Wireless Networks 9, 311–320, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Scalable, Distributed Middleware Service Architecture to Support Mobile Internet Applications

RAJIVE BAGRODIA *, THOMAS PHAN and RICHARD GUY
3809 Boelter Hall, The University of California at Los Angeles, Los Angeles, CA 90095-1596, USA

Abstract. Middleware layers placed between user clients and application servers have been used to perform a variety of functions to support the vision of nomadic computing across varying platforms. In previous work we have used middleware to perform a new capability, application session handoff, using a single Middleware Server to provide all functionality. However, to improve the scalability of our architecture, we have designed an efficient distributed Middleware Service layer that properly maintains application session handoff semantics while being able to service a large number of clients. We show that this service layer improves the scalability of general client-to-application server interaction as well as the specific case of application session handoff. We detail protocols involved in performing handoff and analyse an implementation of the architecture that supports the use of a real medical teaching tool. From experimental results it can be seen that our Middleware Service effectively provides scalability as a response to increased workload.

Keywords: distributed middleware, nomadic computing, ubiquitous computing, load balancing, application session handoff

1. Introduction

Middleware has previously been used to perform a variety of functions on behalf of applications running on mobile and fixed Internet-connected clients, such as performing content transcoding and providing quality-of-service. Such functionality will become increasingly more crucial due to the proliferation of wirelessly connected mobile computing devices. We are investigating the importance of middleware in the Interactive Mobile Application Support for Heterogeneous Clients (iMASH) project [3] at UCLA. In our research we are designing a hardware/software architecture utilising a key middleware tier to support heterogeneous mobile and fixed clients interacting with an application server that provides a repository for data. The goal of iMASH is to research and develop the architecture and protocols to support a wireless computing infrastructure for the new state-of-the-art hospital being built on this campus.

In iMASH we are supporting a particular vision of nomadic computing [26] that allows users to have access to data stored at an application server from mobile and fixed clients regardless of time, location, or platform. In this model, the network acts as a conduit through which application session state migrates from device to device to provide continuity to the user experience. In our work a Middleware Server (MWS) will enable this vision and provide such capabilities as presentation transcoding, enforcement of quality-of-service, and security on behalf of client machines ranging from a wirelessly connected handheld device to an ethernet-connected workstation. Within the past year we have focused on a particular functionality hitherto unresearched: the handoff of an application's session state from one client device to another with the aid of a MWS. This Application Session Handoff, or ASH, facility allows the user to have a seamless, uninterrupted computing experience across client machines. We envision this capability will have increased importance as wirelessly-connected mobile computing behaviour becomes more prevalent among users. In previous work we have shown that this functionality is viable by implementing it within a real-world pedagogical tool called the Teaching File used in the medical school.

However, from our research we have seen that the iMASH-enabled Teaching File application and our initial Middleware design are subject to a number of bottlenecks that limit the scalability of the iMASH architecture. In particular, we have isolated regions of operation in the MWS designed to support the Teaching File that are computationally-bound and hence perform poorly with an increasing number of service requests. Additionally, we have seen that the ASH facility also performs poorly when the workload at the MWS increases.

To counter these effects and to provide more scalable service, we have redesigned our single Middleware Server layer into a multi-server distributed Middleware Service that is more capable of handling increased traffic and workload. Our Middleware Service is designed such that all individual servers within it make policy decisions autonomously, thereby avoiding the need for Service-wide global state being maintained and lessening the danger of having any single server cause a bottleneck. We have designed an accompanying protocol that we have streamlined to provide an efficient means to transfer data only when needed to support both normal client-to-Application Server interactions as well as Application Session Handoff.

This paper is organised as follows. Initially we will define our research space and give examples of the utility of nomadic applications in section 2. We then describe the areas of scalability concern in the current iMASH architecture and the Teaching File application in section 3 followed by our

* Corresponding author.
E-mail: rajive@cs.ucla.edu

proposed resolution to these matters in section 4. Our discussion of an implementation with related experiments and results proceeds in section 5. In section 6 we discuss work related to our own. We conclude the paper with section 7.

2. Nomadic computing

A nomadic computing application is a program that allows the user to leverage network connectivity, provided by a wired or wireless infrastructure, to access and utilise his data productively from any location, on any platform, and at any time. Research in this area is driven by a number of compelling scenarios that enable users to utilise computing services across a number of boundaries. In this section we describe several contexts in which nomadicity is appealing and explain how this style of computing can be realised with the support of a middleware layer.

Consider a typical graphical image application that allows the user to interact with a set of pictures stored at a server on one of several client platforms. The application allows the user to manipulate these images by painting on it and applying image processing filters. When this application is enabled to provide nomadic interaction, the user will be able to run the program across a variety of platforms, including a desktop running on an Ethernet LAN, a laptop connected via an 802.11 wireless card, and a PDA using a cellular data modem. In our research we augment the notion of nomadicity by considering applications that can encapsulate a session, represented by the principal data structures and run-time values associated with the application's current state. This session state can follow the user from device to device, allowing the user to access his data regardless of geographic location, device, or time. Furthermore, for this graphics program, changes made to an image would be available when the user moves to another device due to the mobility of the state.

We can generalise the preceding graphical application to a number of different applications. More importantly, it is easy to see why this type of application behaviour is compelling: it allows users to concentrate on the content of their work rather than making them aware of limitations of their computing device. With mobile session states, the user is presented with a familiar set of data that can be adapted, or transcoded, to fit the device limitations of his current platform while still allowing him interactivity with the data. In order to enable this vision, the applications are supported by a middleware software and hardware layer placed between the clients and the application servers. Such middlware provides the functionality to enable location-awareness, transcoding, security, and other characteristics. We now look at two programs that are reprentative of the types of emerging applications that can leverage this capability. Additionally, we will suggest how these applications can be supported by a middleware tier.

2.1. A nomadic medical teaching tool

A medical teaching program, enabled to provide nomadic interactive access to stored data, provides a demonstration of the promise shown by this research. Consider such a medical application that displays multimedia data downloaded from an application server to a fixed or mobile client device. This application would present medical data, such as images and annotations, with a well-formed interface. We have been working with such a teaching tool called the Teaching File, which has been developed in-house for radiological professors at UCLA. The Teaching File is a Java applet that retrieves medical data from an application server via HTTP, displays images, and allows users to add textual annotations to form instructional files. This program is intended to be used on any fixed or mobile client capable of executing Java applets over the Internet.

Enabling this application to support nomadic interactivity brings to the table two exciting benefits for the user. First, the user will be able to access his data (stored at an appliation) regardless of his current platform or physical location when he operates this application on varying platforms. A runtime system supports this nomadic computing model by transparently tracking his location and transferring the application's session state across devices. In order to efficiently and seamlessly provide this service, a tier of middleware is introduced between the application server and clients. Second, the middleware provides the means for content transcoding and intelligent data updates so that the user's view of data is consistent across all devices.

2.2. A shared collaborative multimedia space

We now turn out attention to see how users of collaborative interactive software can gain the advantages of nomadic applications with middleware assistance. Multi-user interactive software has become increasingly more common. Groupware applications provide a means for geographically dispersed users to show, manipulate, and emphasise ideas using a software architecture that disseminates multimedia information in real-time to other participating users. A wide variety of products have emerged, including research and open source projects, such as Network Text Editor [11], Multicast Multimedia Conference Recorder [14], and phpGroupware [21], as well as commercial software, such as Novell Groupwise [18], IBM/Lotus Notes [12], and Microsoft Exchange [16].

The integration of nomadic computing capabilities into such applications enables a number of enhancements. As before, the principal benefit is that heterogeneity is hidden from the users: the multimedia space is shared even if users are utilising platforms as diverse as desktops and PDAs. By abstracting the shared space as a session state, we can envision that this state will be distributed across the collaboration participants. Users can contribute to the shared space by adding multimedia or manipulating items, and the resultant changes will be made available to other users after the content is appropriately transcoded for each device. Additionally, if a user moves from one platform to another, he can continue the collaborative session automatically regardless of the device's location or display characteristics. Again, the middle-

ware would provide the means to support location migration, user tracking, and data transcoding.

3. Scalability issues for iMASH Middleware Servers

The envisioned freedom granted to nomadic applications in the previous section is provided by middleware support. In our research, a Middleware Server serves a prominent role in the deployment and execution of nomadic programs by supporting a number of different functionalities needed by the clients. The *distributed* iMASH Middleware tier has evolved out of necessity from performance and scalability concerns with our original single Middleware Server design. In this section we describe the initial architecture using a single Middleware Server and its natural progression toward a distributed Middleware Service layer. The arguments leading into the next section are specific to our own Middleware design but can be applied to middleware servers in general.

3.1. The iMASH Middleware architecture

The primary goal of the iMASH research project is to facilitate the network interconnectivity between an *application server* (AS) and users working on heterogeneous mobile and fixed *clients*. In the traditional context, the AS acts a data repository for the clients and must shoulder the load of providing service for a possibly large number of simultaneous transactions. Clearly, such an approach is not scalable under a variety of conditions: increased number of simultaneous clients needing service, increased frequency of transactions, increased size of transaction payload, or increased service at the AS needed by clients.

In the iMASH architecture, shown in figure 1, we have interpositioned a Middleware Server (MWS) layer between the clients and the AS to provide a number of benefits. The eventual design goal called for this layer to contain a number of strategically placed servers that would best service the clients.

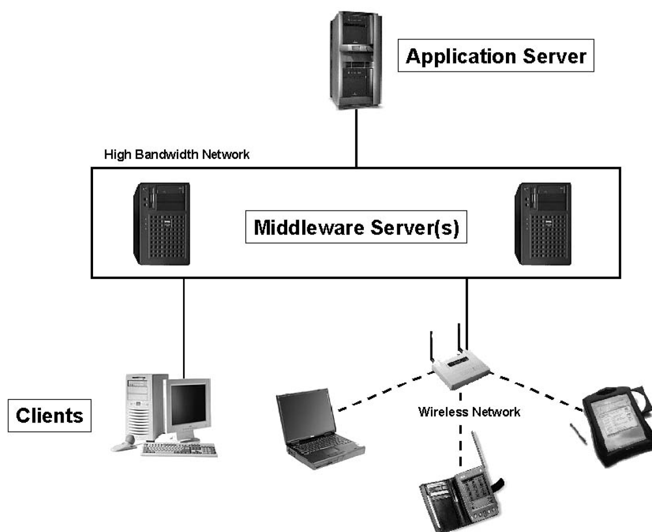


Figure 1. A high-granularity view the iMASH hardware architecture.

We posited that such a design would increase scalability of the original client-server interactions. From this architecture it is clear that a given MWS acts as the sole client for the AS, thereby allowing the AS to better serve its role as a data provider rather than as a manager for heterogeneous clients. In a related vein, the MWS also acts as the service manager for all the clients.

In addition to easing the burden on the AS, the iMASH Middleware is designed to perform (1) user authentication and profiling, (2) presentation transcoding, (3) location tracking and network quality-of-service adaptation, (4) computational off-loading from client to middleware, and (5) application session handoff. In the first year of iMASH research, we have concentrated our efforts in providing the session handoff capability and to a lesser extent computational off-loading, both of which we will discuss in this paper. Even with this subset of capabilities, we have already identified scalability problems with a single MWS configuration. By identifying such areas now and adjusting our hardware/software architecture to meet this challenge, we will lay a foundation for future developmental needs.

3.2. Application Session Handoff

In previous work, we showed a novel use for Middleware to support clients: the Application Session Handoff. We briefly describe this capability and a specific implementation before identifying areas of performance degradation in the next subsection.

Application Session Handoff (ASH) [19,20] allows for users to move the current session state of an application running on one client to another (likely heterogeneous) client with little to no interruption in usage continuity. For example, a user may be working with an iMASH-enabled application on a desktop workstation, and upon deciding that he needs to continue his work on his mobile personal digital assistant (PDA), he activates the ASH capability to seamlessly transfer all existing and relevant session state, such as rendered images, text, or user preferences, from the desktop to the PDA. As part of this transfer, the data's content may undergo filtration to fit the bandwidth, display, and computation limitations of the PDA (for example, a colour image may be reduced to black and white to fit monochrome PDAs). The iMASH runtime system performs all actions required to perform this handoff, thus alleviating the user of any need to explicitly save data to an intermediary medium (such as a disk) and then to reinstantiate the data on the target machine. In the iMASH system the network serves as a conduit between the Middleware Servers providing this capability to iMASH-ready applications on mobile and fixed hosts.

Among the programs we have enabled with this handoff capability is the Teaching File application previously mentioned. We have augmented this multimedia teaching tool with a specific form of session handoff which we have identified as a Two-Way Interactive Session Transfer (TWIST) [19]. Users are allowed to make modifications to the instructional files at the client after retrieving them from the AS; upon an

Application Session Handoff to another client machine, these same modifications are visible to the user, thus making the transfer "interactive". Furthermore, the user retains any capabilities to interact with the AS, including the ability to save the data back to the AS, making the experience "two-way", even in the face of content-adaptation changes made for the benefit of limited clients. In order to implement this capability, some source code modifications were needed. Although our design does not require any modification to the AS or its software, we do assume that the client applications that participate in the iMASH environment can be modified via source code changes. This requirement allows us to add a portion of code which we call the Middleware-Aware Remote Code, or MARC, into the application to provide a client-side proxy that acts on behalf of the user and the application. In our work we have been able to move sessions between wired desktops running Solaris 7 or Windows NT 4.0 SP5 as well as wirelessly connected (via Lucent WaveLAN) devices, such as mobile laptops running Windows 2000, an Aqcess Technologies QBE graphical tablet running Windows 98, and a Compaq IPAQ PDA running Windows CE.

In figure 2 we diagram a typical flow of events to illustrate an application session handoff for the Teaching File. We denote $C_1, C_2, \ldots, C_n$ to be the set of client machines available to a user. Initially a user on client C_1 requests a data object o from the AS.

(1) The AS returns the object o to the MWS, which is immediately cached.

(2) The MWS filters o to suit the limitations of C_1 by applying a filter function f_1 to o; client 1 thus receives $f_1(o)$ from the MWS.

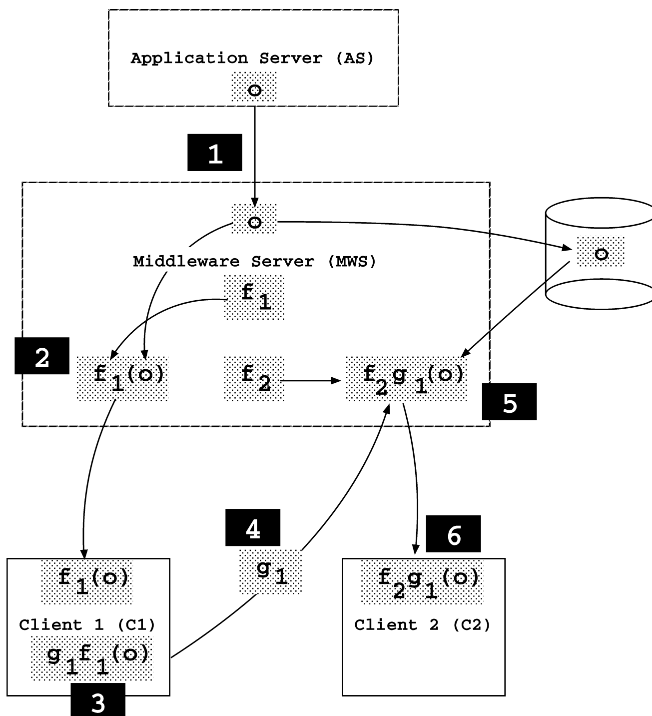


Figure 2. Handoff of an application session.

(3) The user at C_1 proceeds to modify the data by applying an operation g_1 to the object, resulting in $g_1 \circ f_1(o)$.

(4) At this point the user wishes to perform application session handoff to another machine. Because this is a *two-way interactive* transfer, the result of the operation g_1 at C_1 must be made visible to the user at C_2 when the session is reinstantiated. The modified form of the data must thus made available at the MWS so that it can be sent to C_2. Note though that upon handoff, only the operation g_1 is sent back to the MWS. This is an optimisation we have devised that allows us to send back the much smaller g_1 instead of the entire data object $g_1 \circ f_1(o)$. We assume that the operation g_1 can be sent in isolation from C_1 back to the MWS. In our research we have seen that many types of operations can be represented in XML [9] and can be thus orders of magnitude smaller than a binary object [19].

(5) The MWS needs to assemble a user-modified data object to C_2, but as another optimisation, we have stated that the first content filter f_1 may be contextually irrelevant to C_2, so we may simply need the o that was cached and the g_1 that was delivered from C_1.

(6) The result is passed through f_2, the content filter for C_2.

(7) Finally, $f_2 \circ g_1$ is delivered to C_2.

3.3. Areas of scalability concern

Our initial implementation of session handoff with one MWS successfully allowed us to move Teaching File sessions from one machine to another. We have been working with the initial assumption that the MWS machine would have enough bandwidth and computational power to support a reasonable number of clients during our initial phase of iMASH research. However, we have come to a stage of development where we have seen that this architecture will not suffice. We now enumerate and discuss issues that pose scalability problems.

The middleware layer's purpose within the iMASH architecture is to reduce the load on the AS and to provide a variety of services for the clients. However, in moving the burden of workload from the AS to the MWS, we have also moved the potential for a computational and bandwidth bottleneck to the MWS as well. Just as we discussed in section 3.1 regarding how a sole AS lacks scalability, a sole MWS is not scalable under several conditions: (1) increased number of simultaneous clients needing service, (2) increased frequency of transactions, (3) increased size of transaction payload, and (4) increased service needed by clients at the MWS. We must also address the problems specific to the Application Session Handoff capability itself. We do so by evaluating the potential bottlenecks incurred with the Teaching File application.

In figure 2 we showed the sequence of actions needed for the Teaching File application to retrieve data from the AS and then handoff the session to another client. Bottlenecks can occur at two primary locations. First, when the user requests data from the AS, the MWS retrieves the data on his behalf.

Graphical images are stored at the AS in a special proprietary medical format called PACS [24] and must be processed to form a Java Image object. The original Teaching File application performed this processing at the client, but we have moved this workload to the MWS. An increase in the number of simultaneous clients, frequency of requests, or size of request data would cause a bottleneck at the MWS because this image processing is a CPU-intensive task that cannot take advantage of multi-threading on a uniprocessor MWS. In the future, we may need to process unforeseen proprietarily formatted data, which would increase the service time being performed at the MWS. Additionally, although we do not perform it for the case of the Teaching File, content adaptation to fit the characteristics of the client machines will place further load on the MWS.

Second, a bottleneck may occur during the course of an Application Session Handoff. We noted in the previous section that when an ASH occurs, we minimise the amount of data sent from a client C_1 to the MWS because we send only the user operation g_1 instead of the data object $g_1 \circ f_1(o)$. However, when the MWS receives g_1, it must still create the object $f_2 \circ g_1(o)$ that must be shipped to client C_2. The formation of $f_2 \circ g_1(o)$ involves the application of g_1 to object o as well as the application of the content filter f_2 to the resultant object $g_1(o)$. These steps may be non-trivial in nature, and under the conditions expressed in the previous case, may produce a bottleneck in system.

We attempt to solve these problems by implementing a Middleware *Service* (MWService) layer that contains a distributed set of individual servers. We discuss this architecture next.

4. A Middleware Service

In this section we develop an architectural solution that addresses the scalability issues in the last section. Our design involves the use of several servers acting as an aggregate Middleware Service (MWService) to provide the scalable functionality intended by the iMASH environment. In addition, our MWService is governed by a functionally-driven but efficient protocol that aids in Application Session Handoff.

A broad view of this architecture is shown in figure 3. Such an architecture involving the aggregation of separate computing stations is not novel and is well-known as a Network of Workstations. Our MWService is formed from such a network of individual Middleware Servers that can be distributed over either a local or a wide area. This service is guided by the following tenets: (1) no one MWS causes a bottleneck across the entire system, (2) no one MWS maintains any global state of system, (3) each MWS autonomously decides if it is capable of providing service for the clients and if so, autonomously registers itself as an available MWS, and (4) participation between two Middleware Servers to exchange session state does not add significant computational complexity to the Application Session Handoff. If properly designed, our distributed MWService will show scalability under increased load conditions (as mentioned in section 3.3) by limiting the maximum service time experienced by clients to not grow without bound. We next provide a scalable MWService design followed by a discussion of its use to facilitate specific actions between the client and the MWService.

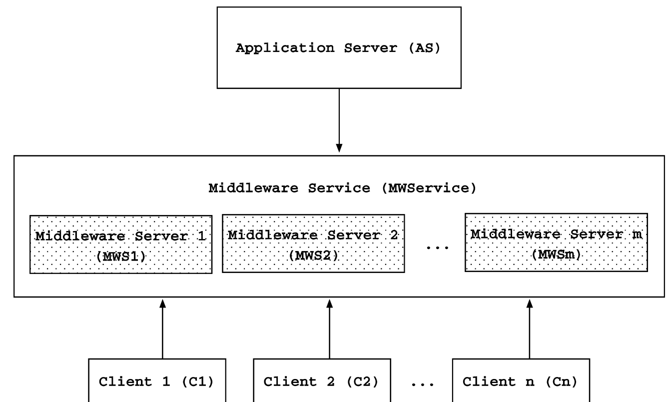


Figure 3. A broad view of the Middleware Service.

4.1. A scalable Middleware Service protocol

The driving motivation for our MWService design is to make available to clients an entity that is capable of providing service upon request. We now explain the details of the protocol that drives this facility. Initially each Middleware Server in the distributed MWService determines autonomously if it is capable of providing service. Such a decision can be based on comparing a representative value, such as the well-known metrics of proximity to mobile clients, CPU load, service queue length, and available bandwidth, against a threshold quantity. The selection of an appropriate MWS availability metric is beyond the scope of this paper; our architecture assumes that the MWS has access to a policy and means to make such a decision. In our experiments we have implemented a simple policy based on CPU load, as will be shown in section 5. If the MWS determines that it is available, it registers itself with a lookup service that will list all capable MWSes for clients to utilise. This lookup service, or registry, allows clients to discover a MWS within the MWService that can perform work on the client's behalf. Such a discovery is possible because the iMASH-aware MARC code within the application, acting on behalf of the client, interacts with the registry. If, over its course of operation, the MWS determines that it can no longer adequately provide service, it deregisters itself from the lookup registry. When the MWS is again capable of operating sufficiently, it registers itself again. If more than one MWS is capable of fulfilling the client's request, a MWS is chosen randomly. If no MWSes are registered, several courses of actions are possible, such as having the lookup registry attempt to poll the MWSes for availability or having the client's service request be dropped with an appropriate error returned.

Note that this registration immediately offers a scalable and distributed means of load distribution and admission control for the MWService. Each MWS determines its availabil-

ity autonomously by evaluating its own metric. By deregistering itself from the MWService lookup registry, the MWS prevents itself from entering a state with an unstably increasing workload. Additionally, the registry itself can be a scalable service that is not subject to a bottleneck; approaches such as the Service Location Protocol [10] and Jini [25] are available.

The MWService continues to maintain scalability even after a client has gained admittance to a particular MWS. Suppose a client C_i has been given service by a MWS, resulting in a portion of the client's session state being cached at the MWS. At some point the MWS may become unavailable and thus deregister itself to avoid any new service requests. However, the MWS should still provide service for C_i and any other clients that were already admitted. For these clients the MWS may need to perform an action such as an Application Session Handoff or to enforce a load-balancing policy. In response to such events, the MWS will perform a *middleware-to-middleware handoff* of the client's session state to another available MWS. This target MWS can be found by the first MWS through the registry. At this point the client C_i should be informed that it must now interact with the target MWS. The client can attain this information through two means. The registry can give the address of the target MWS, but this would require that the client be able to asynchronously receive such updates from the registry as well as maintain its normal communication channel with the MWS. There is a better alternative: at the next instance when the client requests service from the initial MWS, this MWS will give the address of the target MWS. This on-demand updating can be done through the regular client–MWS API.

4.2. Scalability for client-to-AS interaction in the Teaching File

We now discuss how our scalable protocol affects our Teaching File application. Although our goal for the distributed MWService architecture is to provide scalability in the specific case of Application Session Handoff, we must first discuss its scalability properties in the general case of normal client-to-AS interaction. When the Teaching File initially requests the retrieval of data from the AS, the request travels through the MWService layer as before. If the individual servers in the layer are geographically distributed across a wide area, it naturally follows that the clients should be serviced by the nearest possible MWS; the lookup registry will return a result that reflects this approach. Immediately we posit that our distributed MWService architecture provides a natural load distribution if we assume that the clients are themselves geographically clustered across a wide area and can be logically associated with a MWS.

We now consider the case where the servers in the MWService layer provide functionality in a local area environment. We note that the requests issued by the Teaching File have the property of being discrete and connectionless, thereby making each request independent of all previous requests. Also, the MWService actions are completely transparent to the user, who is only aware of his requests to the AS. The intermediate transactions to the MWService layer is handled by the Teaching File's MARC code.

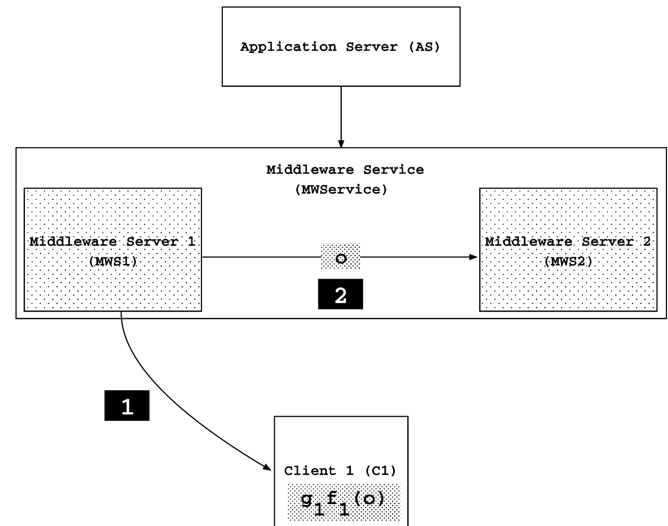


Figure 4. A middleware-to-middleware handoff.

Suppose that after a client C_1 has been associated with (and received initial service from) a particular MWS_1, that MWS may become unable to provide further service. In figure 4 we depict the situation at large. In step 1 MWS_1 has provided service C_1 in the form of delivering a piece of data to the client. At some point the MWS decides it can no longer provide service and redirects future requests from this client to MWS_2. In this situation the application's session state has no reason to evacuate C_1; the data object $g_1 \circ f_1(o)$ can stay on the client. However, because C_1 is now serviced by MWS_2, MWS_1 must send its cached object o to MWS_2 in step 2 (in anticipation of an Application Session Handoff). Note that we must prohibit MWS_2 from attaining object o from the AS instead of MWS_1 for two reasons: (1) we do not wish to unnecessarily burden the AS with another data fetch because the ASH is a concern to the iMASH MWService, not to the AS, and (2) the data fetches may have nonidempotent effects at the AS.

4.3. Scalability for Application Session Handoff in the Teaching File

We now discuss the use of the distributed MWService layer to provide scalability for Application Session Handoff. Recall that ASH is a functionality that allows the user to transfer an application's session state from one client machine to another device with little to no interruption in service with the aid of a Middleware Server. The protocol needed to perform these actions was shown in section 3.2. In this section we describe an efficient MWService protocol that provides the handoff capability across multiple Middleware Servers.

In [20] we showed that within a controlled local area environment, a single MWS can provide the ASH capability with a sufficiently low latency. It is natural to question why we would want to change our existing ASH protocol to handle the case of the new multiple MWS architecture since a sin-

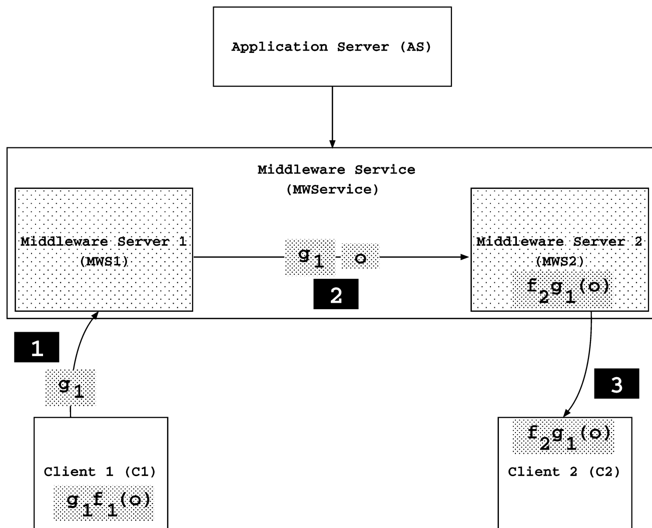


Figure 5. An Application Session Handoff utilising two Middleware Servers.

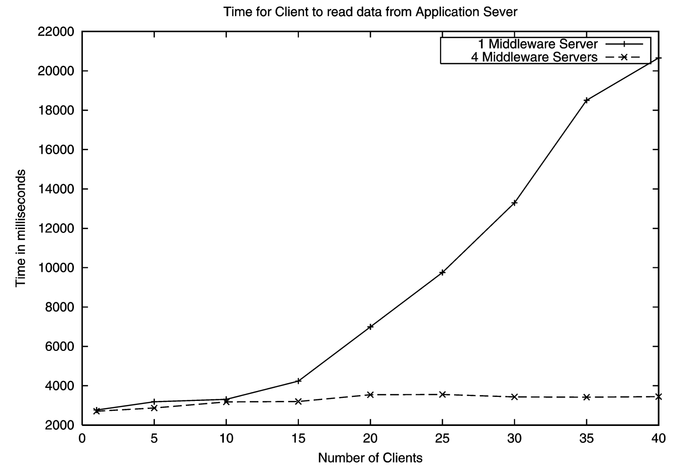


Figure 6. Average latency incurred by clients to read a data object from the AS, with admission-control load distribution.

gle MWS seems capable of facilitating the ASH. Previously in this section we presented an argument in favour of using the aggregate MWService layer to provide scalability for the general case of client-to-AS interactions. In this argument we stated an approach that led to having clients being serviced by multiple MWSes in response to some policy, such as load-balancing or client mobility. Such a conclusion is not novel but does affect the ASH facility in the following way. Consider the situation where a user is working on a client C_1 being serviced by a middleware server MWS_1. When the user wishes to handoff the application session to a second client C_2, it may be the case that C_2 is being serviced by a different server MWS_2. Clearly, the single-MWS approach to ASH is insufficient to handle such a situation because there exists session state that must be conveyed from C_1 through the MWService layer between MWS_1 to MWS_2 and then finally to C_2. It cannot be the case that MWS_1 can facilitate the ASH between C_1 and C_2 by itself; MWS_2 was chosen as C_2's server in the first place because MWS_1 was unavailable.

We now proceed through the series of events, depicted in figure 5, of the protocol that enables ASH across two middleware servers. Initially a middleware server MWS_1 has retrieved a data object o from the AS on behalf of client C_1. The MWS caches object o and also applies a content adaptation filter f_1 on o to match the characteristics of C_1, allowing the client to receive $f_1(o)$. The user at C_1 applies a modification g_1, resulting in an object $g_1 \circ f_1(o)$. At this point the user wishes to perform ASH from C_1 to another client C_2 being serviced by MWS_2. Upon ASH activation, (1) client C_1 sends the user modification g_1 to MWS_1 instead of the entire data object $g_1 \circ f_1(o)$ (recall from section 3.2 that sending solely the user modification is an optimisation we had derived). (2) When MWS_1 receives g_1, it is clear that it has all the pieces required to send the session state to C_2; to send $f_2 \circ g_1(o)$ to C_2, it has the original data object o which was cached, the f_2 data filter which is known for C_2, and the user modification g_1. However, MWS_1 cannot directly service C_2 for the same reason that prevented C_2 from being assigned to it initially. Instead, MWS_1 can determine through an intra-MWService facility that MWS_2 is servicing C_2. Hence, (3) MWS_1, with low cost, sends data object o and the user modification g_1 to MWS_2, which in turn (4) assembles $f_2 \circ g_1(o)$ to send to C_2.

5. Implementation and analysis

We now discuss experiments with our distributed MWService to shows its effectiveness with the Teaching File application. In our experiments we have utilised a heterogeneous workstation cluster of Sun Ultra-2 (at 200 MHz) and Ultra-5 (at 270 MHz) Solaris machines as our MWService, while our Application Server providing a data repository for our Teaching File was run on an Red Hat Linux 6.2 500 MHz Intel box. The client application was executed on a Sun Ultra-5 for these specific experiments, but it has been run previously on wirelessly connected laptops. A prototype version has also worked on a wireless Compaq IPAQ PDA. All the desktop machines were connected via high-speed ethernet. The specific platforms in this testbed were chosen due to availability and convenience factors. The Teaching File application and the MWService code are written entirely in Java using Sun's JDK 1.3. The Application Server is operated as an Apache webserver.

In our first set of experiments, we demonstrate the scalability of our MWService by showing its effect on normal AS-to-client data transfers. In 6 we graph the average latency experienced by clients to read a data object (with a fixed size of 250 kbytes) as a function of an increasing number of clients requesting service. We launch a new Teaching File client at random intervals based on a Poisson distribution with a mean of 15 seconds. Upon launching, each client requests the data object from the AS via the MWS and continues to make such requests at random intervals using a Poisson distribution with a mean of 30 seconds. For each value on the x-axis, we ran an experiment that lasted 5 request iterations past the time needed to launch the needed number of clients. The data plot for the 1-middleware server shows that the latency seen by the

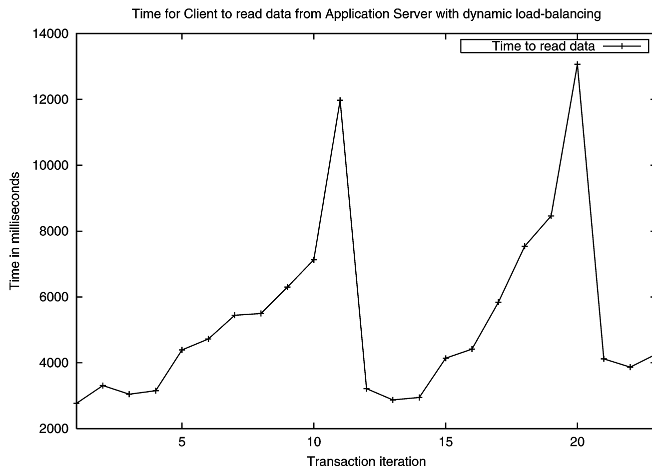


Figure 7. Latency incurred by a single client (out of 40) to read a data object from the AS, with dynamic load balancing.

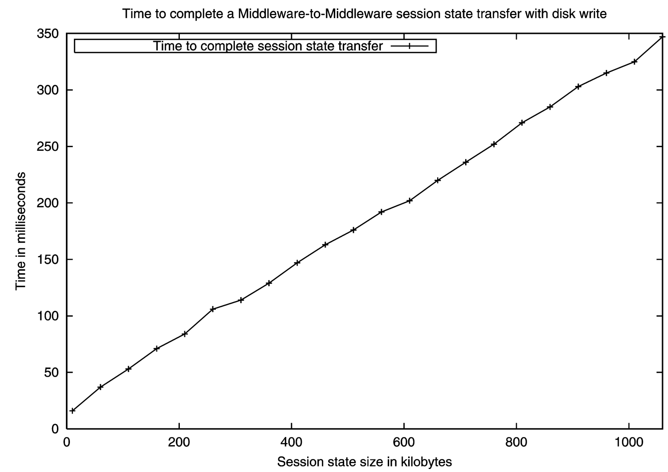


Figure 8. Results of synthetic experiment to perform a transfer of session state from MWS_1 to MWS_2 followed by a write of the session state to disk. The disk access accounts for 94% of the total time on average. The Teaching File data object being fetched in our experiments is about 250 kbytes.

clients grows without bound as more clients are introduced into the system. This growth is due to the fact that the MWS must perform the computationally-bound task of converting the PACS-formatted image object from the AS into a Java Image object.

For the 4-middleware server case, we implemented a very simple admission-control load distribution system. We had noted that for the 1-MWS experiments, the client latency began to grow significantly after 10 clients were launched. We then used this value to implement a control scheme in the 4-MWS system where at most 10 clients can be initially associated with a MWS; once the MWS is full, it will redirect further initial requests to another MWS in a sequential chain. We see that even for this simple admission control policy, the latency experienced by the clients for the 4-MWS case is kept at or below the relatively constant value of 3.5 seconds.

We have additionally implemented a simple load-balancing scheme to demonstrate our middleware-to-middleware design. It is important to note that load-balancing is but one policy that will trigger the MWS-to-MWS response. As mentioned in section 4.1, other reasons can include client mobility and bandwidth considerations. In figure 7 we show the effect of providing the MWS-to-MWS service for the Teaching File client by plotting the latency seen by a particular client to complete a data object read from the AS through the MWS with an increasing number of read transactions by that client as the independent variable. As the client continues to issue read requests over time, other clients are introduced into the system and make their own data requests using the random intervals discussed in the previous experiment. This client's time to completion increases as a result of this increased computational load at the MWS. In this experiment we implement a load distribution policy of having the MWS monitor its current CPU load by checking the average process queue length as returned by the Unix C library call `getloadavg(3C)`. When its load passes an arbitrarily chosen threshold, the MWS refuses further service. In the case of the client under observation in this experiment, it continues to request service from the MWS, but when the server becomes full, it is redirected for service at another MWS. With this easily implemented scheme, we see that the client's completion latency grows and then falls off, showing that the self-regulating load-balancing scheme provides scalability. The overhead of the middleware-to-middleware handoff is not directly visible in this graph, so we have provided a separate synthetic experiment to measure this cost. In figure 8 we show the results of a MWS-to-MWS transfer of state followed by a write to disk of the state by the second MWS. This figure shows the intuitive result that an increasing state size results in a higher MWS-to-MWS latency. We point out that disk access (over NFS) accounts for 94% of the times shown on average. It is important to note while the latency scales with the state size, even at the largest size of over 1 Mbyte, the absolute total latency is under 350 milliseconds, which is considerably smaller than the average latency of 3.5 seconds shown in figure 6 for the 4-MWS case. In the specific case of the Teaching File application, the state size was approximately 250 kbytes, which contributes only about 100 milliseconds to the overhead due to the MWS handoff.

In figure 9 we show the results from an experiment that shows the latency involved with a particular client performing a series of operations over time. Here, the client continually requests a data object from the AS and then performs an Application Session Handoff to another client. The second client in turn writes the data back to the AS. Again, other clients issuing read requests to the MWS are introduced into the experiment as previously described. Initially the clients perform the ASH using one MWS. However, as the load on the MWS increases past a threshold, the clients are again redirected to other MWSes, which corresponds to point 11 in the graph. In this case, each client is redirected to a different MWS. Although this particular policy is simplistic, it shows the beneficial result of an ASH using two intermediate MWSes. Clearly, once the clients have been sent off the initial MWS, the time needed to perform the operations falls off despite the overhead associated with the load-balancing.

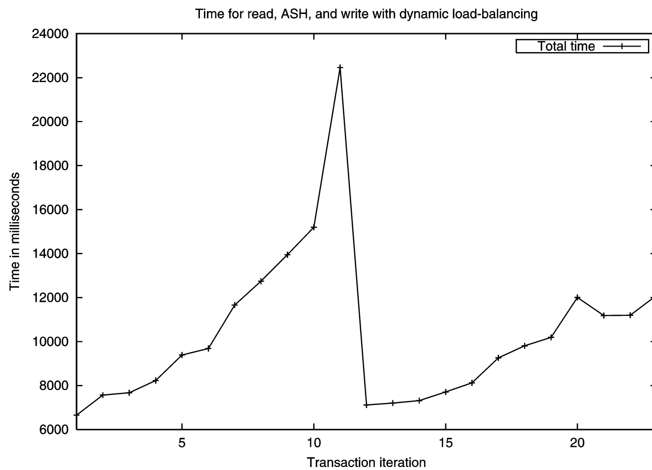


Figure 9. Total latency incurred by client C_1 to read data from the AS, C_1 to perform an Application Session Handoff to client C_2, and C_2 to save data back to AS, with dynamic load-balancing.

This result again shows the scalable, self-regulating behaviour we saw in figure 7, allowing for the overall performance to improve.

6. Related work

Our work builds upon a foundation of other research in the area of middleware architectures to support end clients. The Rover toolkit [13] aids applications in dealing with mobility and bandwidth issues. In Odyssey [17] feedback from the operating system level aided user applications to adapt to changing network conditions. A set of managers at the middleware layer designed to handle client problems was shown in [4]. In [5] researchers in the BARWARN effort developed a middleware tier that provides services such as content transcoding. The Ajanta research work [23] includes the use of mobile agents to aid in collaboration and coordination among users.

The architecture of our Middleware Service is similar in form to that of a Network of Workstations [1] cluster, but our service can be distributed over a wide or local area. In this paper we also discussed load balancing among a distributed set of computers. Load balancing [2,6,15,22] has been an active area of research and is a subject into which we do not delve. Its use in this paper is as a driving impetus and policy to trigger our various scalable mechanisms, such as the middleware-to-middleware session state transfer.

7. Conclusion and future work

In this paper we have developed a distributed Middleware Service architecture that provides reasonable performance when subjected to high workloads. We now revisit the basic tenets we stated in section 4 to further discuss the architecture's characteristics. (1) No one MWS causes a bottleneck across the entire system. As was shown in section 4, only two MWSes are involved in this handoff, and their actions do not have any effect on the rest of the MWService. Furthermore, actions performed at another MWS will not cause interference with these two MWSes. (2) No one MWS maintains any global state of system. From our design it can be seen that a given MWS is concerned with only the client(s) it is servicing and the second MWS to which it may need to handoff. (3) Each MWS acts autonomously. A given MWS makes decisions on its own regarding its availability. (4) Finally, the participation between two MWSes to perform the Application Session Handoff follows an efficient protocol. From our design it can be seen that we do not require any consistency management protocol because we assert that at any given time, the session state of an application exists either at the client, at a MWS, or is in transit. When session state is transferred from one MWS to another, we do not leave behind any residual state that is relevant to correctness, thus precluding the possibility of a distributed state. An application's state cannot simultaneously exist on two MWSes at any instance in time. Additionally, during the course of the handoff, the MWSes simply send data among themselves without any additional unnecessary computation or I/O. It is clear that latency will be incurred in these three steps, however, we have minimised the number of requisite actions that need to be perform to the preceding steps such that the overall handoff latency is low.

We will extend our work in the following manner. We will look into situations with more hardware heterogeneity, such as having less bandwidth, to see its effect on increased ASH frequency. We shall also investigate the properties of streaming data and its effect on the Middleware Service. As we noted earlier, the data used by the Teaching File have a discrete, connectionless characteristic that we leveraged. However, we plan to provide more research via implementation of our Service that aids continuous data. Additionally, we plan to investigate very wide area situations to study further scalability concerns using the with the GloMoSim [8] simulation package. Finally, we plan to utilise more advanced and interesting applications to serve as benchmarks.

References

[1] T. Anderson, D. Culler and D. Patterson, A case for NOW (Networks of Workstations), IEEE Micro (February 1995).

[2] D. Andresen, T. Yang and O. Ibarra, Toward a scalable distributed WWW server on workstation clusters, Journal of Parallel and Distributed Computing 42 (1997).

[3] R. Bagrodia, M. Gerla, S. Lu, R. Meyer, D. Valentino and L. Zhang, Supporting nomadic healers, UCLA Technical Report 200021 (2000).

[4] A. Bond, M. Gallagher and J. Indulska, An information model for nomadic environments, in: *IEEE Proceedings of the Ninth International Workshop on Database and Expert Systems Applications* (1998).

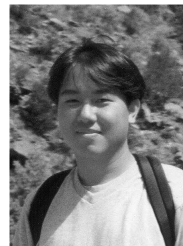
[5] E. Brewer, R. Katz, E. Amir, H. Balakrishnan, Y. Chawathe, A. Fox, S. Gribble, T. Hodes, G. Nguyen, V. Padmanabhan, M. Stemm, S. Seshan and T. Henderson, A network architecture for heterogeneous mobile computing, IEEE Personal Communications (October 1998).

[6] T. Chou and J. Abraham, Load balancing in distributed systems, IEEE Transactions on Software Engineering (July 1982).

[7] A. Fox, S. Gribble, Y. Chawathe and E. Brewer, Adapting to network and client variation using infrastructural proxies: Lessons and prospectives, IEEE Personal Communications (August 1998).

[8] The Global Mobile Information Systems Simulation Library homepage, pcl.cs.ucla.edu/projects/glomosim/
[9] C. Goldfarb and P. Prescod, *The XML Handbook* (Prentice-Hall, Englewood Ciffs, NJ, 2000).
[10] E. Guttman, Service location protocol: Automatic discovery of IP network services, IEEE Internet Computing (July 1999).
[11] M. Handley and J. Crowcroft, Network text editor: A scalable shared text editor for the MBone, in: *Proceedings of the 1997 ACM SIGCOMM Conference* (1997).
[12] IBM/Lotus Notes homepage, www.lotus.com/home.nsf/welcome/lotusnotes
[13] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford and M. Kaashoek, Rover: a toolkit for mobile information access, in: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 95)* (1995).
[14] L. Lambrinos, P. Kirstein and V. Hardman, The multicast multimedia conference recorder, in: *Proceedings of the 7th International Conference on Computer Communications and Networks* (October 1998).
[15] M. Litzkow, M. Livny and M. Mutka, Condor – A hunter of idle workstations, in: *Proceedings of the 8th International Conference of Distributed Computing Systems* (June 1988).
[16] Microsoft Exchange homepage, www.microsoft.com/exchange/
[17] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn and K. Walker, Agile application-aware adaptation for mobility, in: *Proceedings of the 16th ACM Symposium on Operating System Principles* (1997).
[18] Novell Groupwise homepage, www.novell.com/products/groupwise/.
[19] T. Phan, K. Xu, R. Guy and R. Bagrodia, Handoff of application sessions across time and space, in: *The IEEE International Conference on Communications (ICC 2001)*, Helsinki, Finland (June 2001).
[20] T. Phan, R. Guy, J. Gu and R. Bagrodia, A new TWIST on mobile computing: Two-way interactive session transfer, in: *The 2nd IEEE Workshop on Internet Applications (WIAPP 2001)*, San Jose, CA (July 2001).
[21] The phpGroupWare homepage, www.phpgroupware.org/
[22] B. Shirazi, A. Hurson and K. Kavi (eds.), *Scheduling and Load Balancing in Parallel and Distributed Systems* (IEEE Comput. Soc. Press, 1995).
[23] A. Tripathi, T. Ahmed, V. Kakani and S. Jaman, Distributed collaborations using network mobile agents, in: *Proceedings of ASA-MA 2000 Joint Symposium – 2nd International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, Zurich, Switzerland (13–15 September 2000).
[24] D. Valentino, Considerations in implementing large-scale PACS, in: *Conference on RIS, PACS and Teleradiology: Future-Proof Solutions*, Lubbock, TX (1998).
[25] J. Waldo, The Jini architecture for network-centric computing, Communications of the ACM 42(7) (1999).
[26] M. Weiser, The computer for the twenty-first century, Scientific American (September 1991).

Rajive Bagrodia is a Professor of Computer Science at UCLA. He obtained a Bachelor of Technology in electrical engineering from the Indian Institute of Technology, Bombay, in 1981. He obtained his M.A. and Ph.D. degrees in computer science from the University of Texas at Austin, in 1983 and 1987, respectively. Professor Bagrodia's research interests include nomadic computing, parallel simulation, and parallel languages and distributed systems. He has published over a hundred research papers on the preceding topics. The research has been funded by a variety of government and industrial sponsors including the National Science Foundation, Office of Naval Research, DARPA, Rome Laboratory, and Rockwell International. He is an associate editor of the ACM Transactions on Modeling and Computer Systems (TOMACS). He was selected as a Presidential Young Investigator by the National Science Foundation and won the TRW Outstanding Young Teacher award.
E-mail: rajive@cs.ucla.edu



Thomas Phan is a Ph.D. candidate in Computer Science at UCLA. He received his B.S. in computer science and engineering from the University of California, Davis in 1996. His research interests include parallel and distributed computing, ubiquitous computing, programming languages, and parallel simulation.
E-mail: phantom@cs.ucla.edu

Richard Guy received his M.S. and Ph.D. degrees in computer science from UCLA in 1987 and 1991, respectively. He has been a research scientist in the UCLA Computer Science Department since 1996. His research interests include distributed operating systems, networks, and pervasive computing.
E-mail: rguy@cs.ucla.edu