, are undoubtedly relevant to this thesis.

2.3 Mobile computing

Mobile computing refers to the use of mobile devices such as PDAs, mobile phones, and laptops with wireless communication capabilities in place of traditional fixed wireline devices. In some cases fixed infrastructure can assist mobile users, as with wireless access to Internet gateways, but in other cases, mobiles devices may communicate directly with one another, forming ad hoc networks.

In [17], Satyanarayanan outlines four features of mobile computing:

- devices are resource-constrained
- devices are open to security threats (easy to steal, etc.)
- device connectivity is variable and unreliable
- devices need to run on batteries.

These features should be considered when researching any mobile system as they have a profound influence on design. Perhaps of highest concern are the third and fourth points; in wireline systems, it is usually assumed that devices will be available and responsive at any time. In mobile situations however, systems must be fundamentally based upon the fact that connectivity is wireless and devices run out of power.

The remainder of this section cover fundamental aspects of mobile computing that are needed to form the basis of such systems.

2.3.1 Ad hoc networks

Ad hoc networks are networks of devices that are formed spontaneously without relying on fixed infrastructure and are most commonly connected with wireless network technologies such as Bluetooth [7] and WiFi [1]. Bluetooth is a low-powered, low-bandwidth solution that is designed to reduce the need for physical cables in resource-constrained devices. It has a range of approximately ten metres and many small consumer devices such as mobile phones and PDAs now have built-in Bluetooth chips, allowing the creation of *personal area networks (PANs)*. WiFi is a higher-powered wireless technology designed more for devices such as laptops, allowing connection to fixed infrastructure services such as Internet gateways. However, WiFi is also capable of operating in an ad hoc mode and due to its far higher bandwidth, is a technology worth considering in networks looking to exchange large amounts of data such as music and video.

Due to the nature of wireless technologies, the physical location of nodes is far more important than in wireline networks. In contrast to the latter, where nodes are connected in whichever graph topology is most suitable, wireless signals are broadcast from points in space and only have a limited useful range. Consequently, the topology of ad hoc networks changes frequently as nodes move and the techniques for routing messages from node to node are quite different to their fixed counterparts. Many ad hoc routing protocols have been devised which allow the routing of data between nodes that are not directly within broadcast range of one another. These can be classified into two particular strands, Table-Driven routing protocols such as Clusterhead Gateway Switch Routing (CGSR) and On-Demand protocols such as Ad-hoc On-Demand Distance Vector Routing (AODV) [11].

In situations where all nodes are assumed to be within broadcast range of one another (i.e. within one 'hop'), such protocols are not necessary as all packets can be seen by all members of the network. Since this thesis is focusing on generally small networks in PP, JT and CP configurations, it is possible we will not need to concern ourselves too deeply with the particular network protocols employed. We do however, need to at least consider how devices with different wireless technologies can communicate seamlessly. For example, a mobile phone with Bluetooth could communicate with a PDA that uses WiFi by bridging through a laptop with both technologies.

It may be possible to hide some of this complexity with the use of a system like BASE [38] which aims to support a variety of networking protocols via plugins and presents a uniform interface to device capabilities. This additional layer of abstraction may, however, make it more difficult to pursue our goal of minimising battery and network usage.

2.3.2 Service discovery

Above the network protocols, it will be necessary to discover the resources available within the network including the devices present, their capabilities and the higher-level constructs such as shared spaces. Again, BASE [38] proposes a means of exposing the various device capabilities within a network and this may be satisfactory for the scope of this thesis. However, if more functionality is required it may be instructional to examine service discovery protocols such as SLP [39].

2.3.3 Decentralised algorithms

The peculiarities of ad hoc networks, and in particular, their transient topologies, spatially limited connectivity and need to conserve energy, mean that some decentralised algorithms that work extremely well in wireline environments, such as distributed hash tables [37, 40], do not work nearly as well in wireless situations. As a result, many such algorithms need to be rethought in the context of ad hoc networks.

For example, a useful mechanism in distributed systems is one that allows mutual exclusion among processes. This

[1] http://grouper.ieee.org/groups/802/11/

enables distributed applications to enter critical regions of code whilst ensuring that no other process is altering pertinent data elsewhere. Several decentralised algorithms exist (such as Ricart and Agrawala's algorithm [1, p135-138]) but to provide better network- and energy-efficiency, [10] offers an algorithm based on the merging of two common approaches to the problem, *token-asking* and *logical ring*. In this algorithm, any device holding the sole token can execute a critical region after which it is passed in a logical ring amongst the remaining devices. The novel aspect is that passing the token around the loop is only initiated when a device needs the token, thereby conserving energy by reducing network traffic.

Another technique that is useful in mobile domains pertains to replicated objects. There are many reasons for replicating (or *caching*) objects around a network. Specifically, it allows more efficient read access, since consumers can obtain replicas of the object from a node that is closer than the producer and it provides robustness, since even if the original node fails, replicas will still be available from other nodes. The main problem with caching is maintaining consistency between replicas in the face of updates. When an object is updated, all replicas need to be updated also. This can be achieved in many ways, depending upon the required semantics. In some applications, such as web browsing, it is not vitally important for replicas to be completely up to date. In others, such as databases, it may be very important that updates are atomic for the whole system. The problem can be complicated further when there is not just one writer, but several, working on an object. It can be surmounted by selecting a single node as the point through which all updates must flow (the *primary-copy scheme*), but this raises reliability, efficiency and availability problems.

[9] approaches this problem through a distributed voting protocol that allows nodes to vote upon updates. When an update has a majority of the vote, the update is considered complete. In such an algorithm, updates flow through the system slowly and are not atomic, but the approach does work well for certain classes of applications. Other systems such as Bayou [41] and XMIDDLE [42] employ different techniques (discussed in section 2.4).

2.4 Mobile distributed applications

Instead of avoiding distribution at the application level by building on transparently distributed structures such as file systems and operating systems, an alternative approach is to explicitly distribute application components and provide them with means of communicating. There are various ways to do this including sockets, RPC/RMI and libraries that abstract the communication into more manageable concepts such as the Message Passing Interface (MPI) [2]. Additionally, communication can be accomplished through a middleware system.

In general, middleware acts as an intermediary between applications or application components. This is often accomplished by converting data into a common format and providing clearly defined operators for dealing with the data. Explicit formal semantics are usually specified so that application developers can think about the system in a simplified and consistent manner. For example, the middleware system LIME [27] has semantics expressed with the formal mobility language, Mobile UNITY [43].

Existing research has shown it is often useful for middleware to be referentially and temporally decoupled [44, p34–36], so that messages can be passed between clients that are not aware of each other's name or even of their existence. This allows for robust distributed applications, as components can be plugged into the middleware without needing to explicitly reconfigure other components.

2.4.1 Tuple spaces

Linda [45] is a coordination language (a specialised form of middleware that enables synchronisation between parts of applications in addition to pure data sharing) for easily distributing applications and has been implemented both as an extension to many languages and as distinct systems. In essence it uses a global, shared tuple space paradigm and defines three operators for dealing with the space. A tuple is a simple, ordered, record-like data structure of the form $< a, b, ... >$, and a tuple space can be thought of as a shared bag (or multiset) of such tuples. Tuples can be constructed from both *actual* and *formal* elements, where actuals are literal values, and formals are types. An *anti-tuple* or *template* is a tuple that can be used to match against another tuple. Matches are made when the number of

[2] http://www-unix.mcs.anl.gov/mpi/mpich/

elements of two tuples are equal, all actual elements are equal and all formal elements match the type of corresponding actuals. The basic operators in Linda are:

- OUT(*t*) places the tuple *t* into tuple space
- IN(*t*) retrieves a tuple matching the anti-tuple *t* from tuple space. This operator blocks until a match is found.
- RD(*t*) copies a tuple matching the anti-tuple *t* from tuple space. This operator blocks until a match is found.

In addition to these, EVAL(*t*) spawns a new process to evaluate the elements of *t* before it is placed into the tuple space.

As an example, two processes could share a variable 'x' in the following way: Process 1 (P1) would call OUT($<$ 'x', 5 $>$) which would place a 2-tuple containing the actual character 'x' and the actual integer 5 into the global tuple space. Process 2 could call IN($<$ 'x ', ? i $>$) (where ?i indicates a formal field of type integer) at any time before or after P1's operation to then read the value. Since the IN operator is blocking, P2 would simply wait until the matching tuple became available before removing it from the tuple space and substituting the literal 5 into a local variable. The variable 'x' has thus been shared between P1 and P2. If a third process (P3) were to call IN($<$ 'x ', ? i $>$) concurrently with P2, the tuple would arbitrarily go to either P2 or P3, while the other continued to block for another matching tuple to be placed into the tuple space.

Over the years, many additions have been made to the basic Linda concept including multiple or partitioned tuple spaces [46, 47] and probe operators, INP and RDP, that are essentially non-blocking versions of IN and RD. Linda has been successfully distributed over multiple nodes in 'classical' distributed systems [48, 49].

Synchronisation of processes in tuple spaces is provided by the blocking nature of the IN and RD operators. Additionally, since the tuples are *anonymous* (i.e. there are no explicit identifiers of which process created the tuple) and *persistent* (in that tuples within tuple space can survive the termination of the process that created them), the tuple space paradigm allows decoupled communication between processes.

Limbo [50, 51] is an attempt to use the tuple space paradigm in a mobile computing environment as it allows communication that is sporadically disconnected both in time and space. This, it is argued, maps well to the nature of mobile networks as opposed to earlier designs that relied on more established concepts such as remote procedure calls (RPC) and explicit socket-based connections between processes. This work demonstrates that although the tuple space paradigm is well suited to mobile computing environments, it does not in and of itself provide a particularly good means of measuring and modifying the quality of service (QoS) in such situations.

LIME [27] is another implementation of Linda in a mobile environment that attempts to address some of the problems caused by mobility. It is designed to run on small mobile devices and provides automatic merging and separation of tuple spaces as devices come within range of one another. It includes several novel features distinct from ordinary Linda tuple spaces, notably the concepts of *reactions*, probing (i.e. the INP and RDP operators) and group matching operators. Reactions extend the operators available to allow the developer to specify a template tuple and a callback to be executed when a matching tuple is inserted into the tuple space. This is similar to a subscription in a publish-subscribe system (see section 2.4.2). LIME has been used for several demonstrator applications and has also been used as the basis of a transparent Java class loader [52] and a service discovery system [53].

The Event Heap [44] (originally implemented on top of IBM's T Spaces [54]) uses tuple spaces as a basis for its model. It is part of the iROS operating system [55] and is intended to act as middleware within an interactive workspace. Its role is to connect the various devices within the workspace in an attempt to decouple application logic and thus increase robustness and decrease interdependency. The Event Heap's storage abstraction is an 'event', though it is different to events as commonly found in publish-subscribe systems such as Elvin [56] (see section 2.4.2) and more closely resembles a variation of tuples. Events can persist or expire after a time limit. The system supports reactive subscriptions to events and persistence of events, making it a sort of tuple space/publish-subscribe hybrid. However, it is designed to act as a physically centralised service running on a reasonably powerful server and cannot be easily distributed across the devices that make use of it.

The MAGNET [57] project aims to allow applications to automatically adapt to changing environments. Rather than focusing on low-level QoS problems such as intermittent connectivity, MAGNET defines QoS to include available

resources and capabilities within the environment such as whether a printer can produce colour output. The system is designed to allow users to move portable devices from location to location and have them automatically configure themselves to take advantage of local services that suit the applications. It achieves this by using tuple space structures called *information pools* and enabling applications to provide their own tuple-matching functions that can make use of environmental context. Each device runs a local pool, managed by a MAGNET Server and these servers communicate with remote servers to create distributed pools.

There are many other tuple space systems, including T Spaces [54], JavaSpaces [58] and LuaTS [59]. Most support similar features to those described above.

2.4.2 Publish-subscribe systems

Publish-subscribe systems differ from most tuple spaces in that the structures placed into the shared space do not persist. Rather, they are compared to a stored set of *subscriptions* and when a match is found, are passed on to the clients who posted the subscriptions.

The typical operators for a publish-subscribe system are:

- SUBSCRIBE(*subscription*)
- UNSUBSCRIBE(*subscription*)
- PUBLISH(*event*)

A key feature of publish-subscribe systems is the subscription language which determines how matches between subscriptions and events are made.

In Elvin [56] for example, a subscription typically has the format "REQUIRE(x) && $x > 5$" and an event might be a key/value pair indicating that the key 'x' has value '6'. In this case, any client who subscribed with the above subscription would receive the event. If, however, the key/value pair 'x' and '4' were published, they would not be notified. Elvin's subscription language allows variable types including integers, floating point numbers and strings and logical operators such as AND and OR. Elvin uses a physical client-server architecture. Subscribers and clients connect to an Elvin server and periodically send it subscription requests and event notifications. Whenever an event of interest to a client is published, the server sends a notification. The Elvin server thus acts as an intermediary between a publisher and any number of subscribers. For reasons of scalability and security, Elvin also provides a *federation protocol* which allows multiple Elvin servers to federate into a single routing network. Elvin also supports a novel feature called *quenching* which essentially allows producers to register with the server that they would like to be notified when subscriptions relevant to their publications are registered, unregistered or modified.

2.4.3 Blackboards

Blackboards originated as a means of solving complex problems using the divide-and-conquer approach. The concept stems from the analogy of a group of experts collaborating around a blackboard to solve a problem. As one expert solves part of the problem and writes some results on the blackboard, another is stimulated to solve the next part based on the existing partial solutions [60].

Blackboard systems have three main components [61]:

- Knowledge Sources (KSs) which are the 'experts' and can solve parts of the overall problem
- the blackboard which is a shared repository for full and partial solutions
- a controller which schedules the next KS to run.

The concept is to have a controller that knows about the various KSs and is able to schedule their execution to progress towards a final solution. When a KS is installed on a blackboard, it notifies the controller of the types of data in which its interested. When such data appears on the blackboard, the controller considers the KS for activation (possibly along with other KSs who have registered for similar conditions). The shared space of the blackboard allows various KSs to read sub-problems they are capable of solving and write back results or partial results having manipulated that data. Hence the KSs communicate only indirectly via the blackboard using an agreed format. This powerful concept means very complex problems can be solved in discrete chunks and reduces the explicit relationships between program modules.

This is the model employed by the Hearsay-II speech understanding project [62], a system that attempts to convert audible speech into the intention of the speaker. Various KSs attempt to interpret the input in various ways and place their hypotheses on the blackboard which other KSs use as the basis for future hypotheses.

More recent blackboard systems, such as the MICA [63] project, have refined the concept by removing the controller component and replacing the KS components with concurrently executing intelligent agents. MICA also structures the data placed on the blackboard by typing objects in a similar way to object-oriented programming. The objects on the board (called *mobs*) can extend other types of objects and make it simpler for agents to observe all objects placed on the board of a particular type or any of its derivatives. There is an SQL-like query language that retrieves objects from the board and a time-to-live mechanism that expires objects automatically after certain periods if desired. The blackboard is intended to be used both for long-term storage of objects as well as short-term message passing between agents. Although there is no explicit concept of 'events' as with the publish-subscribe paradigm, agents can register with the blackboard to be informed of the creation of specific types (or derived types) of objects.

2.4.4 Toolkits

Rover [12] is a toolkit for building mobile applications. It is based on a traditional client-server approach with two major differences: relocatable dynamic objects (RDO) and queued remote procedure calls (QRPC). Intended for environments that still have a fixed server infrastructure, Rover allows parts of applications to be migrated to mobile devices in the form of RDOs and asynchronously queue RPC calls via the QRPC mechanism. Together, these essentially enable applications to execute in a range of ways from fully client-server, to fully local; the onus is on the application developer and the network connectivity to decide how the application is actually split.

Bayou [41] is designed to maximise the availability of data within mobile environments and has been designed specifically with mobile computing in mind. To ensure that any client can read and modify shared data, even when completely disconnected from all other devices, it provides *read-any/write-any* semantics meaning devices can always read and write data in which they are interested. This requires liberal replication of data and as such, Bayou provides only weak consistency between these replicas around the network. It employs an anti-entropy peer-to-peer protocol for propagating updates: when nodes receive updates, they integrate them into their working copy (thus 'reducing entropy' in the system). Two types of update conflict are possible in such scenarios, *write-write* conflicts where two incompatible writes occur to the same data, and *read-write* conflicts where a node updates some data based on the value of some other stale data. These conflicts are resolved in Bayou through application-specific resolution policies. To help applications maintain some sort of sensible operation in the face of weakly consistent data, Bayou provides *session guarantees*. These can guarantee, amongst other things, that any node within a session will only read its own updates. Due to the nature of weakly consistent updates, it is possible for divergent update paths to appear throughout the network. This is alleviated by occasionally *committing* a particular update as the definitive version, achieved by retaining the notion of a primary server responsible for deciding which updates are valid and in which order. This sacrifices the fully decentralised nature of the system in exchange for a valid, though weakly consistent, update mechanism.

XMIDDLE [42] is built on the notion of common disconnection of devices that wish to read and write shared data. Instead of managing replication of discrete objects as in Bayou and others, XMIDDLE structures data hierarchically in an XML tree. Also unlike Bayou, XMIDDLE does not promote epidemic replication of data to peers: data must instead be explicitly *exported* by its owner and *linked* by those wishing to share it, much like a network file system. When nodes are disconnected, they are free to update cached copies of this data and upon reconnection it is reintegrated using application-specific metadata.

The one.world [13] architecture is designed to simplify the development of applications for use in pervasive computing environments. It is based on three principles: *expose change*, *compose dynamically*, and *separate data and functionality*. It is argued that since these sorts of environments are so dynamic, change should be embraced instead of hidden with abstractions. To this end, the platform makes heavy uses of *leases* which are essentially renewable handles to resources. The system also advocates a hierarchical design philosophy such that components can comprise other components which makes dynamic composition of applications at run-time relatively straightforward. Data and code are separated with a tuple space abstraction - all data in the system is in the form of tuples, and all code operates on the tuple space through a *structured I/O* interface. This interface, while providing much of the functionality of tuple spaces, is conceptually closer to a communication channel, and also allows the transmission of events to elements of the application. Replication of data throughout the system is not intrinsic to the system, but implemented at higher layers. One such implementation is based on two-tier replication. This policy maintains a master copy of the data on a particular node with replicas distributed to other nodes. When these nodes are connected, updates are handled directly by the master, but when disconnected, clients can continue to make modifications which are reintegrated upon reconnection. This is in contrast to Bayou's epidemic replication policy, where groups of nodes disconnected from the master can synchronise independently and possibly diverge from the 'true' state.

2.5 Context

Much has been written in definition of the term *context*:

> "...precise location information ...and direction, altitude, light, and humidity measurements (via sensors)" [64].
>
> "Context in human-computer interaction includes any relevant information about the entities in the interaction between the user and computer, including the user and computer themselves." [65].
>
> "...information sensed about the environment's mobile occupants and their activities ..." [66].
>
> "The term 'context' is quite broad and usually used to include both simple (or atomic) context, e.g. the user's current latitude and longitude coordinates as sensed using a GPS device, and interpreted (or composite) context, e.g. a prediction of the user's current activity based on a variety of sensory information such as location, heart rate, proximity to others etc." [67].
>
> "...relevant context information includes the capabilities of the mobile devices, the characteristics of the network connectivity, and user specific information." [15].

In general, the term context can refer to any piece of information in some way relevant to a task. More specifically, it can include bottom-up data such as reported by or aggregated from sensors and inferred situational data, as well as top-down data such as users' preferences and input.

Contextual information is often incorporated into applications in an ad hoc fashion [65]. For example, the developers of a location-dependent application may use location context by explicitly reading the values from an attached GPS device. This ad hoc approach, while initially simple, makes it hard to reuse notions of context in other settings.

As such, some attempts to formalise the descriptions of contextual information have been proposed. [15] identifies six features that are necessary for such a representation:

- structured: a structured representation simplifies operations on the information
- interchangeable: the information must be transferable between entities that wish to use it (this essentially amounts to requiring the format to be capable of serialisation)
- composable/decomposable: this means the representation does not have to be self-contained
- uniform: if a representation is uniform, it simplifies the integration of new devices and concepts into existing applications

- extensible: to cope with future additions to the representation without modifying existing applications
- standardised: standardisation means that contextual information can be exchanged between multiple vendors without worrying about compatibility.

Comprehensive Structured Context Profiles (CSCP) is a language based on the Resource Description Framework (RDF) [3] that attempts to provide a representation, based on these features, that is suitable for representing, storing and transferring contextual information. CCSP gains most of these features simply by basing itself on RDF (specifically, the XML implementation of RDF) since RDF is interchangeable, standardised, extensible and uniform. Decomposability in CCSP is achieved by allowing documents to refer to other documents to provide defaults. In this way, a document can concentrate on overriding only those pieces of information that differ. CCSP has several means of resolving conflicts that might arise in this scenario, such as merging, overriding and appending.

Beyond representation, some research advocates systematically generalising the integration of contextual information into applications. The Context Toolkit [14, 68], for example, is designed to sit between the raw sources of contextual information and applications. Analogous to GUI widgets, the toolkit proposes *context widgets* as a means of encapsulating contextual concepts (such as identity, presence and activity). Applications are thus able to deal with higher-level context such as the presence of a number of people in a room instead of having to derive this explicitly from low-level sensors. Widgets are composed of two types of further abstractions: *generators*, that produce raw sensor data, and interpreters, which can take this raw data and refine or reject it. Thus, a presence widget could be constructed by listening to a number of generators that detect motion, floor pressure, etc. and interpreters that examine this evidence to determine the presence of people. The widget would expose a simple interface that could be queried by applications and it could also be configured to notify applications when specific conditions are met (akin to a publish-subscribe system). Above widgets, it is possible to interpose *servers*. Servers allow the further aggregation and abstraction of contextual information provided by widgets. For example, a building server could track the presence of a number of people throughout a building, and perhaps link them to a particular identity as well. The modular design of the Context Toolkit would be well suited to a distributed system, and abstractions afforded by widgets could dramatically ease construction of a context-aware system over heterogeneous devices.

2.6 Other research areas

There are several other areas worth researching. In particular, it would be beneficial to examine distributed databases for insight into highly coherent caching and updating protocols and message-oriented middleware such as TIBCO [4] and Gryphon [5] and distributed object middleware including CORBA [6] to better understand the needs of some middleware systems.

3 Initial work and contributions

3.1 Model

To test the hypothesis that contextual information in ad hoc networks can improve data sharing between devices, we need to construct a model to allow experimentation. After consideration of alternatives such as complete distributed operating systems and distributed file systems, it seems a data sharing platform based on middleware is the most appropriate choice. Distributed operating systems, while extremely flexible in the breadth of applications they can support and in their ability to balance load and replicate data, are somewhat brittle in ad hoc networks (as alluded to in section 2.1). Distributed file systems, while offering many desirable properties such as persistence and clean semantics are not necessarily applicable in all situations (for example, where event notification is desired and persistence of data

[3] http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
[4] http://www.tibco.com
[5] http://www.research.ibm.com/gryphon/home.html
[6] http://www.omg.com

is not). Middleware systems can also offer clean semantics [27], support heterogeneous devices such as laptops and PDAs [42, 63], enable a wide variety of applications [13] and appear to be better suited to highly variable network topologies [12, 41].

Middleware takes many forms (as discussed in section 2.4). Each type is typically suited to a certain class of application. Some applications, such as instant messaging, are suited to an event-based publish-subscribe architecture. Others, such as large number factorisation, require persistence of shared objects for potentially long-lived computation and some applications, such as computer-supported cooperative work (CSCW) environments, need synchronisation mechanisms.

Our initial model is based on a synthesis of these many forms of middleware. Thus far we have devised a set of abstractions (termed *middies*) for generalised middleware that can be used as a basis for designing applications with properties and semantics desirable to their purpose. The model uses the notion of a distributed shared memory as a means of communication between devices. This memory corresponds to the PP, JT and CP spaces outlined in section 1.2.

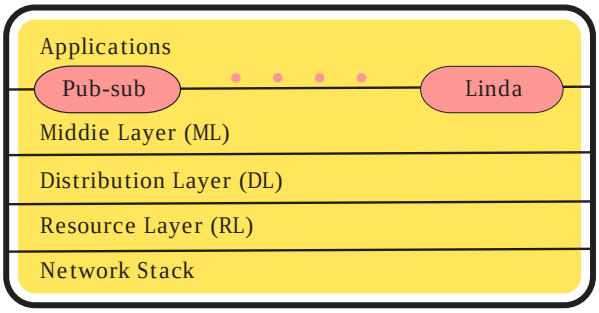


Figure 3: The Middie layered architecture.

The model is based on a layered architecture (as shown in Figure 3) which is intended to run on all devices that wish to use the middleware. At the bottom is the network layer which provides network connectivity between the devices. Above this is the Resource Layer (RL) which keeps track of which devices are part of the system and the services and resources available on these devices. It is the responsibility of the Distribution Layer (DL) to store data around the network and make sure they are recovered when needed for matching operations, etc. It should be possible to reimplement the DL as desired with new distribution policies in mind and not require reworking of the surrounding layers. The Middie Layer (ML) houses the generalised middleware abstractions. Middleware libraries use the ML to implement such middleware paradigms as publish-subscribe and tuple spaces. Finally, the applications sit on top, using the middleware libraries to perform their data sharing with similar applications on other devices.

Ideally, the layers should be as minimal as possible to allow efficient operation on resource-constrained devices such as PDAs and should be easily portable to a variety of heterogeneous architectures, which means no platform-specific elements (such as mobile code) are yet to be considered in our model.

By considering the various middleware systems described in section 2.4, it is possible to identify several general abstractions. Firstly, each of the systems has a concept of a logically centralised server to which the communicating parties connect. In the case of Linda, this is the tuple space. With publish-subscribe systems it is the subscription server, and with blackboards, it is the blackboard itself. We thus generalise this to the concept of shared *spaces*, which are logically centralised and can be referenced by any clients by simply providing a common name.

Similarly, each middleware has some unit of storage, be it a tuple, an event, object or an hypothesis. This can be generalised to a *block*, which we define simply as a flat, unstructured array of bytes.

Some middleware systems are reactive and need some mechanism for notifying clients of events. This is particularly evident in publish-subscribe systems, but similar mechanisms exist in others such as LIME and the Event Heap. It is conceivable that this could be emulated by constant polling from the communicating processes but this has multiple problems, of which the two primary ones are inefficient use of resources and the possibility of ‘missing’ events if the polls are not rapid enough [44, p56]. To address these issues, we introduce the concept of *reactors* which can be registered with spaces essentially as callbacks to be activated under certain conditions.

Each middleware system also has means of extracting data from the shared space. In tuple spaces for example, tuples that are matched to templates are returned to the caller. In publish-subscribe systems, events matching a subscription as defined by a subscription language are propagated to clients, and in MICA, objects can be extracted on a variety of

criteria, including their name and by using a query language. Clearly, any generalised middleware will need a *matcher* abstraction, which can be used to compare two blocks and return one if it is deemed to 'match' the other.

The concept of a client process, or *member*, is also needed and eventually it will also be necessary to provide some sort of security mechanisms, although this has not yet been fully considered for our model. Our current research indicates this will take the form of *capabilities*.

The Distribution Layer allows a space to be spread physically across multiple devices, while still remaining logically centralised. This is achieved by one or more members acting as *space providers*, meaning they are willing to devote memory to the system for the purpose of storing blocks. The DL on each device ensures that any block it is trying to store from the Middie Layer will be distributed to at least one space provider and that any member that is the target of a reactor triggered by the block will be made aware of its existence. Any DL on a device acting as a space provider may also receive commands to store or return blocks from other devices.

The DL collects service and resource information from the Resource Layer concerning all of the space providers that are available and decides how to distribute the blocks it has received according to its distribution policy. This policy can be altered by replacing the DL or perhaps via a plugin mechanism.

The interface presented to the Middie Layer is quite simple. It offers a mechanism to store or retrieve blocks and to join or leave a named space. When joining a space, the ML indicates if the device will be acting as a space provider and declares the device's unique member identifier. When leaving a space, any blocks stored on the device may need to be redistributed, depending upon the distribution policy.

Further details of the middies model can be found in [69], which has been submitted to ICPS'04.

In addition to designing a middleware model, we have briefly looked at the sorts of contextual information that are available in the scenarios in which we are interested. An initial categorisation has identified the following:

- **Environment**: location, nearby users, nearby services, nearby devices.
- **Device**: battery life, network connectivity, storage capacity, CPU speed, input/output modalities.
- **Application**: data access patterns, data types, security policies.
- **User**: high-level directives (e.g. "don't share this file", "don't store data on this device"), general preferences.

In future work. we intend to further organise this information and explore ways of producing and integrating it with the middies model discussed above.

3.2 Published and unpublished work

- "Middies: Passive Middleware Abstractions for Pervasive Computing Environments". Co-authored with Adam Hudson and Aaron Quigley. Submitted to 'International Conference on Pervasive Services 2004 (ICPS 2004)'.
- "BlueStar: Beacon + MPC based location detection". Co-authored with Belinda Ward, Aaron Quigley, Chris Ottrey, Bob Kummerfeld. To appear at 'Position, Location and Navigation Symposium (IEEE PLANS 2004)', April 26-29, 2004 - Monterey, California.
- "AR phone: Accessible Augmented Reality in the Intelligent Environment". Co-authored with Adam Hudson, Mark Assad and David J. Carmichael. Presented at OZCHI 2003.
- "A Demonstration of Mobile Augmented Reality". Co-authored with Adam Hudson, Mark Assad and David J. Carmichael. Demonstrated at OZCHI 2003.
- "Secure Identity Management and Nymity within the Intelligent Environment: A Survey and Discussion Paper". Part of the CRC Nymity project corpus.

- "Identity Management in Context-Aware Intelligent Environments". Presented at the Doctoral Colloquium at UbiComp 2003, published in the Adjunct Proceedings.
- "Adhocracy: Location-based, Ad Hoc Information Storage". Submitted to the UbiComp 2003 workshop on Location.
- "Nymity (and other made up words)". Seminar presented to School of I.T.
- "Motorola Introduction". Presented at Motorola Labs Australia.

3.3 Future plans

- April 2004: Deeper context study completed.
  It is clear that far more research into the areas of context and context-awareness in pervasive computing scenarios is required. This study will allow us to refine the various classes and types of context that may be useful to us.
- May 2004: Completed model design and confirmed hypothesis.
- June 2004: Begin construction of prototype model.
- July 2004: Present middies work at ICPS '04.
- September 2004: Present work at UbiComp 2004 workshop.
- October 2004: Completed construction of prototype model.
  The prototype will, for the most part, take the form of Java libraries and applications. If possible, we will use J2ME to enable deployment to severely constrained devices such as mobile phones.
- November 2004: Journal paper.
- February 2005: Begin experiments.
  At present we aim to test low-level aspects such as how various distribution policies (including random and single-node 'server' distribution, full replication and context-aware distribution algorithms) affect battery life and by building sample applications on top of these layers we aim to run high-level experiments that explore general data availability in the face of partial network and device failures. We expect our experiments to run mostly in simulation (possibly using simulation software such as UBIWISE [18]), but it is also hoped we will be able to build a physical prototype of the system that demonstrates the applicability of the model.
- July 2005: Journal paper.
- September 2005: Present work at UbiComp 2005.
- October 2005: Commence writing of thesis.
- March 2006: Submit thesis.

References

[1] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice-Hall Inc., 1995.

[2] A. Tanenbaum and M. Kaashoek, "The Amoeba microkernel," 1994. [Online]. Available: citeseer.nj.nec.com/article/tanenbaum94amoeba.html

[3] F. Douglis, J. K. Ousterhout, M. F. Kaashoek, and A. S. Tanenbaum, "A comparison of two distributed systems: Amoeba and Sprite," *Computing Systems*, vol. 4, no. 4, pp. 353–384, 1991. [Online]. Available: citeseer.nj.nec.com/douglis91comparison.html

[4] M. Satyanarayanan, “Scalable, secure, and highly available distributed file access,” *Computer*, vol. 23, no. 5, pp. 9–18, 20–21, 1990.

[5] P. J. Braam, “The Coda distributed file system,” *Linux J.*, vol. 1998, no. 50es, p. 6, 1998.

[6] B. D. Noble and M. Satyanarayanan, “Experience with adaptive mobile applications in Odyssey,” *Mobile Networks and Applications*, vol. 4, no. 4, pp. 245–254, 1999. [Online]. Available: citeseer.nj.nec.com/noble99experience.html

[7] P. Johansson, E. YZ, l Mario, G. Manthos, and K. UCLA, “Bluetooth an enabler of personal area networking,” 2001. [Online]. Available: citeseer.nj.nec.com/johansson01bluetooth.html

[8] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. D. Kim, B. Zhou, and E. G. Sirer, “On the need for system-level support for ad hoc and sensor networks.” [Online]. Available: citeseer.nj.nec.com/barr02need.html

[9] P. Keleher, “Decentralized replicated-object protocols,” in *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC’99)*, 1999. [Online]. Available: citeseer.nj.nec.com/keleher99decentralized.html

[10] R. Baldoni, A. Virgillito, and R. Petrassi, “A distributed mutual exclusion algorithm for mobile ad-hoc networks,” in *Proceedings of the 7th IEEE Symposium on Computers and Communications (ISCC 2002), Taormina, Italy, 1-4 July 2002*, Jul 2002, pp. 539–545.

[11] E. Royer and C. Toh, “A review of current routing protocols for ad-hoc mobile wireless networks,” 1999. [Online]. Available: citeseer.nj.nec.com/royer99review.html

[12] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek, “Mobile computing with the Rover Toolkit,” *IEEE Transactions on Computers*, vol. 46, no. 3, pp. 337–352, 1997. [Online]. Available: citeseer.nj.nec.com/joseph97mobile.html

[13] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall, “Programming for pervasive computing environments,” University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-01-06-01, June 2001.

[14] D. Salber, A. K. Dey, and G. D. Abowd, “The context toolkit: Aiding the development of context-enabled applications,” in *CHI*, 1999, pp. 434–441. [Online]. Available: citeseer.nj.nec.com/salber99context.html

[15] A. Held, S. Buchholz, and A. Schill, “Modeling of context information for pervasive computing applications.” [Online]. Available: citeseer.nj.nec.com/547105.html

[16] J. Pascoe, “The Stick-e Note architecture: Extending the interface beyond the user,” in *Intelligent User Interfaces*, 1997, pp. 261–264. [Online]. Available: citeseer.nj.nec.com/pascoe97sticke.html

[17] M. Satyanarayanan, “Fundamental challenges in mobile computing,” in *Symposium on Principles of Distributed Computing*, 1996, pp. 1–7. [Online]. Available: citeseer.nj.nec.com/satyanarayanan96fundamental.html

[18] J. Barton and V. Vijayaraghavan, “UBIWISE, a ubiquitous wireless infrastructure simulation environment,” HP Labs, Tech. Rep. HPL-2002-302, 2002.

[19] Y. Aridor, M. Factor, and A. Teperman, “cJVM: A single system image of a JVM on a cluster,” in *International Conference on Parallel Processing*, 1999, pp. 4–11. [Online]. Available: citeseer.nj.nec.com/aridor99cjvm.html

[20] R. Veldema, R. Bhoedjang, and H. Bal, “Jackal, a compiler based implementation of Java for clusters of workstations.” [Online]. Available: citeseer.nj.nec.com/308851.html

[21] P. Doyle and T. Abdelrahman, “Jupiter: A modular and extensible JVM,” in *Proceedings of the Third Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*. ACM, June 2001, pp. 37–48. [Online]. Available: citeseer.nj.nec.com/doyle01jupiter.html

[22] M. Ma, C. Wang, F. Lau, and Z. Xu, “Jessica: Java-enabled single system image computing architecture,” 1999. [Online]. Available: citeseer.nj.nec.com/article/ming99jessica.html

[23] E. G. Sirer, R. Grimm, A. J. Gregory, N. Anderson, and B. Bershad, "Distributed virtual machines: A system architecture for network computing," Dept. of Computer Science & Engineering, University of Washington, Tech. Rep. TR-98-09-01, 1998. [Online]. Available: citeseer.nj.nec.com/sirer98distributed.html

[24] D. D. Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt, "The evolution of the grid." [Online]. Available: citeseer.nj.nec.com/535794.html

[25] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa, "WebOS: Operating system services for wide area applications," in *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, 1998. [Online]. Available: citeseer.nj.nec.com/vahdat97webos.html

[26] L. Chen, "AgentOS: the agent-based distributed operating system for mobile networks," *Crossroads*, vol. 5, no. 2, pp. 12–14, 1998.

[27] A. L. Murphy, G. P. Picco, and G.-C. Roman, "LIME: A coordination middleware supporting mobility of hosts and agents," Washington University, Tech. Rep. WUCSE-03-21, Apr. 2003.

[28] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*. New York, NY: ACM Press, 1986, pp. 229–239. [Online]. Available: citeseer.nj.nec.com/li89memory.html

[29] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: distributed shared memory based on type-specific memory coherence," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*. ACM Press, 1990, pp. 168–176.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun Network Filesystem," in *Proc. Summer 1985 USENIX Conf.*, Portland OR (USA), 1985, pp. 119–130. [Online]. Available: citeseer.nj.nec.com/sandberg85design.html

[31] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp file system," in *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, 1991, pp. 226–38. [Online]. Available: citeseer.nj.nec.com/liskov91replication.html

[32] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A scalable distributed file system," in *Symposium on Operating Systems Principles*, 1997, pp. 224–237. [Online]. Available: citeseer.nj.nec.com/thekkath97frangipani.html

[33] B. Grönvall, I. Marsh, and S. Pink, "A multicast-based distributed file system for the internet," in *Proceedings of the 7th workshop on ACM SIGOPS European workshop*. ACM Press, 1996, pp. 95–102.

[34] B. Atkin and K. P. Birman, "MFS: An adaptive distributed file system for mobile hosts," Department of Computer Science, Cornell University, Tech. Rep., 2003.

[35] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," 2002. [Online]. Available: citeseer.nj.nec.com/adya02farsite.html

[36] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*. ACM Press, 2001, pp. 149–160. [Online]. Available: citeseer.nj.nec.com/stoica01chord.html

[38] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "BASE - a micro-broker-based middleware for pervasive computing," in *PerCom*. IEEE Computer Society, Mar 2003, proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom'03), March 23-26, 2003, Fort Worth, Texas, USA.

[39] E. Guttman, C. Perkins, J. Veizades, and M. Day, “Service Location Protocol, Version 2,” IETF RFC 2608, June 1999, http://www.ietf.org/rfc/rfc2608.txt.

[40] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001, pp. 329–350. [Online]. Available: citeseer.nj.nec.com/article/rowstron01pastry.html

[41] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch, “The Bayou architecture: Support for data sharing among mobile users,” in *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, Santa Cruz, California, 8-9 1994, pp. 2–7. [Online]. Available: citeseer.nj.nec.com/demers94bayou.html

[42] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich, “XMIDDLE: A data-sharing middleware for mobile computing,” *Wirel. Pers. Commun.*, vol. 21, no. 1, pp. 77–103, 2002.

[43] P. J. McCann and G.-C. Roman, “Compositional programming abstractions for mobile computing,” *Software Engineering*, vol. 20, no. 10, 1998.

[44] B. E. Johanson, “Application coordination infrastructure for ubiquitous computing rooms,” Ph.D. dissertation, Stanford University, 2002.

[45] D. Gelernter, “Generative communication in Linda,” *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[46] ——, “Multiple tuple spaces in Linda,” in *Parallel Architectures and Languages Europe (PARLE ’89)*, ser. LNCS, E. Odijk, M. Rem, and J.-C. Syre, Eds., vol. 366. Eindhoven, The Netherlands: Springer-Verlag, 1989, pp. 20–27.

[47] S. C. Hupfer, “Melinda: Linda with multiple tuple spaces,” Yale University, Tech. Rep. YALE/DCS/RR-766, Feb. 1990.

[48] R. A. Whiteside and J. S. Leichter, “Using Linda for supercomputing on a local area network,” in *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1988, pp. 192–199.

[49] C. J. Fleckenstein and D. Hemmendinger, “A parallel ‘make’ utility based on Linda’s tuple-space,” in *Proceedings of the seventeenth annual ACM conference on Computer science: Computing trends in the 1990’s*. ACM Press, 1989, pp. 216–220.

[50] S. P. Wade, “An investigation into the use of the tuple space paradigm in mobile computing environments,” Ph.D. dissertation, Lancaster University, 1999.

[51] N. Davies, S. Wade, A. Friday, and G. Blair, “Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications,” in *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP ’97)*, Toronto, Canada, May 1997, pp. 291–302. [Online]. Available: citeseer.nj.nec.com/davies97limbo.html

[52] G. P. Picco and M. L. Buschini, “Exploiting transiently shared tuple spaces for location transparent code mobility,” in *Coordination Models and Languages*, 2002, pp. 258–273. [Online]. Available: citeseer.nj.nec.com/picco02exploiting.html

[53] S. R. Handorean, “Coordination middleware supporting rapid deployment of ad hoc mobile.” [Online]. Available: citeseer.nj.nec.com/592848.html

[54] P. Wyckoff, “T spaces,” Online, 1998, available: http://www.research.ibm.com/journal/sj/373/ wyckoff.html.

[55] B. Johanson, S. Ponnekanti, E. Kiciman, C. Sengupta, and A. Fox, “System support for interactive workspaces,” Mar. 2001, available: http://graphics.stanford.edu/papers/iwork-sosp18/.

[56] B. Segall and D. Arnold, “the building: A publish/subscribe notification service with quenching,” in *Proceedings AUUG97, Brisbane, Australia, September 1997*. Distributed Systems Technology Centre, University of Queensland, Australia, 1997.

[57] P. Kostkova, S. Crane, J. McCann, and T. Wilkinson, “MAGNET: QoS-based dynamic adaptation in a changing environment,” HiPeX, Department of Computer Science, City University, London, UK, Tech. Rep., 1998.

[58] “JavaSpaces(TM) service specification,” Online, http://java.sun.com/products/jini/2.0/doc/specs/html/ js-TOC.html.

[59] M. A. Leal, N. Rodriguez, and R. Ierusalimschy, “LuaTS - a reactive event-driven tuple space,” *Journal of Universal Computer Science*, vol. 9, no. 8, pp. 730–744, 2003.

[60] D. D. Corkill, “Blackboard systems,” *AI Expert*, vol. 6, no. 9, pp. 40–47, Sept. 1991.

[61] ——, “Collaborating software: Blackboard and multi-agent systems & the future,” in *Proceedings of the International Lisp Conference*, Oct. 2003.

[62] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, “The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty,” *Computing Surveys*, vol. 12, no. 2, pp. 213–253, June 1980.

[63] M. W. Kadous and C. Sammut, “MICA: Pervasive middleware for learning, sharing and talking,” 2003, submitted to PerWare 2004.

[64] T. G. Kanter, “Attaching conext-aware services to moving locations,” *IEEE Internet Computing*, pp. 43–51, 2003.

[65] A. K. Dey, D. Salber, M. Futakawa, and G. D. Abowd, “An architecture to support context-aware applications,” GVU Center, College of Computing, Georgia Institute of Technology, Tech. Rep. GIT-GVU-99-23, 1999.

[66] A. Dey, J. Mankoff, G. Abowd, and S. Carter, “Distributed mediation of ambiguous context in aware environments,” in *Proceedings of the 15th annual ACM symposium on User interface software and technology*. ACM Press, 2002, pp. 121–130.

[67] “The casco project: Investigating context aware support for cooperative applications in ubiquitous computing environments,” Online, http://www.lancs.ac.uk/ug/fittond/casco/.

[68] G. D. A. Anind K. Dey and D. Salber, “A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications,” *HCI*, vol. 16, pp. 97–166, 2001.

[69] D. Cutting, A. Hudson, and A. Quigley, “Middies: Passive middleware abstractions for pervasive computing environments,” 2004, submitted to ICPS’04.