iMASH: Interactive Mobile Application Session Handoff

R. Bagrodia[*], S. Bhattacharyya, F. Cheng, S. Gerding, G. Glazer,
R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, G. Zorpas
UCLA Computer Science, Los Angeles, CA 90095-1596

Abstract

Mobile computing research has often focused on untethering an in-use computing device, rather than enabling the mobility of the computation task itself. This paper presents an architecture, implementation, and experimental evidence that together validate a new continuous computing concept, application session handoff. The iMASH architecture leverages previous work on proxies, content adaptation, and client awareness to provide a unique, middleware-enabled capability for continuous computing. Implementation in both socket- and RPC-based environments shows that very fast, secure session handoff of non-trivial client/server applications across heterogeneous client devices and networks is feasible: experiments on a number of applications yielded handoff latencies ranging from 0.5s to 2s.

1 Introduction

July, 2005

Maria Salas, eight months pregnant, is injured in a car accident. Jim Brown stops to help and seeing Maria, he uses his wristwatch to call 911 while sprinting back to his car to get his broadband wireless PDA. Jim transfers his conversation from watch to PDA as he returns to Maria, since the PDA has bi-directional slow-scan video support.

The PDA also has a personal area network transceiver, which is used to discover and communicate with Maria's medical alert chip. This data is forwarded to the 911 operator, who uses it to identify and notify Maria's physician.

Dr. Hughes is out walking in the nearby mountains when her watch chimes urgently. Glancing at the display, she sees "patient injured," pulls out her PDA and instantaneously transfers the messaging session to the PDA. A PDA-based application retrieves medical history data about Maria as Dr. Hughes jogs back to her car.

As she begins driving, her PDA session is transferred to her car computer, which has a powerful processor, display, and voice-recognition. This system retrieves from Maria's medical records a range of image and video data that exceeded the PDA's capabilities. As she enters the city limits, network support is transparently switched from the rural wide-area network provider to a metropolitan-area network provider with higher bandwidth. The Medical Center computing infrastructure senses the switchover, and upgrades the quality of the image data it is sending.

Dr. Hughes joins a video-conferencing session with the 911 operator, emergency room, Jim, and en route rescue personnel. Guided by the emergency room and Dr. Hughes, Jim is able to apply limited, but situation-specific life support measures until the rescue team arrives.

As Dr. Hughes enters the Medical Center parking garage, her session transfers back to her PDA, which has a high bandwidth connection available on campus. She remains in contact and while waiting for the elevator, she consults further chart details and enters some additional notes. As she enters the emergency room, she transfers her PDA session to a system with wall-mounted displays so the trauma team can easily see it. The transferred session includes partially completed progress notes not yet formally saved in the patient records database.

Maria soon arrives, and the trauma team and Dr. Hughes are ready for her: they already know most of what they need to do to heal her and save her unborn child. In fact, the rescue team has already begun critical steps under the joint guidance of the trauma team and Dr. Hughes.

Although much of the technology implied in the scenario exists today, either as off-the-shelf products or research prototypes, a critical missing link is software support for *continuous computing*, which would enable a session to move seamlessly between heterogeneous platforms. A primary goal of the iMASH project is to develop the concepts, architecture, and prototype implementation to provide effective support for such a service. A key enabler is *application session handoff* which leverages existing work on proxies, content adaptation and client-awareness to provide a unique capability for continuous computing that satisfies the problem constraints outlined below.

The core concept, challenge and claim of application session handoff is that it is possible to easily identify a subset of application state that concisely captures the se-

[*]This research is supported by NSF Grant ANI-9986679. Contact authors at {rajive, sbhattac, fred, sbg, glenn, rguy, jizr, jinsong, phantom, eskow, maneesh, zorpasg}@cs.ucla.edu

mantics of a partially completed computation; efficiently transfer that state subset to a different client device; and effectively resume work on that target platform—all with very low latency while satisfying appropriate security, scalability and heterogeneity constraints.

Each of the above three steps is a challenge in its own right. Our main effort to date has been on the second challenge of efficiently transferring state to another, possibly heterogeneous client device, and that focus is reflected below. The key insight in our approach is that object type knowledge can be exploited to manage the amount of state that needs to be transferred between clients, in much the same manner that proxy transcoding manages the amount of state delivered from server to client.

In this paper we outline application session handoff, detail a supporting architecture and its implementation, and validate the concept with experimental evidence.

1.1 Problem Characteristics

The continuous computing problem space is quite large and diverse. Our particular problem domain, medical informatics, displays a multidimensional character that constrains the solution space in a number of ways:

client/server computing: Much of medical informatics today uses this model; other models (e.g., peer-to-peer) clearly warrant future research.

heterogeneous client devices: Up to six orders of magnitude routinely distinguish client devices in dimensions such as CPU speed, memory, secondary storage, display area, input ports, and network bandwidth and latency. The permutations of dynamic conditions over these ranges effectively yields enormous heterogeneity.

heterogeneous network infrastructure: In a hospital, most intranet infrastructure can be provisioned with high-capacity, low-latency components, but the last hop characteristics will likely vary widely.

user mobility: Highly mobile users traverse a range of settings in which physical heterogeneity imposes diverse constraints on computation and/or communication capabilities, in turn motivating a user to dynamically (and unpredictably) choose from a spectrum of client platforms. This implies that content adaptation is essential: data delivered from server to client should be tailored to satisfy constraints imposed by current network conditions and device characteristics.[FGBA96]

Mobility further introduces a new problem: the need for in-progress computation and communications to move across heterogeneous client platforms with low latency. The combination of movement, diversity, and timeliness has been an unsolved problem.

legacy servers: Medical informatics administrators are conservative about "enhancements" to their server environments (cf. [vMCT95]). Any novelties we propose need to leave a legacy server largely untouched: no additional code may be added to the server software or even share the processor, and no changes in the semantics of client/server interaction are allowed.

client application-aware: It is acceptable for client applications to be aware of changes to the legacy environment. In the spirit of Odyssey[Nob97], we believe that application awareness can be beneficial to both the application and the enhanced system environment. Client awareness implies that the application be amenable to augmentation: source code is available, and semantic knowledge is accessible.

diverse data types: Medical databases contain high-resolution still image series (gray-scale, false-color, full-color), gray-scale medium-resolution video, dictation audio, and traditional text-based records.

These domain-driven characteristics have shaped the development of application session handoff and the resulting iMASH architecture. Our work is applicable to other domains to the extent that they generally share the same criteria; a change of constraints might enable (or preclude) different architectural choices.

1.2 Related work

Our work builds on several well-known areas (proxies, content adaptation/transcoding, client-awareness) and is complementary to recent work on continuous computing. The seminal work on proxy-based content adaptation is the BARWAN project[FGBA96], in which the argument for adapting data on-the-fly is made. Recent work by Lum and Lau [LL02a, LL02b] focuses on the decision-making aspects of content adaptation, including both the complexity of making such decisions, and the trade-offs between on-the-fly approaches and storing pre-adapted version of data. The benefits of client-awareness of environmental changes in Odyssey are presented in [Nob97].

Relevant recent work in continuous computing includes the One.World project at the University of Washington [GDLB02], which looks at the pervasive application migration problem from the program development environment, and proposes a structured view of communication channels that requires the explicit disconnection and reconnection of servers. (iMASH very deliberately preserves open channels from a server's perspective; our work at the application level contrasts with the approaches of Snoeren [SB00] and Zandy [ZM02] at the network level.)

One.World also argues for a homogeneous execution platform, to support the movement of code, where we presume code is already resident on a device. An exhaustive treatment of conventional process migration is found in [MD00]; our focus on heterogeneity precludes the direct application of these techniques. Roman et al. [RKC01] propose the use of reflective middleware to support continuous computing across heterogeneous platforms.

Early portions of iMASH research have been presented at a handful of workshops [PGB01] [LGGB02] [PZB02] [SKP02] ; this paper provides an overall integrating context for that work, and presents new experimental data based on an implementation of the entire architecture.

1.3 Road map

The next section provides an overview of application session handoff (ASH). Section 3 describes an architectural design that implements iMASH handoff. The architecture is followed by a report in Section 4 on our experience with iMASH, in the form of a series of experiments on various iMASH-enabled applications. Conclusions and future work are presented in Section 5.

2 Application Session Handoff

Application session handoff addresses all of the characteristics delineated in Section 1.1. The most challenging of these are heterogeneous client devices and networks, especially when poorly provisioned in these regards. Impoverished clients are unable to use emulation to execute foreign tasks and low-bandwidth network links can not deliver large data objects or sizable binary process images in a timely fashion. The ASH approach looks through the opaque shroud that traditionally envelops an executing process, so that it can apply needed conversions to essential state that allow for the computation to move to and continue on the target device. iMASH enlists the assistance of the application itself to identify essential application state, a step we call *semantic savepointing*. The state is transferred to an intermediate host, where conversions are applied to hide the heterogeneity between source and target clients. The converted state is then delivered to the target client, where it is used by a target-native version of the application to "restart" where computation paused on the source client.

This client-aware approach is surprisingly well supported by existing applications—in particular, those that savepoint state for recovery (e.g., Microsoft Word 2001). Such an application is already structured with preservation of essential state as a first order concern, and also is prepared to resume execution based on well-formed state. This issue is explored further below. Modern applications are also commonly re-targeted to heterogeneous platforms; versions of the Microsoft Office suite, for example, have been created for Windows NT workstations and PocketPC PDAs. Note further that content adaptation is increasingly common, though typically on data delivered from server to client; content adaptation as an integral feature of handoff is novel.

In client/server environments, the bulk of data used by a client application is provided by server(s). ASH provides proxy-based content adaptation to ensure that data delivered to a client is compatible with client constraints, and arrives in a timely fashion. Content adaptation is driven by both relatively static user, device, and application profiles, and typically dynamic network profiles. A proxy is employed to shield legacy servers from change, and to relieve potentially weak or overburdened clients from the complexities and costs of content adaptation.

ASH anticipates narrow last-hop network links in the upstream direction as well, and so it exploits caching at the proxy to eliminate the need to move server-supplied data *from* the client. Newly created data, of course, must be transferred explicitly. ASH also incorporates, with the assistance of the application, a facility that enables client operations on data to be concisely shipped and quickly re-applied to a cached data copy. Together, these two features enable the first half of ASH ("session suspend") to be accomplished with low latency.

The second half of ASH, session resume, involves the delivery of session state to the target client and initializing the target application. ASH satisfies the constraints of the target client and its network environment in part by content adapting the session state before delivery. ASH also supports the possibility that the target client application is sufficiently different from the source client version that it may only desire a subset of session state elements. Prior to state delivery to the target, an XML description of the state elements is given to the client, which in turn requests the session (sub)set of interest. It is this subset that is adapted and delivered.

Session resume, like session suspend, is designed to operate with low latency. Our experimental evidence (see Section 4) shows this to be the case, except when a poorly ported application requires excessive resources on a weak client—such as a large Java application on a slow PDA. In such situations, modest latency (<10s) handoff results.

A client application under consideration to be iMASH-enabled is required to be a suitable candidate for *semantic session savepointing*: it must be feasible for an application programmer to identify points in the application's execution at which the essential semantic state can be frozen. Interactive applications tend to naturally have such points, even when multi-threaded, as a side-effect of mechanisms that support user interaction. In many cases, application session handoff will be formally instigated by the user via the legacy interaction mechanism.

Note that in general, saving client state back to the server, terminating the local execution and starting an application on the target device (conventional "checkpoint/restart") is not acceptable: the client state may be incomplete, and thus either is not acceptable to the server, or is not appropriate to expose to other clients.

Freezing client/server interprocess communication (IPC) requires a bit of care. Discrete object transfers are straightforward (simply wait for the entire object to be delivered), but streaming data is more difficult: in general, handoff should occur at a "good" point in the stream, which is at the least a protocol-specific issue. For example, a reasonable point to freeze processing of a motion JPEG stream is at a frame boundary; but for an MPEG stream, a full "group of pictures" is the proper semantic unit boundary. A correct freeze may require cooperation between the client application and the session handoff service [LGGB02]. (See also Section 3.3.1.)

We have iMASH-enabled a variety of applications, including several web browsers, a paint program, a remote shell utility, a radiology teaching tool, and various video players. In some of these cases, source code was available but adequate documentation was not, which made the tasks of implementing semantic savepointing and session restart non-trivial. (For example, the remote shell had about 100 separate elements of session state.) In others, source code was *not* available, and yet we were able to substantially support application session handoff for these applications by creating a simple "wrapper" application for each that is itself iMASH-enabled and in turn executes the black-box application as a child process.

The Galeon browser (http://galeon.sourceforge.net), was iMASH-enabled in *under one hour* by using the wrapper technique. In this case, we leveraged Galeon's existing recovery mechanisms that aggressively maintain current application state (e.g., per-window history and onscreen window placement) in a set of XML-based files. The wrapper simply re-packages the files as session state during handoff, and the target side wrapper creates files expected by Galeon on startup. Additional time (about a day) was required to enhance the wrapper with an HTTP proxy capability so that Galeon's HTTP requests would route through iMASH, thus enabling iMASH content adaptation benefits. By leveraging iMASH's extensible content adaptation architecture (see Section 3.3.2), just one more day was invested in designing and building a content adapter for the XML-based state, so that automatic device-specific window resizing is performed during handoff, as shown in Figure 1.

iMASH-enabling an application could be greatly eased and possibly even fully automated with appropriate application programming language constructs and compiler hints/directives available to the programmer. This is a topic of ongoing research (cf. One.World [GDLB02]).

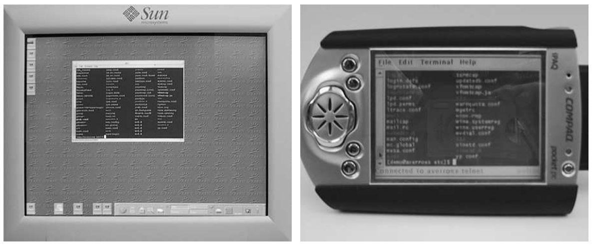


Figure 1: Pre- and post-handoff views of an application session handoff from a 19" workstation to a 4" PDA screen. Note the large application window has been resized to fit the smaller display.

3 iMASH Architecture

In our problem domain, the two most significant factors affecting architectural design decisions are client device heterogeneity and unaware (immutable) legacy servers. The former argues against an operating system-level architecture, to avoid repeated re-implementation for each operating system; the latter implies that a conventional middleware-level client/server solution is inadequate, because no place exists to host "server side" middleware. Further, clients are often ill-suited to hosting additional computation burdens.

The iMASH solution is middleware-oriented, but we place minimal middleware burden on clients—and none on legacy servers—by introducing a *middleware service* hosted by (new) additional platforms placed between clients and legacy servers. This service acts as a client proxy to legacy servers, and is in the critical path between clients—effectively a proxy between them.

The architecture material is organized in three parts: Section 3.1 motivates the need for multiple middleware servers to instantiate the overall middleware service; the resulting impact on application session handoff is then considered in Section 3.2; and in Section 3.3 we delve into the details of middleware server structure and operation.

3.1 Middleware service

In general, the middleware service must be reliable, secure, scalable, transparent to legacy servers, low latency, and impose low overhead on clients—all while providing its basic service, application session handoff. To achieve this, we believe that the middleware service must itself be a distributed system. The iMASH middleware service is provided by a collection of *middleware servers* that work cooperatively to support application session handoff. Figure 2 shows this structure.

A middleware server (MWS) mediates *all* communication between client and server, and it additionally maintains application session state on behalf of clients so as to assist with low-latency application session handoff between client devices. It also provides the bulk of the se-

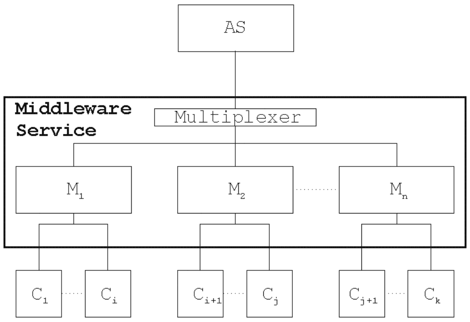


Figure 2: Middleware server architecture, with a number of middleware servers (M_n) on the middle tier and a multiplexer just above, and their current clients (C_k) on the lower tier.

curity services on which the above two functions rest.

3.1.1 Multiple middleware servers

The iMASH middleware service is routinely expected to perform potentially time- and space-expensive computation (e.g., adaptation, encryption, session state storage). With thousands of client devices and application sessions anticipated in a large hospital, it is reasonable to project the inadequacy of a single middleware server, and so a scalable architecture that allows a number of middleware servers is essential.

There are additional benefits to multiple middleware servers, if the system is properly designed. For example, multiple middleware servers also enable the deliberate placement of middleware servers at distinct points in a network topology, such as close to a wireless base station. One could also place a MWS at a remote location, so as to position adaptation closer to the client device and user. As a client device moves geographically, the "best" path between an application server and client may change and thus motivate a change of middleware server. Even when a client is stationary, the presence of network congestion or other conditions might spur a change of middleware: a sufficiently better path to the client device might exist from a different MWS, which could justify the cost of a change. The ability to change middleware servers on the fly further allows for dynamic load balancing (see [PGB01]) at both session admission and intra-session.

The need to maintain legacy application server unawareness of iMASH conflicts with the need to change middleware servers: the middleware proxy role hides the client application from the legacy server, but naturally substitutes its own visibility. iMASH deals with this problem by introducing a very thin service layer ("mux") which multiplexes communications between a legacy application server and middleware servers.

The mux has no role beyond hiding a change of middleware server from the unaware legacy server. From the application server's perspective, the mux *is* the client. Following instructions from the middleware servers, the mux merely directs (or redirects) a stream to the appropriate MWS. In our testbeds, we typically implement the mux as an application-level router on a conventional PC; there are obviously programmable router solutions that would be expected to incur much lower latency. If the legacy server operating system supports mobile sockets [ZM02], the mux could be eliminated entirely.

3.2 ASH types

To jointly support both multiple client devices and multiple middleware servers, iMASH provides application session handoff in two directions: client handoff, in which the device executing an application changes; and, middleware server handoff, in which the middleware server supporting the session changes. The resulting three handoff types are depicted in Figure 3.

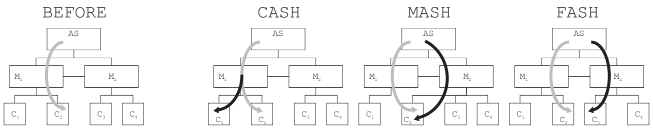


Figure 3: The three types of application session handoff. The gray arrow indicates a pre-handoff data flow from an Application Server (AS) through the middleware service to a Client (C). The black arrow shows the corresponding post-handoff relationship for the respective types of handoff.

We use CASH to denote a client-only application session handoff in which both source and target clients are supported by the same MWS. We use MASH to denote a middleware-only application session handoff, in which the client application continues execution on the same device while the session support switches to a different MWS.[1] We use FASH to denote a third type of handoff, full application session handoff, in which a new client and a new MWS support the session after handoff.

The motivation for CASH is driven by the need for a mobile user to change devices when moving among heterogeneous environments. CASH is almost always user-initiated since the user is switching physical devices. This implies that CASH generally occurs on a coarse time scale, with a typical frequency of minutes to days. Users may also be willing to tolerate a few seconds of handoff delay. In the opening scenario, the session transfer from PDA to car computer is an example of CASH.

MASH is driven by infrastructure concerns: scalable performance and flexibility in network topological and geographical placement, and responsiveness to network performance issues. MASH is rarely user-initiated, and

[1] MASH is loosely analogous to cellular telephony handoff between cell sites, but occurs at an application level quite independent of the network level.

so must appear to be delay-free. In the scenario, the WAN-to-MAN city limits switchover is a MASH.

FASH is much closer to CASH than MASH in both spirit and design: like CASH, it is driven by the user, and the change of middleware servers is an internal side effect. An example of FASH in the scenario is the session transfer from PDA to emergency room computer.

3.3 Middleware server architecture and operation

The iMASH middleware server architecture supports application session handoff through a variety of mechanisms, as diagrammed in Figure 4. Here we describe several of the most important ones, including session handoff management, content adaptation, and security.

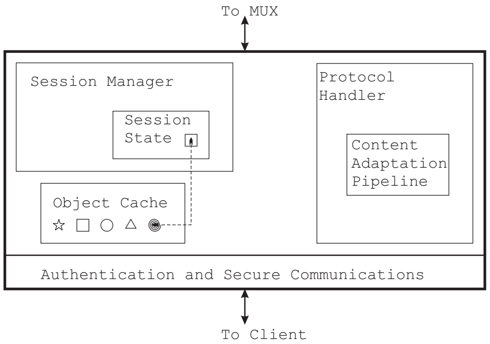


Figure 4: The middleware server architecture, showing the Session Manager, Object Cache, Protocol Handler, Content Adaptation Pipeline, and Security Layer.

3.3.1 Handoff Management

Session handoff management naturally divides between the types that require a semantic session savepoint (CASH and FASH), and the one that does not (MASH).

Savepoint-based Handoff Effective savepoint-based handoff relies heavily on the ability to exploit actions taken in the normal course of events that occur in client-to-MWS-to-legacy server interaction. In particular, iMASH caches at the MWS discrete (i.e., non-streaming) server-supplied objects for later use in session handoff.

Recall from the opening scenario Dr. Hughes' use of the PDA to retrieve Maria's medical records. An application on the PDA is executing and making queries of a legacy medical records server at the (remote) hospital. The middleware server mediating the connection recognizes that the requested medical record's complex data structure includes several data types for which the PDA is generally inadequate: high-resolution images whose size exceeds the PDA's memory, and video which exceeds the PDA's processor speed to display. None of the instances of these types are passed on to the PDA client in their original form; however, the images are cached at the MWS, and "thumbnails" (content-adapted versions) are delivered to the PDA, along with unmodified text-based data objects.

At this point, the PDA-based client application has received a mix of original and content-adapted data objects from the legacy server, via the MWS. Upon entering the car, Dr. Hughes initiated a CASH from the PDA to car computer—which happens to be different (in this case, more capable) in many key respects, especially CPU speed, memory capacity, and display size. Ideally, the transferred session will automatically incorporate data object versions appropriate to the target device, unbiased by the source device's limitations.[2]

To effect the handoff, the current essential application state from the PDA must be collected and shipped to the middleware server, after which the source client application is terminated. At the MWS, the session state is cached, content adapted as needed, and delivered to the target device for insertion in the target application.

In our architecture, the handoff request is forwarded by the iMASH-enabled application through the *iMASH client layer* support library (previously linked into the application) to the MWS currently serving the session. The MWS verifies that the handoff is authorized (see Section 3.3.3), and if so, arranges with the target client (and target MWS, if a FASH) to be prepared for session handoff. In particular, a session skeleton is prepared on the target client, and the target application begins execution and waits for the savepoint to arrive. The (source) MWS then invites the source client application to savepoint.

iMASH application state is divided into three types: *server objects*, i.e., those obtained from the legacy application server, such as an X-ray image; *application objects* created locally by the client application, such as text comments about the image; and *private objects* created by the application, such as a temporary buffer holding copies of the image and comments. An application savepoint only includes server objects and application objects: private objects are those not intended to be part of session state. Server objects are logically, not physically, part of the savepoint, as they are already stored at the MWS, having been cached there on the way from the application server to the client. Application progress is represented explicitly as an application object in the savepoint.

When the savepoint is complete—which might not be immediately, depending on application activity and semantics—it is transferred to the source client's MWS. At this point, the source client cleans up its local session state and exits, retaining no session knowledge at all.

[2]This *must be the case* to avoid a greatest common denominator effect, in which the least capable client constrains the session's future—and thus neuters the value of handoff.

The MWS merges the savepoint with any existing stored session state, and sends a summary list of session state elements to the target client. For FASH, the session state is first copied to the target MWS, which then finishes the handoff interaction with the target client.

The target client selects which elements of session state it wishes to receive, and makes an aggregate request to the MWS for them. The MWS sends copies of the selected session state elements to the target client after possible adaptation. When the target client has received all of the requested elements, it initializes its application-specific semantic state and "resumes" execution.

The extra hop through the middleware server is essential: clients are not presumed to have communications hardware and services in common; weak clients are not forced to deal with heterogeneity-induced data format conversions; and low-latency handoff requires an economy of upstream data movement which middleware servers can provide by caching objects and other application session state. The MWS maintains session savepoint data as a part of the session state—even after execution resumes at a target client device.

Storing a copy of server-supplied objects at the MWS is a key issue in providing very low latency application session handoff: while a number of large objects may have been received by the client application over a long period of time, handoff must be accomplished in a short interval. iMASH must avoid moving data from client to MWS wherever possible, since in many interesting scenarios the upstream channel is severely constrained.

Further, client heterogeneity often makes the version of an object at the source client uninteresting elsewhere: adaptation done to the object on its way from the server to the source client is client-specific. A very different adaptation probably should be performed for that object when destined (during ASH) to a heterogeneous client.

Middleware-only Handoff Middleware-only application session handoff is different than CASH and FASH because the client is typically not the initiator. Instead, the middleware server initiates the handoff, perhaps due to its awareness of a better path to the client through a different MWS, or perhaps because the current MWS is overloaded. This is a significant difference, because in CASH and FASH the client explicitly chooses when to do a handoff, and latency incurred at that point may be accepted by the client as the cost associated with the benefit. But in MASH, the user does not routinely participate in the middleware handoff decision, and so should not be noticeably penalized by handoff latency. We are exploring both passive (upcalls) and active (downcall, polling) approaches to communicate important changes in network conditions to iMASH. The middleware prototype responds to an external application-level network monitoring task that periodically assesses packet delay.

Once a MWS has decided that a MASH should occur, it contacts the target MWS and indicates its desire for a MASH. The target MWS has an opportunity to deny the handoff (perhaps it is overloaded), but in the successful case it prepares a skeleton session in anticipation of receiving session state from the source MWS. Upon receiving a positive acknowledgment from the target MWS, the source MWS notifies the mux that it should immediately redirect specific streams to the target MWS.

The mux knows nothing about the content of a stream flowing through it from the application server to a middleware server. Therefore, a stream is redirected at an arbitrary point. The target MWS must buffer the redirected stream, until it has received all session state from the source MWS. In addition to routine session state, the source MWS must also forward any buffered stream data that has not been processed, so that this data may be prepended at the target MWS to the front of its buffered stream data to preserve the correct stream data ordering.

3.3.2 Content Adaptation

Content adaptation has been well-trod ground for several years [FGBA96, Nob97, LL02a, LL02b]. iMASH exploits content adaptation to ensure that only client-device appropriate data is delivered to a client application, whether the data is coming directly from a legacy application server, or indirectly in response to application session handoff. The former case is analogous to standard proxy-based content adaptation [FGBA96]; the latter is an iMASH-unique variation which relies on a middleware server-cached copy of an original object as a basis for handoff-induced content adaptation.

The basic iMASH content adaptation architecture is fairly conventional: it includes tools that recognize ISO/OSI Layer 4 and above protocols, parse as needed to extract objects (possibly deconstructing complex aggregate objects), perform selected content adaptation, and reinsert the adapted object(s) into a stream of a type expected by the client application. The architecture is extensible, in that it anticipates new protocols and data types. It also allows for a series of adaptations on an object, much like a UNIX-style pipeline of filters.

Profile-driven Content Adaptation The general content adaptation problem is one of constraint satisfaction: given an object with certain characteristics, a set of constraints on those characteristics which must be met, and a set of adapters which can transform characteristics, which adaptation(s) should be applied in which order to minimally meet the constraints?

iMASH employs a three-step process—data characterization, command generation, and pipeline execution

[PZB02]—which today relies on simple heuristics to determine what adaptations to apply. Each of the three steps is designed for extensibility, so that additional data types and adaptation functions can be added readily.

Object characterization is performed by a type-specific function which provides an XML representation of an object's attributes. If an object has no matching characterization function, iMASH simply passes the opaque object on the client with no adaptation.[3]

Constraints are obtained from several sources such as client device, application, user, and network, and represented as XML profiles. Profiles are by nature extensible. A *client device profile* typically describes the processor speed, memory capacity, display dimensions, and nominal bandwidth of the currently-in-use network interface. A *user profile* often contains user preferences such as a "patience factor" to express the duration of a tolerable per-object latency. A *network profile* describes recent dynamic network conditions; ongoing work is examining a tight integration of network quality of service information with content adaptation, especially of streaming data. Profiles are merged within the pipeline, giving general precedence to the most-restrictive constraints, to produce the set of constraints to be satisfied. A profile can also explicitly indicate that *no* adaptation may be performed.

The object characteristics and constraints are then fed into the *command generator*, where they are compared in a pre-determined order, and adapters are chosen which can transform the object to satisfy the constraints. A sequence of adapters is then executed on the object, and the final resulting object is given to the protocol handler component for re-insertion into the protocol stream and ultimately, delivery to the client.[PZB02]

3.3.3 Security

iMASH enables and encourages a much larger number and variety of client devices than the legacy systems which it enhances, and many of these clients are expected to be mobile. Mutual authentication between middleware servers, and between clients and middleware servers, must be accomplished on a large scale and done with very low latency during handoff.

In the medical domain (and others), information privacy and integrity is essential. Recent U.S. legislation imposes significant potential liability on those who deal with medical data [Hel00]. Today's mobile computing environment relies heavily on wireless technology with well-known exploitable weaknesses (e.g., see [BGW01]), so it is prudent to ensure privacy and integrity above this level. We also want to limit exposure resulting from a stolen client device or compromised middleware server.

[3]This is a simple policy decision. Another alternative is to return a null object.

Further, the novel iMASH notion of application session handoff raises new security issues above the network layer: How will trust relationships be transfered and maintained during handoff? How to transfer/adapt session encryption keys to ensure privacy when transferring a session to a new device? Who is authorized to trigger a session handoff, and to (and from) what client devices? What policies should govern such authorization, and what components enforce them?

Basic Solutions iMASH uses a bi-level security framework, layering a user/session authentication and authorization over a device level authentication [SKP02]. As the first step in (re)joining an iMASH network, a client device performs a mutual authentication with a single middleware server.[4] The authentication protocol and all subsequent data communication use WTLS[WAP], which uses a certificate-based, public key authentication procedure followed by the use of secret key symmetric encryption to protect data flowing on possibly lossy transport services (e.g., UDP over wireless). Each device and MWS has a unique certificate issued from a trusted authority; the certificate is assumed to be stored in a tamper-proof container on the device. We assume our certificate authority is scalable, reliable, and accessible.

Because critical steps in the public key encryption methodology are very computationally expensive—up to ten seconds or more to encrypt a 1KB block on a modest PDA—iMASH is designed so that it only requires a client device layer authentication when the client first joins the network, which in a fully iMASH environment is at boot time.[5] When the authentication handshake is complete, a low-cost, secure, encrypted, channel exists between the parties that uses a symmetric encryption algorithm[6] key produced dynamically during the handshake.

This initial *device control channel* is used to enable extremely quick generation and exchange of additional keys which in turn are used to establish secure *session control channels* (one per session) and *session data channels* between a client application and a MWS used for mediating communication between client and legacy application server. Each session has a distinct session control channel based on a unique key, so that sessions are individually protected. iMASH creates new keys for each application-requested session data channel to minimize potential data exposure through the compromise of a single key.

Some clients may find that even relatively inexpensive symmetric key encryption is too costly to impose on all data transfers. iMASH allows a user to specify that null

[4]Initial MWS selection can be arbitrary; a "poor" choice can easily be rectified by a subsequent middleware handoff.

[5]Middleware servers are presumed to be powerful enough that occasional costly authentication in the critical path is tolerable.

[6]WTLS supports DES, 3DES, RC5, and IDEA.

encryption is acceptable for session data channel communication for non-sensitive data. All device control channel and session control channel communication is required to use strong encryption, however, because these channels are used to create keys for new session/data channels.

The user/session level authentication takes place during session creation: a password-based protocol is used to authenticate a user to the MWS, prior to the initialization and execution of a client application.

iMASH security policies require that authorization be given in several places. First, the MWSs have policy regarding which sessions may handoff to which devices (e.g., a MWS may disallow moving a session containing patient records to a public workstation). Also target devices reserve the right to refuse sessions, so that users may allow only their own sessions to be transfered to their own PDA.

Secure Handoff Session handoff poses interesting novel security challenges. First, at device boot time, a device is authenticated to a particular MWS. The handoff target device is authenticated to some MWS, which may be different than the MWS of the source client. In the case of middleware handoff, the client device is not in general authenticated to the target MWS. In both CASH and FASH, client devices do not trust each other. In all cases, there is no time to do a computationally expensive handshake to establish trust with the new MWS or client device. This handshake avoidance is especially critical if one of the clients is a low-end device.

iMASH exploits the relationships that a client trusts the MWS to which it is connected, and the MWSs trust each other. The fact that all clients have a secure device control channel (DCC) is also critical. The transitive trust relationship allows a client to trust a second MWS in MASH or transfer its session to a second device in CASH or FASH. The existing secure DCCs enable new encryption keys to be created for all control and data channels on any handoff. This ensures greater security in the event that a device (or middleware) is compromised: once a session moves from the affected host, that host has no knowledge of the encryption keys used post-handoff.

4 Experiments

We have developed a prototype implementation of our middleware-based application session handoff architecture, and have iMASH-enabled a number of applications. This section outlines our experimental testbed and then describes several experiments using a mix of iMASH-enabled applications and middleware service implementations. The first two experiments use a socket-based iMASH architecture implementation with browser and remote shell applications; the third experiment employs an RPC-based iMASH prototype and a video player application. The experimental results presented below demonstrate that application session handoff is a viable approach to very low latency continuous computing.

4.1 Experimental Testbed

The purpose of the experiments reported here is to further establish the validity of the iMASH approach to application session handoff with a full-featured implementation under stress. To do so, we created a testbed that enables us to do controlled assessment of various aspects of iMASH. The testbed is designed to be reasonably representative functionally of a small scale iMASH environment. It contains a single application server, a single multiplexer (mux), two middleware servers, and four clients, as shown in the architecture diagram in Figure 2.[7] All of the experimental results presented here were produced on this testbed.

The physical testbed is largely constructed from initially homogeneous components that are selectively reconfigured (hardware and/or software) to produce desired degrees of heterogeneity. While an "ideal" testbed would have a variety of heterogeneous hardware and software pieces, there is also value in underlying homogeneity: for example, we were able to quickly trace an unexpected transient latency variance to a particular machine simply by exchanging roles with an identical machine. (We replaced the unreliable box, and re-ran the experiments.)

Our base machines are Dell Inspiron 4000 and 8000 laptops running Pentium III processors at 800MHz with 256KB cache, 128MB RAM, and standard issue 20GB disks. Network support is provided by PCMCIA-based 3COM 10/100Mbps "575" ethernet cards nominally running at 100Mbps through a pair of stacked 3COM OfficeConnect Dual Speed 8 switching hubs.

Our operating system environment is a Redhat Linux 2.2.17-8 kernel with most daemons enabled with default parameters (sendmail, ftpd disabled). Our client applications, autotester and all iMASH components are currently written in Java; the base JVM is Blackdown 1.3.1.[8]

In addition, the testbed includes a single Compaq iPAQ 3670 PDA running a 200MHz ARM processor with 64MB internal memory (32MB RAM, 32MB flash); a 240x320 pixel, 12-bit color display; and, a PCMCIA-based 10Mbps wireless WaveLAN IEEE 802.11b card. The iPAQ runs a Familiar Linux 2.4.7 kernel, and Blackdown Java 1.3.1.

In all of the results presented here, the application server, mux, and middleware servers execute on indi-

[7] as shown in iMASH design and current implementation support an arbitrary number of all components. Ongoing work is using simulation to extrapolate predicted large scale behavior.

[8] We compile our Java code with the "green threads" option, in large part to avoid known scheduling interaction deficiencies between this JVM and Linux 2.2 kernels.

vidual machines as described above. This is consistent with the target domain scenario of well-provisioned back-end infrastructure. Note that the mux component in the testbed is relatively weak and thus imposes much higher latency than would a "real" mux implemented on a programmable router. We also expect a MWS to be a powerful machine, as adaptation is often CPU intensive.

The testbed is set up with four client machines: three Dell laptops and one iPAQ PDA. One of the laptops is configured as a "powerful workstation", with a 100Mbps network card and "large" full color screen.[9] A second laptop plays the role of mid-range desktop, with a 1,000x1,000 pixel full color display and 100MBps network. A third laptop represents a truly mobile laptop, with the same parameters as the mid-range desktop excepting a 56Kbps network connection. The iPAQ is the fourth client, and has the screen size, color depth, and bandwidth constraints listed above.

The first two of the three experiments additionally share a common high-level testing regimen: we designed an *autotester* to drive an iMASH-enabled application with a specified synthetic workload, networking environment, and handoff profile.[10] The autotester software is designed to exercise specific aspects of iMASH in a randomized, yet repeatable, manner. The autotester is script-driven, and causes iMASH sessions to be created on specific client devices with specific applications and synthetic workloads, and externally triggers session handoff to specific clients.

4.2 Minibrowser Experiment

Each iMASH testbed component—legacy application server, mux, middleware server, client, or autotester—is hosted on its own machine. Additionally, we wrote a specialized "minibrowser" application which at startup obtains a pre-determined workload script of objects to request along with inter-request delays.

The workload consists of randomized sequence of requests for each of 200 unique image files, ranging in size from 0.5KB to 1.25MB. The sizes are roughly evenly distributed over the entire range, with some tail heaviness toward the smaller sizes. The inter-request delay is Poisson distributed with a mean of 2 seconds. Once started, the minibrowser executes the workload without further direct interaction with the autotester. Upon completion, the client exits, and the iMASH session terminates.

Concurrent with the minibrowser execution of the object request workload, the autotester uses a randomized yet repeatable script to trigger application session handoff (either CASH, MASH, or FASH) of the minibrowser session. In the work presented here, handoffs are injected into the application session execution with an inter-handoff request delay of 10 seconds.

The client application is a simple Java-based object viewer which can request and then display JPEG images. The savepoint state provided by the source client during handoff is small: a reference to the current object being displayed, and a small amount (under 100 bytes) of additional state. The session state actually transferred to the target client is typically much larger: the currently displayed object will be retrieved from a MWS cache during handoff, content adapted as appropriate for the target client, and then delivered to that client.

We conducted two minibrowser experiments. The first compares the performance of an iMASH-enabled environment with the non-iMASH version of that system. The second studies the performance of handoff across a range of clients, handoff types, and object sizes.

4.2.1 non-iMASH vs iMASH performance

The utility of an iMASH-enabled client is determined in part by the impact of the iMASH infrastructure on "normal" activity: the user-visible performance of conventional client-server interaction must be comparable in both non-iMASH and iMASH-enabled environments.

To assess the impact of iMASH on normal activity, we conducted experiments to measure the latency experienced by a client when requesting objects of varying size from the server (using HTTP *get* operations). We also varied the client device, to understand the impact of heterogeneity. One experiment employed an iMASH environment with application server, multiplexer, single middleware server, and single client application and device. The second experiment (the non-iMASH case) used the same application server; the client application is an iMASH-free version of the iMASH-enabled application. In both experiments, identical sets of randomized object requests were used, and identical client devices were used.

Figure 5 shows the results of this experiment. The

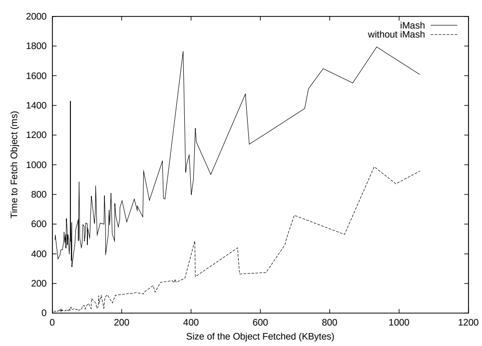
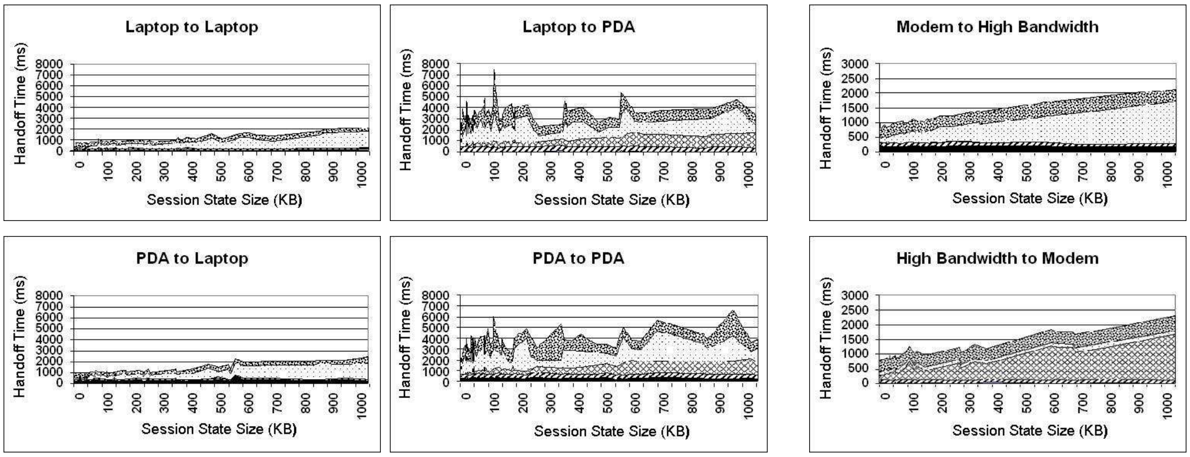


Figure 5: Average client latency experienced on object request (Y-axis), sorted by object size (X-axis), for iMASH and non-iMASH environments.

[9] "Large" means that no attempt will be made to content adapt an image to fit the screen

[10] This autotester is currently specific to the socket-based iMASH implementation, so it was not used for the video experiment.



(a) (b)

Figure 6: Client latency experienced on CASH (ms), as a function of session state size (KB). Note varying ranges on Y-axes. The upper curve represents the total handoff latency. Each band below the curve represents successive phases of handoff, from bottom to top.

- state transfer from source client to middleware server
- initializing a skeleton session on the target client
- adapting the session state before delivery to the target
- delivering the session state to the target client
- execution resumption on the target client

graph shows the average latency experienced by the application when requesting an object of a particular size. For brevity, we only show the "workstation" results. The graph shows that the latency increases linearly with the exception of a couple of outliers. The latency burden is about 0.5s for smaller objects, and approached 1s for larger object sizes. This baseline accounts for the additional costs of moving data up through a protocol stack on the middleware server, into a Java application, and back down through the stack to the client. The latency beyond the baseline 0.5s is largely due to the over-the-wire encryption incorporated by default by iMASH. We conclude that the additional latency of iMASH is tolerable even when no specific handoff benefits are exploited.

4.2.2 iMASH handoff performance

The purpose of this experiment is to determine the cost of application session handoff in terms of latency visible to the client application and therefore, the user. We also wish to understand the source(s) of such latency. In this experiment, a minibrowser application session was created and driven by an infinite loop over the 200 object randomized workload described above. Against this workload, a randomized series of handoffs was performed.

The graphs in Figure 6 summarize the results. They show the wall-clock latency (Y-axis) experienced by the application during handoff, with the session state primarily composed of the most-recently requested object, whose size is indicated. The data is separated by source and target client device type: Figure 6(a) shows data from handoffs involving a "full featured" laptop client and a wireless PDA client; Figure 6(b) shows data from handoffs between a wired 100Mbps laptop client and a 56K modem laptop client. Each data point is the alpha (= 0.10) mean of all occurrences of similar data.

The client device and network profiles for the PDA client and 56K modem client reflect the (greater) constraints faced in comparison with a typical wired workstation type of client. These constraints are sufficiently significant to cause the middleware server to invoke content adaptation in each case prior to delivering data to them; a small display size is the key profile constraint on the PDA, while slow network transmission is the prominent issue for the modem client.

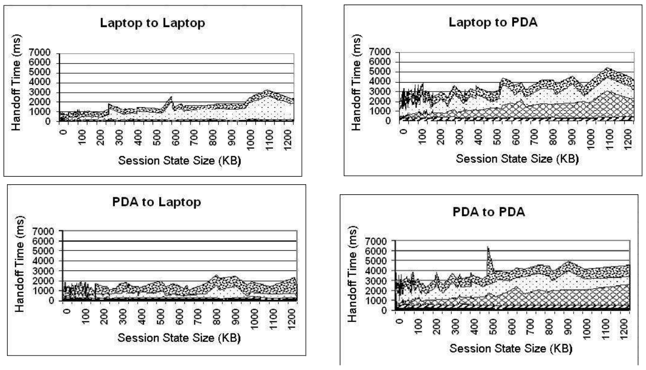
Perhaps the most important observation is that handoff incurs a latency ranging from 0.5 to 7 seconds, depending on state size and target device. At a more detailed level, we see in these results that when the target requires content adaptation, a latency proportional to the source session state size is incurred. The time taken to deliver the state to the target is proportional to the source state size when no target constraints are in place, but when adaptation is invoked, this component is constant relative to the specific constraint. We also see that the time to resume execution is essentially a constant function of the CPU type—because the PDA is executing the same code as the laptops, it has a longer resumption delay.

From the same experiment that yielded the CASH results, we also extracted MASH results. They show that for session state sizes under 400KB, a nearly constant

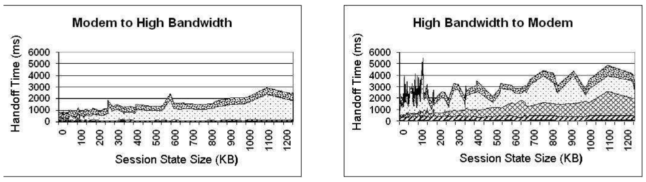
210ms of latency is incurred to transfer a session from one MWS to another. For larger session sizes, a simple linear regression yields a latency increase of about 13ms per 100KB increase in state size—which closely matches the expected transmission delay over the 100Mbps network links employed in this experiment.

In Figure 7 we show corresponding FASH results. Because the effort involved in a FASH is somewhat similar to performing both a CASH and MASH, we expect to see slightly larger delays here—and, in fact, we see here the same basic trends as in the CASH results above, noting that in most cases a slight increase in the latency is present. This increase is largely attributable to the delay in copying session state from one MWS to another.

We conclude that the delays imposed by CASH, MASH, and FASH are indeed acceptable: system-initiated handoffs (MASH) are well under a second, even for large session states, and thus are unlikely to be noticed by users; user-initiated handoffs are generally under two seconds.



(a)



(b)

Figure 7: Client latency experienced on FASH, as a function of session state size. The X-axes units are bytes, ranging from $\approx$ 1KB to $\approx$ 1.3MB; the Y-axis units are milliseconds. The upper curve represents total handoff latency.

4.3 Remote shell experiment

The Java Telnet Application, or JTA [JM] is a Java based remote login client. The JTA emulates a VT320 terminal and can be used to connect to either a telnet or an SSH server. It was selected for integration within the iMASH infrastructure primarily due to the usefulness of having such an application with handoff capabilities, in addition to the desire to test the iMASH architecture with an application possessing substantial state. The JTA must save approximately 100 individual pieces of state during handoff, which together often total 67KB. The largest single piece of state is the window buffer; it contains the 100 most recent lines displayed on the terminal and can grow in size to as much as 32KB. In addition to its significant amount of state, the JTA itself is a relatively large application, and has a source code base which exceeds 16,000 lines of code. Although this code already resides on the target, it must be loaded by the JVM at invocation.

We employed the same autotester as used in the mini-browser experiment, but with a workload appropriate to a remote shell application: executing various commands that generated varying amounts of "stdout" output, and randomly performing handoffs. Figure 8 shows a histogram of the latency experienced on each handoff. Each of the four types of handoffs were performed (randomly) 84 times; the frequencies are based on the aggregate total set of handoffs. The handoffs cluster as we expected, with the laptop-to-laptop values showing the lowest latency ($\mu = 1.2s, \sigma = 0.26s$); the pda-to-laptop values are next quickest, with $\mu = 3.9s, \sigma = 0.20s$; the laptop-to-pda values ($\mu = 9.0s, \sigma = 0.17s$) and pda-to-pda results ($\mu = 10.6s, \sigma = 0.17s$) show a distinctly longer latency.

This increased latency is dominated by the time it takes to load and begin execution of the JTA application on the PDA: while the startup cost on the laptop averages 0.48s, on the PDA the average is 6.8s. If the PDA's application startup cost was more like the laptop, the user-visible delay would be quite comparable. Recall that this application was not "ported" to the PDA; the code was simply loaded into its persistent storage and executed. We can reasonably conjecture that a version of JTA tuned for a PDA-class device would be much leaner and exhibit a strikingly lower startup delay, and we therefore conclude that heterogeneous application session handoff of complex applications with extensive state and a large code base is achievable while incurring modest delay.

4.4 Video player experiment

The previous experiments explored application session handoff in the context of discrete data. This experiment examines how well iMASH performs on streaming data, since latency is critical to the user experience of real-time streams. The overall system architecture (application server, middleware server, client) is standard iMASH. We developed a simple streaming server and matching display application based on Sun's Java Media Framework (JMF) [Sunb], and also incorporated JMF into the content adaptation pipeline (CAP) component of an RPC-based iMASH middleware server. JMF had the benefits of being Java-based—as is our MWS—and it has

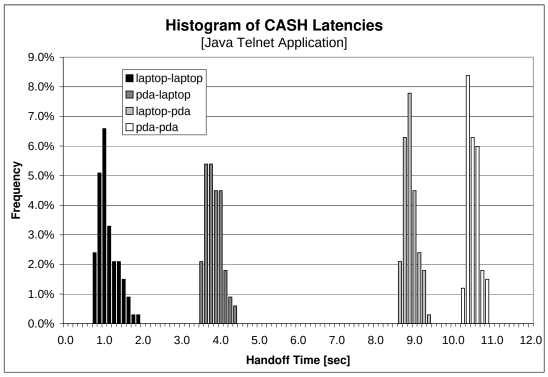


Figure 8: Frequency histogram of client latency experienced on client-only handoff (i.e., single MWS), for ≈ 340 handoffs of JTA application.

an extensive library of format transcoders suitable for a range of output stream bandwidths. It also uses the Real-Time Transport Protocol (RTP) [IET] as a delivery substrate. A downside is that JMF is not designed to switch transcoding methods on-the-fly, and so the CAP implementation must destroy and reconstruct a JMF instance to change the transcoding in mid-stream.

In our first streaming data experiment, we assessed the basic cost of switching transcoding methods, in the absence of handoff. In this experiment, an external

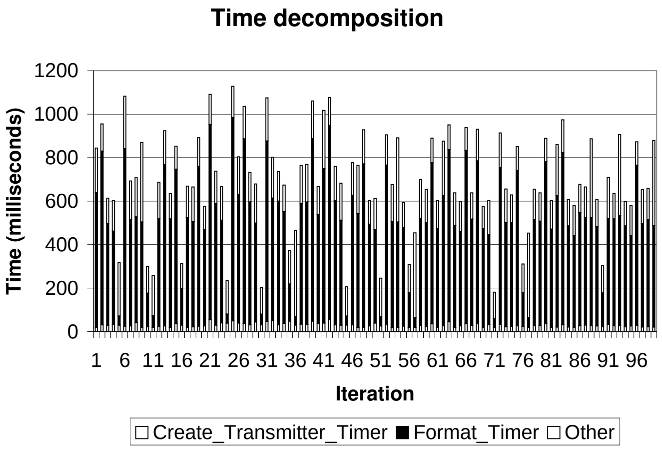


Figure 9: Latency to switch video transcoding method

thread notified the CAP that a significant change in available MWS-to-client bandwidth had occurred, and a new transcoding should be considered.[11] The CAP selects an appropriate transcoding method, kills the current JMF, and restarts a new one. The graph in Figure 9 shows the latencies experienced for a series of about 100 artificially-induced transcoding format changes. The alpha (=0.05) mean is 694ms to switch transcodings, but substantial unexpected variance clearly is visible in both the overall values (the entire vertical bar) and especially in the "Format Timer" component—typically 2/3 of the total.

Closer study revealed that this component delay is highly correlated with the transcoder selected: all of the five transcoders used in the experiment showed much less variance when grouped by transcoder type–but ranged from a mean of 50ms for the cheapest method to a mean of about 500ms for the most expensive. Sun's documentation for this portion of JMF (see [Suna]) indicates that this step may be very time consuming, as JMF may need to communicate with a server, read from a file, etc., in the course of obtaining its required resources. The choice of transcoder type (and perhaps implementation) is a dominant factor in our results, with "nicer" delays around 200ms readily achievable.

Our second streaming data experiment measured the cost to perform a CASH (client-only application session handoff) on the video player application used above. In this experiment, a user begins to play a video stream, and then randomly performs a series of handoffs back and forth between two clients. We measured the length of the interval between the time when the MWS received a message from the source client requesting handoff, and the time when the MWS began producing a stream for the target client. Note that with such applications, the latency experienced by the user is heavily dependent on size and content of the player's input buffer, which is frequently many seconds worth of data.[LGGB02].

The results of about 80 handoffs are shown in Figure 10. The mean delay is 1567ms, with a standard deviation of 587ms. Here again we see a noticeable variance, much of which is attributable to the cost to create a new transcoder object within JMF. Since the typical delay is well under two seconds, we conclude from this experiment that very fast heterogeneous application session handoff

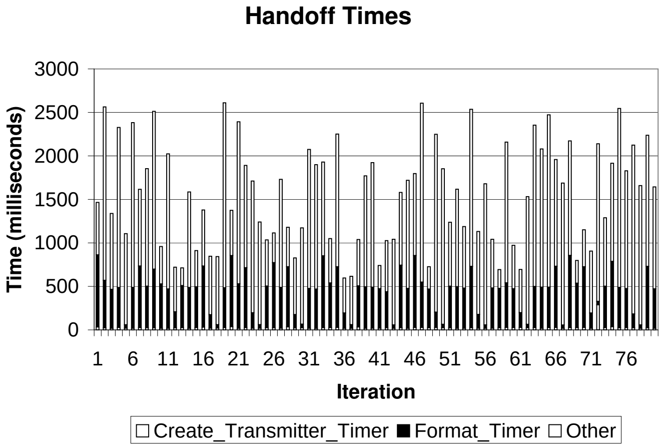


Figure 10: Streaming video handoff latency

[11]Effective ways to learn at the middleware level of bandwidth and other network QoS changes is the subject of ongoing research in our lab.

of a streaming data application is feasible using iMASH.

5 Conclusions and Future Work

Mobile continuous computing in the medical domain has been challenged to date by a lack of software support for the inherent heterogeneity found there. iMASH fills the gap in a significant way by exploiting *application session handoff*, a novel legacy server unaware, client application aware, secure approach to moving a computation across heterogeneous platforms with very low latency.

The architecture presented in this paper has been validated by experience with a number of prototype legacy applications and middleware server implementations. Further proof of the iMASH concept is found in experimental data obtained from implementations, which shows that the user-visible latency incurred by application session handoff is sufficiently low (typically, under two seconds) to allow deployment of iMASH in the real world—and enable the opening scenario to reach fruition.

The iMASH architecture supports streaming data, and our "streaming CAP" rapid prototype is currently being re-engineered for integration into our production middleware implementation. Construction of a detailed simulation model of iMASH is also underway. We expect the model to validate early design decisions which had foreseeable scalability implications, and also anticipate insight on unforeseen scalability opportunities.

Future work will look at several critical areas. It is important that a deeper understanding of interaction between layers be obtained, especially between wireless (re)transmission and content adaptation at higher layers. We also need to better understand (and perhaps loosen) the layer boundaries between mobile IP and ASH: cross-layer interaction may well be appropriate here.

The iMASH architecture, with multiple middleware servers and multiple client devices, appears ripe for enhanced reliability and robustness. Extensive caching at both MWS and client, coupled with mobility-inspired handoff, should be exploitable for automatic fail-over in response to either MWS or client failure.

Finally, client devices sometimes act as significant data sources (essentially servers), such as when participating in a video conference. The extent to which the iMASH architecture can be inverted is worthy of further study.

References

[BGW01] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: the insecurity of 802.11. In *Proceedings of the MOBICOM*, 2001.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the 7th ASPLOS*. ACM, October 1996.

[GDLB02] Robert Grimm, Janet Davis, Eric Lemar, and Brian Bershad. Migration for pervasive applications. Submitted for publication, 2002.

[Hel00] David Hellerstein. HIPAA and health information privacy rules: Almost there. *Health Mgt. Tech.*, April 2000.

[IET] IETF. RTP: Real time protocol, RFC3267. ftp://ftp.rfc-editor.org/in-notes/rfc3267.txt.

[JM] Matthias Jugel and Marcus Meissner. JTA, java telnet/SSH application. http://www.javassh.org.

[LGGB02] Jinsong Lin, Glenn Glazer, Richard Guy, and Rajive Bagrodia. Fast asynchronous streaming handoff. In *Proceedings of the IDMS/PROMS 2002*, 2002.

[LL02a] Wai Lum and Francis Lau. On balancing between transcoding overhead and spatial consumption in content adaptation. In *MOBICOM proceedings*, 2002.

[LL02b] Wai Yip Lum and Francis C.M. Lau. A context-aware decision engine for content adaptation. *IEEE Pervasive Computing*, pages 41–49, July 2002.

[MD00] D. Milojicic and F. Douglis, et al. Process migration survey. In *ACM Computing Surveys*, 2000.

[Nob97] Brian D. Noble, et al. Agile application-aware adaptation for mobility. In *Proceedings of the 16thSOSP*, pages 276–287. ACM, October 1997.

[PGB01] Thomas Phan, Richard Guy, and Rajive Bagrodia. A scalable, distributed middleware service architecture to support mobile internet applications. In *Proceedings of the IEEE Workshop on Wireless Mobile Internet*, 2001.

[PZB02] Thomas Phan, George Zorpas, and Rajive Bagrodia. An extensible and scalable content adaptation pipeline architecture to support heterogeneous clients. In *Proceedings of the 22nd ICDCS*, 2002.

[RKC01] Manuel Román, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *Distributed Systems Online*, 2(5), 2001. http://dsonline.computer.org.

[SB00] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *ACM/IEEE Int'l. Conf. on Mobile Computing and Networking*, August 2000.

[SKP02] Erik Skow, Jiejun Kong, and Thomas Phan, et al. A security architecture for application session handoff. In *Proceedings of the ICC*, 2002.

[Suna] Sun Microsystems. javax.media interface controller. http://java.sun.com/products/java-media/jmf/2.1.1/apidocs/javax/media/Controller.html.

[Sunb] Sun Microsystems. JMF: Java media framework. http://java.sun.com/products/java-media/jmf/.

[vMCT95] Erik van Mulligen, Ronald Cornet, and Teun Timmers. Problems with integrating legacy systems. In *Annual Symposium on Computer Applications in Medical Care : Toward Cost-Effective Clinical Computing*, 1995.

[WAP] WAP Forum. Wireless transport layer security specification. www1.wapforum.org/tech/ documents/WAP-261-WTLS-20010406-a.pdf.

[ZM02] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *ACM/IEEE Int'l. Conf. on Mobile Computing and Networking*, 2002.