

UNIVERSITY OF CALIFORNIA

Los Angeles

**Enabling Mobility With Application Session
Handoff**

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Erik Skow

2003

© Copyright by
Erik Skow
2003

The thesis of Erik Skow is approved.

Richard Guy

Songwu Lu

D. Stott Parker

Rajive Bagrodia, Committee Chair

University of California, Los Angeles

2003

TABLE OF CONTENTS

1	Introduction	1
2	Architecture	6
2.1	Middleware Architecture Overview	6
2.2	Session Control Thread	8
2.2.1	Architecture	9
2.2.2	Implementation	9
2.2.3	Future Work	12
2.3	Server Proxy	13
2.3.1	Architecture	13
2.3.2	Implementation	14
2.3.3	Future Work	17
2.4	Mux Manager	18
2.4.1	Architecture	18
2.4.2	Implementation	18
2.4.3	Future Work	19
2.5	Client Proxy	20
2.5.1	Architecture	21
2.5.2	Implementation	21
2.5.3	Future Work	23
2.6	Message Handler	23

2.6.1	Architecture	24
2.6.2	Implementation	25
2.6.3	Future Work	25
2.7	Middleware Communication Module	26
2.7.1	Architecture	26
2.7.2	Implementation	26
2.7.3	Future Work	30
2.8	Session Database	31
2.8.1	Architecture	31
2.8.2	Implementation	32
2.8.3	Future Work	32
2.9	Security	33
2.9.1	Access control	35
2.9.2	Future Work	35
2.10	Protocol Handler and the Content Adaptation Pipeline	36
2.11	Mux	38
2.11.1	Architecture	38
2.11.2	Implementation	39
2.12	iMASH Client Layer	40
2.12.1	Device Layer	41
2.12.2	Session Layer	42
2.12.3	GUI	43

3	Protocols	45
3.1	Device connects to MW	45
3.2	Session Creation	48
3.3	Data Channel Creation	49
3.4	Session Handoff – CASH or FASH	51
3.5	Middleware Handoff	57
3.6	Suspend	59
3.7	Pull (Restore)	61
3.8	Get Session Object	62
3.9	Reconnect Data Channel	64
4	iMASH Applications	66
4.1	General Design Constraints	66
4.2	Networking behavior	67
4.3	Asynchronous Signaling and Handoff Semantics	67
4.4	Save pointing and Restoring State	68
4.4.1	Application Objects	69
4.4.2	Reference Objects	69
4.4.3	G-Functions	70
5	Results	72
5.1	Applications	72
5.2	Extensibility	73
5.3	Performance	74

5.3.1	Data retrieval overhead	74
5.3.2	Handoff performance	75
6	Conclusions	79
A	iMASH messages	81
B	Acronyms	97
	References	99

LIST OF FIGURES

1.1	iMASH architecture	4
2.1	Middleware Architecture Overview	6
2.2	Server Proxy Architecture	14
2.3	Datagram Stream Converter	16
2.4	Client Proxy Architecture	21
2.5	Middleware Communication Module Architecture	27
2.6	MWCM Connectivity Graph	28
2.7	Session Database Architecture	32
2.8	Mux Architecture	39
2.9	iMASH Client Layer Architecture	41
3.1	Protocol Figure Key	46
3.2	Session Creation Protocol	47
3.3	Data Channel Creation Protocol	50
3.4	Client Handoff Begin Protocol	52
3.5	CASH Protocol	52
3.6	FASH Protocol	53
3.7	Client Handoff Finish Protocol	54
3.8	MASH Protocol part 1	58
3.9	MASH Protocol part 2	59
3.10	Suspend Protocol	60

3.11 Pull (Restore) Protocol	61
3.12 Get Session Object Protocol	63
3.13 Reconnect Data Channel Protocol	64
5.1 Average client latency experienced on object request (Y-axis), sorted by object size (X-axis), for iMASH and non-iMASH environments.	74
5.2 Client latency experienced on CASH 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.	77
5.3 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.	78

LIST OF TABLES

ABSTRACT OF THE THESIS

**Enabling Mobility With Application Session
Handoff**

by

Erik Skow

Master of Science in Computer Science

University of California, Los Angeles, 2003

Professor Rajive Bagrodia, Chair

This thesis describes the second generation of the iMASH architecture, a solution to the problem of user mobility within pervasive computing. The architecture enables Application Session Handoff (ASH) a term that has been coined to describe the movement of a running application from one device to another. This is not to be confused with process migration, ASH is a more lightweight solution that only migrates critical application state between devices rather than a running process stack. iMASH further separates itself by enabling handoff between disjoint devices running perhaps different operating systems.

The iMASH architecture is designed around an scalable middleware service that enables the required mobility and addresses the newfound problems associated with applications running in a dynamic environment. This thesis describes the problems and solutions encountered in building the iMASH middleware service as well as describing the details of how the system works.

CHAPTER 1

Introduction

The iMASH project aims to address the problems associated with ubiquitous computing. The emergence of a wide range of mobile wireless devices such as laptops, PDAs, cell phones, and smart watches has enabled a new form of computing. Computing is available to users all the time and in any location. Also, with the lowering cost of such devices, it is not uncommon for people to have many different computing devices available to them. In their homes, they may have traditional desktop computers, web pads, or less traditional computing power embedded within other types of devices such as kitchen appliances. On the road, a whole new set of devices might be available to a person either in their car, or that they carry with them. Finally, there are likely to be many more devices that can be used when a person gets to work.

In the traditional computing model, applications are run on individual devices, such as a person using a word processor email client or web browser. People tend to use distinct computers for different tasks. Web browsing may be done at home or at work with a laptop computer; games definitely on a desktop with a good display; personal information management and scheduling on the PDA. The application “lives” on a specific device and a user generally chooses a device that has good characteristics for the given application. Games need a good display, so they are on a desktop, while appointment minders need good mobility, so are run on a PDA.

iMASH attempts to alter the model of one device per application. It liberates an application to give it the ability to move and adapt from one device to another. In the iMASH system, an application follows the user and runs on whatever device is available to it. When a person is at home, an application can be running on their desktop, but when they leave, the application moves with them, moving onto their PDA or cell phone. When they get to work, it can again migrate to their office desktop.

This environmental paradigm poses many problems. First applications must be able to move from device to device, which may not share the same type of hardware. The processing core may change, as well as IO devices such as screens for output and keyboard, mouse, and voice for input. Memory availability and network bandwidth also change when devices change.

iMASH addresses the mobility challenge through application session handoff (ASH) [PGG01], with its variants: Client only ASH (CASH), Middleware only ASH (MASH) and Full handoff (FASH) [LGG02]. The heterogeneity is handled by utilizing Java for a virtual machine and having a middleware capable of data transcoding [PZB02b].

This thesis describes the second generation iMASH middleware architecture built to enable iMASH services. It has roots in two different initial independent architectures. The first was an RPC-based model with a relatively tightly coupled mobile application and static middleware. Static objects such as images and web pages could be fetched for a client through RPC via the middleware service. The middleware could then transcode the data to take into account client capabilities and network conditions. The second architecture was built to handle streaming media. The client devices and the middleware were loosely coupled and the middleware acted much like a proxy. A client application could request a streaming

video or audio clip from an application server that was proxied through the middleware. Handoff was done through socket communication that could start and stop applications on the client devices. The application would be started on the new device and the middleware would redirect the streaming data to the new device. This second streaming architecture could also handle multiple middleware and all the various types of handoff.

The architecture that arose from these two distinct sources incorporates the ideas from both. It handles both streaming and static data, scalable middleware, object transcoding and the many forms of handoff. It also incorporates a new security model that allows for data security, authentication, and authorization in a mobile environment.

The result of this architecture merge has provided a system that not only has increased functionality, but also good performance. As shown in [BBC03], most iMASH handoff operations take place in only a few seconds, which is well within a users tolerance.

The iMASH architecture is made up of several components as shown in Figure 1.1. A group of middleware servers (MWS) and multiplexers (MUXes) make up the middleware service. The middleware servers provide the main architectural component in the iMASH system. These servers provide several services to the iMASH clients including data proxying, content transcoding, security, and application session handoff (ASH). The middleware service interacts with a thin layer on each client device known as the iMASH client layer (ICL) to enable application check-pointing and application restoring following an application session handoff.

Various aspects of the iMASH architecture have been discussed in several papers by the iMASH team[LGG02, PXG01, PGG01, SKP02, PZB02b, PZB02a, Pha02, BPG02, BBC03]. This thesis describes the second generation iMASH

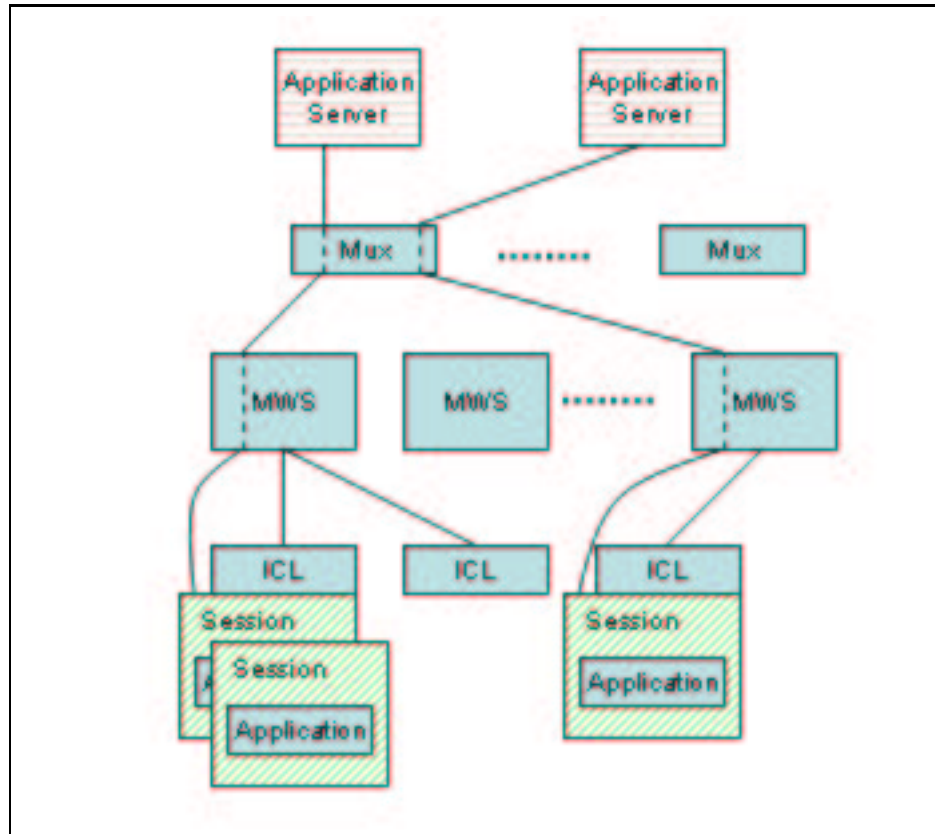


Figure 1.1: iMASH architecture

architecture in detail as well as provide insight into its strengths and weaknesses.

This document is organized as follows: Chapter 2 describes the system architecture, including the middleware servers, the muxes and the client layer. Chapter 3 discusses the various protocols used for both authentication and handoff. Chapter 4 gives details on designing applications that will work well within the iMASH architecture. Chapter 5 has experimental results and chapter 6 has conclusions. Appendix A and B provide reference on acronyms and detailed information on messages sent within the iMASH protocols.

CHAPTER 2

Architecture

This chapter documents the iMASH architectural components. It begins with the design of a middleware server and then continues to describe both the multiplexer and client layer.

2.1 Middleware Architecture Overview

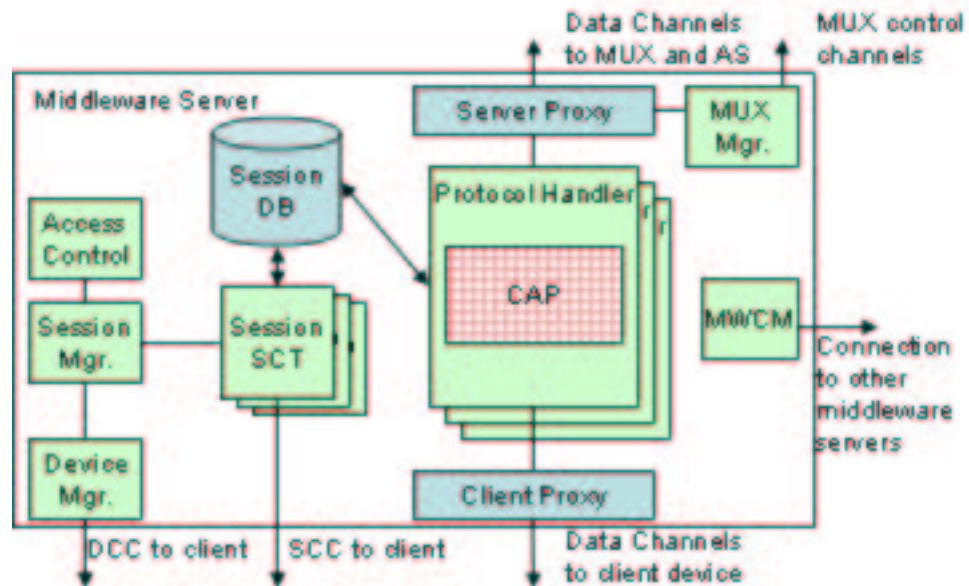


Figure 2.1: Middleware Architecture Overview

The iMASH middleware server is the key facilitator to the iMASH service.

As shown in Figure 2.1, the middleware server (MWS) is made up of several components. There is a session manager which manages client sessions. A client session consists of all the data and meta-data associated with running an application through the iMASH service. Multiple sessions may exist on a middleware server at a time. Sessions are also the unit of mobility in the iMASH service. A session may start out on one middleware server and client device and then migrate through handoff to another middleware and client pair.

Each session has a session control thread (SCT) that directs it. The session control thread takes commands from the client device and other middleware servers and performs iMASH tasks. The SCT performs access control via the access control module and interact with devices connected to the middleware through the device manager. However, the main task of the SCT is managing session data channels.

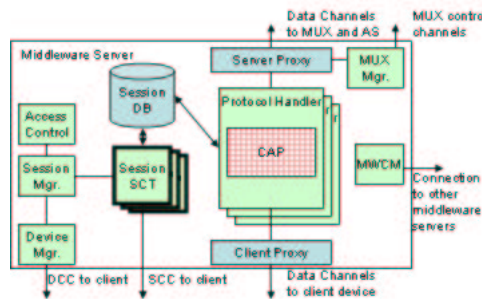
When an application running on a client device requests a new data channel to an application server, it is translated into a request to the SCT. The SCT then sets up a middleware proxied data channel using the client proxy (CP) and server proxy (SP) components. Once the connection is set up, the SCT initiates a protocol handler (PH) instance to manage the connection. The protocol handler's job is to parse the data stream and modify it if necessary. This modification is done when either network, mobility, or device characteristics prevent data from being consumed as it is being sent. In this case, the protocol handler will parse out data objects or data stream to be transcoded by the content adaptation pipeline (CAP).

The remaining components of the middleware server are the session database and the middleware communication module (MWCM). The session database stores session information used by the middleware and client state during hand-

offs. It also acts as an object store and cache for the protocol handler. The MWCM serves as a means to communicate with other middleware servers. It is used to query information and to initiate and complete middleware handoffs.

The following sections describe each of the middleware components in detail for both architecture and implementation. Each section also ends with a portion devoted to future work which describes possible directions to improve the design and implementation

2.2 Session Control Thread



The Session Control Thread (SCT) as the name implies is the main thread for a running session. It has the responsibility to manage the session control channel (SCC) and handle session events. It interacts with virtually every other object in the middleware. It may create and destroy data channels, initiate and complete handoffs, and respond to client events. The SCT is created by the session manager whenever a client initiates a session with the middleware. A SCT is also re-created on a new middleware in response to a middleware only or full handoff. Once created, the SCT runs autonomously and moves with the session. On handoff, all the session specific data is moved from one middleware to the other and the SCT is resumed there.

2.2.1 Architecture

The SCT is designed as a single control thread with a interface for other threads to make local calls. These calls are mainly signals to inform the SCT to do the various tasks of handoff. The SCT manages all of the objects associated with a session and keeps track of state as the session moves. To facilitate this state transfer, much of the state is kept in the session database. Thus, to move a session, only one session database record needs to be transferred to the new middleware. The SCT also has the responsibility of creating and linking objects to form data channels. Once a data channel is running, it is fairly autonomous. To execute handoffs, these data channels are suspended, resumed, stopped, and restarted as necessary by the SCT.

The main thread's job is to listen for commands from the client via the iMASH Client Interface (ICI). It listens for the following messages: *login*, *start_new_session*, *reconnect*, *data_channel_request*, *query_available_devices*, *push_to*, *handoff_to_me*, *discover_MW*, *change_MW*, *suspend*, and *get_session_object*. These commands are used in the various stages of authentication, creation of data channels, and hand-off. Additional information regarding these messages can be found in section A. The protocols that are used by the SCT and other components are described in Chapter 3.

When the SCT receives one of these messages, it interfaces with all the other middleware components to carry out the tasks required.

2.2.2 Implementation

The SCT is organized as a event loop that simply receives messages and then carries out tasks. The implementation details of each message is described in the

following paragraphs.

A *login* message is the first message received for a valid session. This message causes the middleware to check with the access control module to determine if the user is a valid iMASH user. If the check passes, the user is noted as authenticated and can access the other session services. In the case of failure, the SCT will send back an error over the SCC and let the user try and login again.

After a user has authenticated to the system, the next message sent to the middleware is a *Start New Session* message. This identifies this session as a new session (as opposed to resuming an old session, which also requires authentication). When this message is received, the SCT registers the session with the session database and creates a unique session ID for the session.

The *data channel request* message is used to request a new session data channel. After checking to make sure that a user is authenticated, the SCT utilizes the services provided by the client proxy and server proxy to connect the data channel through to the application server and client device. Once the channels are connected, the SCT starts up the protocol handler to handle possible transcoding on the channel. The data channel is also registered in the session database so it will move with the session on middleware handoff.

Related to the *data channel request* message is the *reconnect* message, which reconnects a data channel after handoff. In this case, only the client end receives a new connection. On the application server side, the old connection is reused. The process then continues like a data channel request with the two sides being connected by the protocol handler.

The *query available devices* message is a discovery message used to find all available targets for a handoff. In response to this message, the SCT checks what devices are connected to the current middleware. It then uses the middleware

communication module to forward the request to all other middleware. Once all the responses have come back, the SCT sends the concatenated list to the client, which can then choose a target device.

A *push to* message indicates that a handoff should take place. This is perhaps the most complicated task that the middleware must process. It begins by determining the target middleware, that is, the middleware that the target device is directly connected to. The SCT then halts all the data flows through the middleware in preparation for handoff. This is followed by the client device cleaning up the application and sending a *client session* message to the middleware. At this point, the client session is saved in the session database and handoff begins. Local handoff (CASH) is handled through the device manager and remote handoff (FASH) is handled through the middleware communication module. Details of the CASH and FASH protocol can be found in the protocols section.

A *handoff to me* message is used to complete a handoff for the middleware. In this message, the client device provides the session ID created when the session started. The SCT responds to this message by associating all the saved data in the session database with the new session. It also sends the check-pointed client session to the new device so it may restart the application.

The *change middleware* message is used to signal a middleware handoff. The middleware response is similar to a normal handoff. In fact, the middleware simply performs the first part of a full handoff, transferring all middleware state to the new middleware. However, the protocol ends differently, with the second middleware simply reconnecting back to the original device.

Discover middleware messages are used by the client device to locate new middleware. In the current implementation, this is only done prior to a middleware handoff. The SCT simply checks with the middleware communication module to

identify all the of the middleware it has a connection to. The SCT then sends a *ack discover middleware* message back to the client with the middleware list.

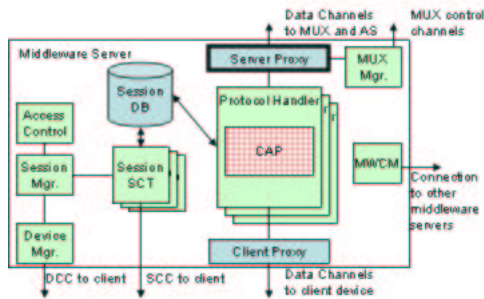
A *suspend* initiates a session suspension by the SCT. This is essentially just the first part of handoff. The client follows this message with a *client session* message. The SCT simply suspends the data flows and stores the client session in the session database. This session can be later resumed with an *handoff to me* message containing the session ID.

The *get session object* message is used extensively in handoff. After a session has successfully been moved from one device to another, the target device must request session data. This is done through the *get session object* message. Upon receiving this message, the SCT attempts to retrieve the object from the object store in the session database. If the object is found, the object is then sent to the CAP for transcoding to the new device characteristics. Once transcoded, the middleware sends a *ack get session object* message containing the object. In the case that the object is not found, an error is sent back to the client.

2.2.3 Future Work

The Session Control Thread is a fairly straight forward event loop, however there are places where it could be improved. The main deficiency is robustness with the client. The SCT allows for a session to be suspended and resumed through their respective protocols, however it does not handle unexpected client disconnects. Most of the tools already exist, but instead of doing something smart, the session dies if not suspended cleanly. Also, suspended sessions remain resident in the middleware forever. Any session that is suspended for too long should be terminated. This addition would also involve the session database.

2.3 Server Proxy



The server proxy (SP) component of the middleware architecture is responsible for managing data channel connections up through the mux to the application server. It has the ability to create both TCP and UDP channels to the mux and controls connections from the mux such that the data channels are connected through to the application server. The SP also has the ability control the various data channels. Using this functionality, the session control thread (SCT) is able to start, stop, resume flows. It is also able to create new data channels to the mux and transfer existing channels to other middleware. These abilities are used in the various handoff scenarios. Server proxy also provides a common interface to the protocol handler. Regardless of the network transport protocol, TCP or UDP, the server proxy creates streams that are read and written by protocol handler. This eases the load on protocol handler such that there need not be a separate instance for TCP and UDP flows.

2.3.1 Architecture

The Server Proxy is comprised of a channel interface, channel hash table, generic abstract data channel, and both specific stream and datagram data channels. As mentioned previously, the SP interface has the ability to start, stop and resume

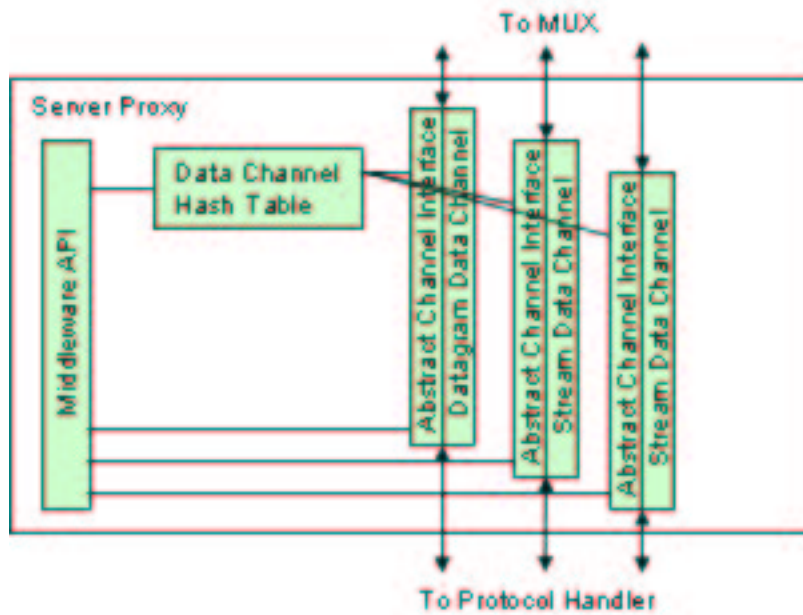


Figure 2.2: Server Proxy Architecture

data channels. Additionally the SP allows the Session Control Thread to insert End Of Stream (EOS) markers into the stream, push back data onto the stream, as well as ignore socket exceptions. These functionalities are used in the process of the various types of handoff.

2.3.2 Implementation

The abstract server proxy data channel provides a common interface to both the TCP and UDP data channels. This is important so that other components, specifically the protocol handler, does not need to have special interface code to handle the two different types of sockets. The generic case is actually a non-blocking stream data channel. Thus, the protocol handler only needs to read and write streams to do socket IO. Server Proxy hides many of the handoff details as well as datagram packetization.

Stream data channels are the simple case for server proxy. The data channel is a lightweight wrapper around an iMASH socket that provides the extra stream management functions. The main addition of the stream data channel wrapper for the iMASH socket is to make the socket input streams non-blocking. At the time of implementation, server proxy was designed for Java 1.3.1, which did not include non-blocking socket IO.

Non-blocking input streams are implemented by introducing a wrapper around the input streams before they are provided to calling objects. The stream data channel returns the non-blocking input stream to the protocol handler when it requests a data channel input stream from the server proxy. The non-blocking nature is necessary to prevent deadlock on handoff. This deadlock occurs because socket replacement, that occurs at handoff needs to occur between socket read calls. With blocking sockets, the Server Proxy could not alter the sockets without causing a socket error to the Protocol Handler, which in turn would close the connection.

The non-blocking input stream is implemented as an object with an internal thread and a circular buffer. An internal reader thread does blocking reads on the socket and external reads go to the buffer. This solution works from a mechanical perspective, but it is costly. Every stream data channel has an extra thread allocated to avoid the blocking nature of Java sockets. However, one of the many features added to Java 1.4 is non-blocking IO. By using these sockets instead, substantial processing and complexity overhead can be removed from every stream data channel socket read. Datagram data channels are a more complicated case, because they provide extra functionality to create streams out of incoming packets and change outgoing writes into packets to be sent over the network.

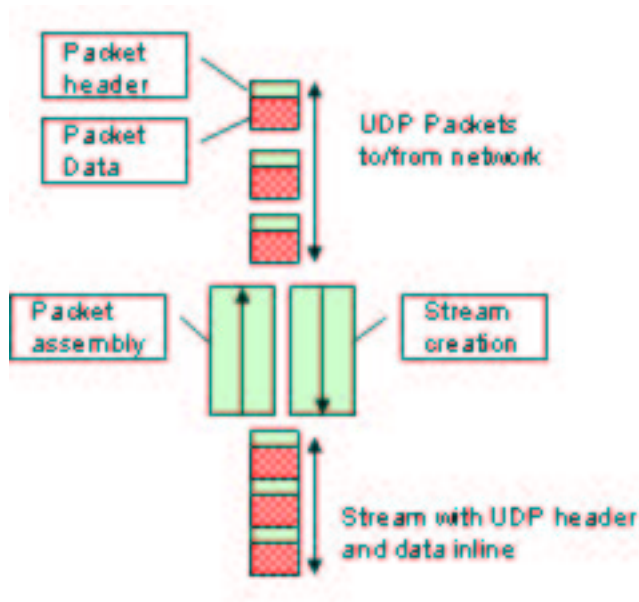


Figure 2.3: Datagram Stream Converter

Datagram data channels contain several sub-components designed to fit them into the same interface provided for stream data channels. Datagram data channels contain iMASH datagram sockets internally and a datagram stream converter. The datagram stream converter is shown in Figure 2.3.

Unfortunately, the datagram stream converter is more inefficient than the stream non-blocking IO fix. The converter contains two internal threads, one to assemble datagrams into a stream and the other to create datagrams from a stream. This handles both upstream and downstream data. Creating a stream from a UDP packet is done by in-lining the whole UDP packet. That is, each field of the datagram packet is written to the stream, not just the data. This decision has several ramifications. First, for the protocol handler (PH) to modify data in a UDP stream, it needs to parse UDP packets. Second, if the PH modifies the data in any way, it needs to reassemble the data into valid UDP

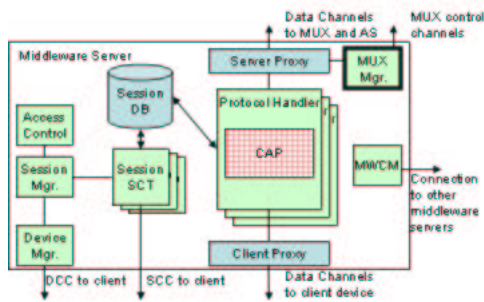
packets.

The additional header information was added into the stream for several reasons. First, it is helpful to the corresponding datagram stream converter in the client proxy to recreate the UDP packets. Second, the protocol handler may need the extra information in the UDP packets to decide what to do with the data. A simple example of this is using the sequence numbers. Before transcoding the data, the protocol handler may need to reorder or request retransmission of critical packets to transcoding.

The second half of the stream converter creates datagrams out of streams. This thread reads the stream, picking off the UDP fields. It then uses this information to create a packet and send it over the network to the desired location.

2.3.3 Future Work

Far more work went into the UDP implementation than the TCP component of the server proxy, but it still needs the most work. Besides being a bit of a non-intuitive interface, the UDP implementation is quite fragile. If any mistakes are made by other components on the streamified UDP packets, such as putting in malformed headers, the server proxy will kill the data channel. Also, UDP needs to have more options for how the data is assembled into a stream. The option should exist to pass data through with or without UDP headers. Protocols that run on top of UDP may be able to handle lost packets or data, thus it may actually complicate the protocol handler by having to remove UDP headers from the data before transcoding when it is not necessary.



2.4 Mux Manager

Mux Manager (MM) is a long lived component of the middleware that facilitates mux discovery and mux communication. The MM is instantiated by the MWS at boot time and interacts with instances of Server Proxy to provide connections through a mux to application servers.

2.4.1 Architecture

Mux Manager has three logical components: a discovery service, mapping service, and mux control channel service. The discovery service finds available muxes, the mapping service maps application servers to muxes, and mux control channel service maintains mux control channels (MCCs) to all the muxes and provides an interface for other components to communicate to the muxes via the MCCs.

2.4.2 Implementation

The Mux Manager discovery service is currently implemented in a simple manner. The administrator provides a listing of valid muxes in a configuration file (mws.conf). The discovery service uses this list to create control channels to the muxes. If a session is transferred to a middleware in a FASH or MASH that is connected through a unknown mux, the mux manager learns about the new mux

and begins initiating connections through it. Thus, the middleware are able to learn about muxes in the system through gossip.

The mapping service is designed to provide a mapping of muxes to application servers. Currently, all muxes and application servers are in one group and the MM chooses a mux randomly. However, the extension to this service would be to allow a middleware administrator to delegate certain application servers to specific muxes. An example of this would be an administrator choosing to send all web traffic through a set of muxes. The mapping service could realize that the middleware is requesting a connection to port 80 and send the request to an appropriate mux.

The control channel service allows the middleware to communicate with a mux that it has found through the handoff or the mapping service. It provides a message handler that can be used to send and receive commands from the given mux. If the control channel service encounters an unknown mux (this could happen from a handed off session), the control channel service attempts to connect to the new mux. If it is successful, it will return the message handler and notify the discovery service of the new mux.

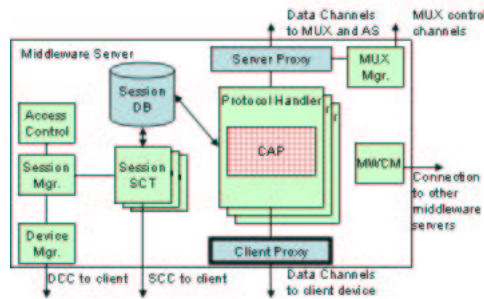
The only component that interacts with the mux manager is the server proxy. When connections are requested from server proxy, it in turn requests an available mux and communicates with that mux via mux manager services.

2.4.3 Future Work

As discussed earlier, the Mux Manager mapping service simply chooses a random Mux to handle the new data channel. Having some administrative choices in this domain would be desirable to channel different types of connections through different muxes. Also, the Mux Manager does not take into account what the

load is on a given Mux. It might be better to ask all the available muxes what their load was and assign the new data channel to the least burdened one.

2.5 Client Proxy



The client proxy (CP) component of the middleware is responsible for managing data channels to client devices and handle the device handoffs as a session moves. It can handle both TCP and UDP data channels and is controlled by the session control thread SCT. Through control input from the SCT, the client proxy can start, stop, and resume data flows. Through this interface, the SCT is able to stop data flows to one device, handoff the session to a new device, and then allow the target device to resume the data flows. This is the standard procedure for client only handoff (CASH) or full handoff (FASH). Like server proxy, client proxy also interacts with the protocol handler to serve content adapted data to the clients, depending on device characteristics. The job of client proxy is to manage the data channels for each session and to hide the details of handoff to other components, such as protocol handler.

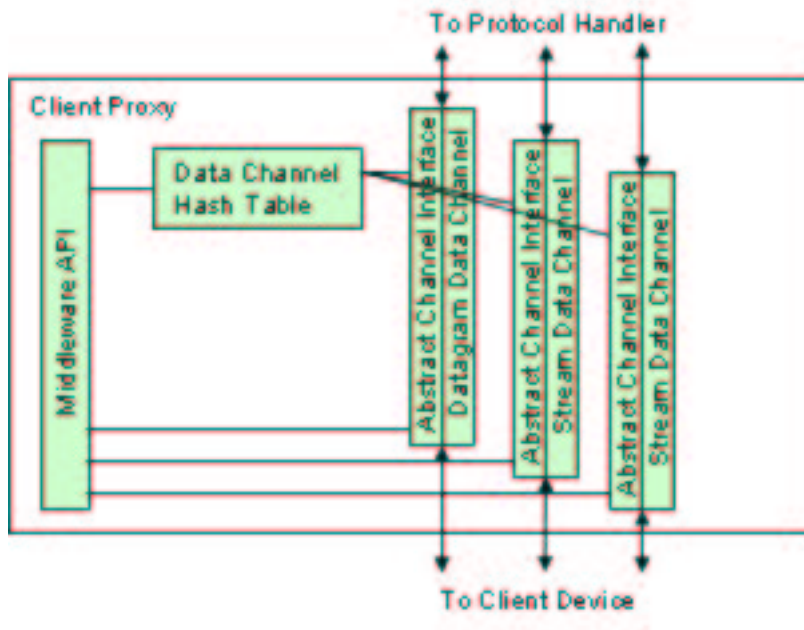


Figure 2.4: Client Proxy Architecture

2.5.1 Architecture

Client proxy has a similar structure to server proxy as shown in Figure 2.4. It is composed of a channel interface, a channel hash table, and the data channels. CP allows the session control thread to start, stop, and resume data channels. It also handles the act of connecting back to client devices after they have requested a data channel.

2.5.2 Implementation

Similarly to server proxy, client proxy contains an abstract data channel object that provides a common interface to both the TCP and UDP data channels. This is important so other components, specifically the protocol handler does not need to have special interface code to handle the two different types of sockets. The

generic case is actually a non-blocking stream data channel. This abstracts the differences between TCP and UDP data channels. Both are treated as though they were stream (TCP) oriented data channels to other components. Like server proxy, the individual datagram packets are formed into a datagram stream to be fed through protocol handler, which can then read from streams. On the way out, the client proxy expects a datagram packet stream, that it then breaks into packets and sends out to the client.

New data channels are created by a call to `createDataChannel` for TCP channels and `createDatagramDataChannel` for the UDP case. Client proxy first checks if the channel is already in the hash map, and if not, begins creating the data channel.

TCP data channels are created through a connect back protocol. The client originally requests the data channel over the session control channel SCC. Along with other information, the client provides a port for the middleware to connect back to. The client proxy, upon being called to create a connection, connects back to the client device on this port. Once the client proxy has connected the port, the session control thread (SCT), finishes off the data channel connection and data begins to flow from the application server to the client.

UDP data channels have a little more complexity. In addition to being created in the same connect back style¹, the UDP data channels have to spawn an additional thread to create a stream of UDP packets on the upstream and send packets out from a stream in the client direction. This “streamifying” of UDP packets is done to provide a common interface to the protocol handler.

Handoff is done in a similar manner to initial connection. On the start of a handoff, the SCT suspends all data channels for a given session. The session

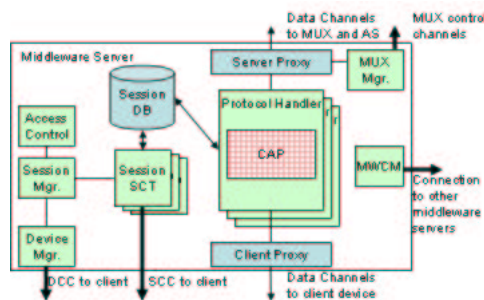
¹this is not strictly true, because UDP does not actually connect, just port numbers for the channel are created and sent back to the client

and application are then transferred to the new device. Once the application is running on the target, the target device issues a resume call to the SCT, which in turn calls the redirect data channel function in the client proxy. This causes client proxy to connect back to the new device, similarly to channel creation. After connecting back to the new client, client proxy utilizes the functionality provided by iMASH handoff sockets to handoff the data channel to a new socket, thus connecting the data channel streams to the new socket. Once the handoff is complete, the SCT calls the resume function, which once again allows data to flow through the client proxy, directing the data to the new device.

2.5.3 Future Work

Client Proxy exhibits the same deficiencies as Server Proxy in regards to UDP data channels. The handling of UDP data needs to be improved as well as allowing more options in regard to how UDP data channels are set up.

2.6 Message Handler



Message Handler (MH) is a lightweight component used to send and receive messages over Sockets. It is used for control communication between all iMASH components, clients, middleware, and muxes. It's main function is to allow several

different components to share one TCP channel without one component receiving messages destined for another.

2.6.1 Architecture

The Message Handler is broken up into three parts, a message queue, a receiver thread, and an API. The message queue stores all messages received, but not yet delivered. The receiver thread populates the message queue by listening to a socket input stream and the API allows higher level components to send and receive messages using the message handler. The message handler hides the details of delivering messages to multiple threads using the pipe asynchronously.

The API allows components to send messages with or without a message ID. On the receiving side, it allows a component to wait for a message with a given ID, type, or both. If messages are sent without an ID, the message handler will assign a globally unique ID to the message. The idea with this interface is that a sender and receiver on two different nodes can send and receive messages in the following way: Node A sends a message that gets assigned a globally unique ID. Node B listens for any message with a given type. It then responds by sending a message back of any type, but with the same ID as the first message. Thus, initial messages and acknowledgments have the same ID. In this way a general server can accept messages from a variety of different client threads sending the same types of messages. The clients will be able to distinguish the replies because of the ID. This also allows the clients to receive any message in return with the correct ID. This is useful in error conditions, as the server is able to send back an error message instead of the expected response without fouling up the message handler.

2.6.2 Implementation

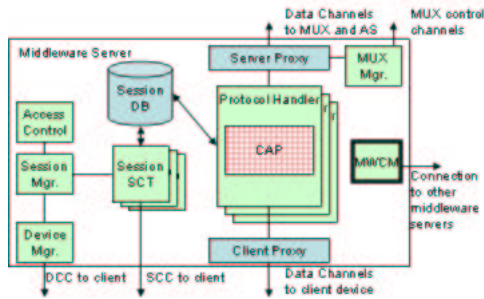
As described in the architecture, the message handler has three logical components, a message queue, a receiver thread, and an API. The message queue is simply an array containing the messages in the order they were received. The receiver thread is an infinite loop that fills the message queue. It blocks to receive a message and then puts it in the message queue. If the queue is ever not long enough to add a new message, the receiver thread doubles the size of the queue and inserts the message.

The API allows a receiver to receive a message with some list of types or a given ID or both. However, ID is favored over type in matching messages. If a receive call contains a non-zero ID, the receive will look through the queue first message with a matching ID. Otherwise, the receive will search for a message with the first matching type. If no matches are found, the thread that called receive is put to sleep in a busy wait to poll the message queue every so often for a matching message.

2.6.3 Future Work

The receive call busy wait/polling is a poor implementation. The code should implement a wait and signal algorithm in the case that the message is not in the queue at the time the receive call is made. The usual case is that receive is called before the message arrives. Thus, on average the receiver has to wait half the polling interval more than necessary after a message arrives. Considering that the message handler object is used for all middleware control traffic, between muxes, middlewares, and clients, this could incur significant critical path delay. At the time of this writing, the sleep time is 100 milliseconds. Considering the number of messages sent in handoff, this could be responsible for over a half second delay.

2.7 Middleware Communication Module



The Middleware Communication Module (MWCM) is the Middleware component used to communicate with other middleware servers. This is a long lived component that creates and manages communication. It also provides the interface for various multiple middleware tasks.

2.7.1 Architecture

The MWCM is a multi-threaded component with several sub-components, shown in Figure 2.5. It contains an interface to send messages to other middleware, a connector service that makes initial connections to other middleware, a daemon process that listens for middleware connections, and a listener service that accepts and handles commands received from other middleware. Through either the daemon thread or the connector, Middleware Communication Channels (MCCs) are created to all other middleware. This forms a fully connected graph of middleware servers(see Figure 2.6).

2.7.2 Implementation

The MWCM is instantiated by the main MWS component and is given a list of neighbor middleware servers. On initialization, it starts two tasks, the dae-

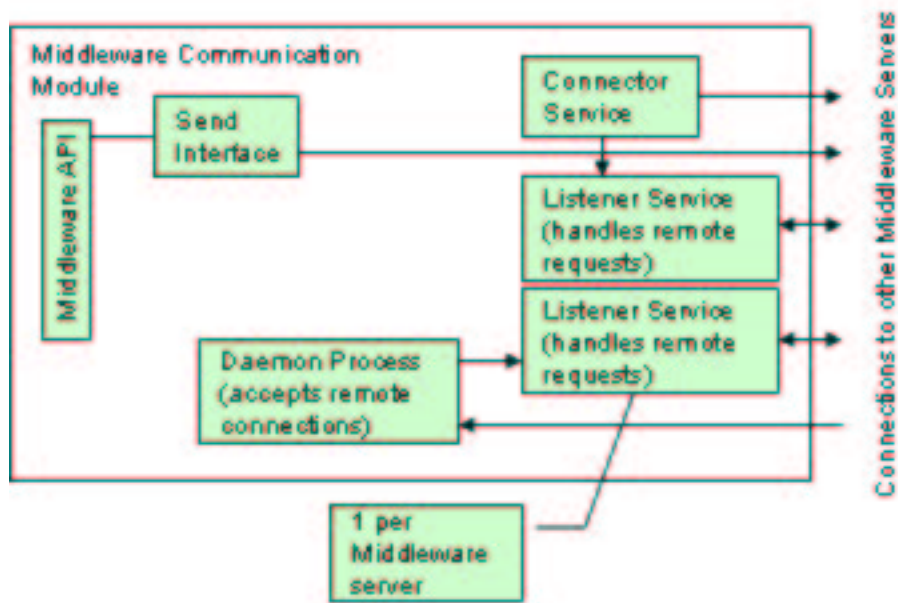


Figure 2.5: Middleware Communication Module Architecture

mon process and the middleware connector. The daemon process listens for connections from other middleware and the connector tries to make outbound connections.

The connector goes through the list of middleware servers and tries to connect to each one. If it fails making a connection, it tries again after a short random interval. On each connection attempt, the connector first checks if a connection has already been made to that middleware server. This would happen if the other middleware server connected to the current one. These connections are maintained in a hash table that has collisions when a middleware server connects to one it is already connected to. This is the case of two middleware servers connecting to each other simultaneously. In this case, one of the connections is dropped.

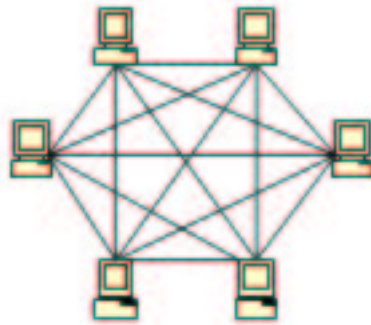


Figure 2.6: MWCM Connectivity Graph

The daemon thread is a simple server that listens on a designated port for connections from other middleware servers. When a connection is made, it tries to add the connecting middleware server to the hash table. If it already exists, the connection is dropped, otherwise, the middleware server is added to the table.

Upon a new connection either through the daemon process or the connector a listener thread is instantiated to handle requests from that middleware server. This waits for any commands over the MCC and handles them appropriately.

The interface provides several abilities to other middleware server components. First is an interface to both send and receive messages. These services look up the appropriate MCC and either send or receive a message from them. Second, the interface allows a middleware server to query all connected devices. This allows a middleware server to find all the client devices connected to the entire middleware service. This is used to locate an appropriate target for handoff. Currently, this service just sends a query to all connected middleware servers and then waits for the responses. It then concatenates the responses and returns the results. Third, the interface allows a caller to learn which middleware server this middleware server is connected to. This is used for middleware server handoff (MASH) and is done by looking in the hash tables to find out which middleware

servers are currently connected.

All the interfaces assume that a middleware server may disconnect at any time. When sending or receiving a message from another middleware server, the calling components need to handle the case that a middleware disconnects after the call is made, but before the message is sent/received. If a middleware server is disconnected at the time of a send or receive call, the MWCM will attempt to reconnect to the middleware server before doing the send or receive. This allows a MWS to be brought down and back up without interfering with other MWSs. Only MWSs attempting to handoff a session to a down MWS will be affected.

The MWCM listener handles responding to many messages associated with gathering information and facilitating middleware only handoff (MASH) or full handoff (FASH). Upon receiving a Query available devices command, the listener requests the device list from the local connection manager and sends a response back to the requesting MWS. Both FASH and MASH commands cause a more complex response from the listener.

When the MWCM receives a MASH command it starts the middleware hand-off process. First, it uses information from the MASH message to create a local session object. This object accepts the information from the previous middleware and will handle the session once the handoff is complete. The MASH message also contains the Internet address and port that the device executing the MASH is listening on. (Handoffs work similarly to initial session creations, in that the device requests a new session, or a handoff, and provides a port for the middleware to connect back to.) The MWCM then opens a connection to the target device, creating the new session control channel (SCC). The MWCM then starts the session control thread (SCT) for the new session.

A FASH command causes a similar initial reaction as the MASH command.

It begins by creating a local session object from data received in the MASH command. It then waits for the previous middleware to send the rest of the session data. The second message contains cached objects that need to be available to the new device. Once the MWCM has received the cached object, it starts the process to push the session to the new device. It sends a push request to the target device over the DCC. The target device will then complete the FASH by initiating a pre-session and then merging the data from the transferred session.

2.7.3 Future Work

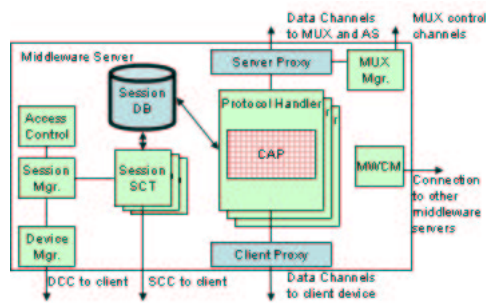
The MWCM is a first cut at middleware communication. There are several deficiencies in the design. The first is the middleware discovery algorithm. Middleware discovery is done through configuration files at each middleware. Also, each middleware must know the addresses of all middleware that it will connect to. At a minimum, the middleware should be able to share information about other middleware through a gossip protocol. In that way, each middleware could have a configuration file that connects it to at least one other middleware. The middleware could then share information to become fully connected.

Device discovery is another shortcoming. Currently, to do full handoffs, a middleware sends a message to all other middleware to find target devices. Minimally, caching should be done to avoid this overlay broadcast. Other options include not querying all middleware. It is unlikely that an individual user needs to know all the possible device targets. Through use of GPS or other means, filtering could be done to only ask about devices that are near the user.

A fully connected network of middleware in itself does not scale. This may not be an issue if the middleware are able to handle enough clients. However, if this is not the case, middleware communication should be redesigned to allow

for a less connected network through caching, multi-hop routing, P2P, or other means.

2.8 Session Database



The session database is the middleware component that stores all information relevant to a session in a session record. Like other components of the iMASH system, data in the session database is accessed using the session ID. The data includes information about the session profile, data channels, transcoding state, and a transcoding history. The session database also contains the object store and cache for the session.

2.8.1 Architecture

The session database stores two types of things: session data, and data objects requested over the session data channels. Both are indexed with the session ID. The session database is essentially a hash table containing pointers to records for each session. In each session's record, the session data is stored. There is also an additional hash table containing all the stored objects for a session. These objects have an additional identifier that is used as the hash key. This identifier is set by the protocol handler when an object is stored and is protocol dependent.

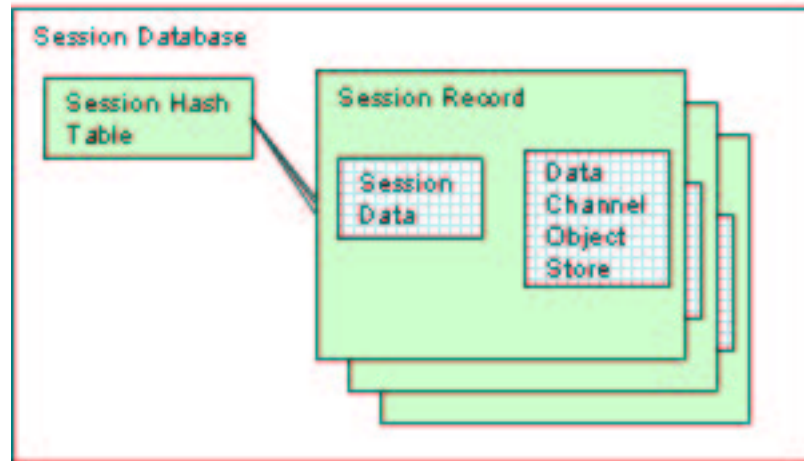


Figure 2.7: Session Database Architecture

For web traffic, this identifier is generally the URL of the object.

2.8.2 Implementation

As stated in the architecture section, the implementation consists of a hash table for the session records and then an additional hash table for each session object store. The important note in the implementation is that this is a memory only implementation. When an object is put into the middleware object store by the protocol handler, it remains in memory.

2.8.3 Future Work

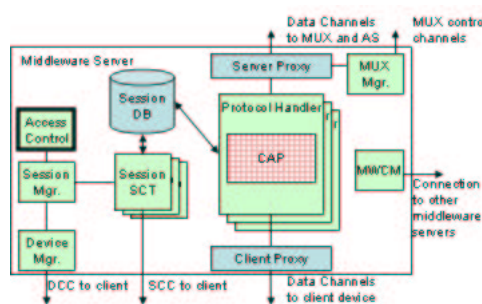
The session database has many deficiencies, the main problem being wasteful use of memory. All objects are stored in memory and because of this, a large number of session objects will cause poor performance. In a more robust implementation,

objects could be written out to disk. Then, there is no limit to the number of objects that can be saved to the object store. Ideally, the object store should have a cache type behavior, such that old items get removed. However, since cached items are used in handoff, the simple solution to retain everything was chosen to avoid the complexity of items not being present.

The current implementation is also one of the main limiting factors in the middleware. Only so many connections can be handled due to the fact that only a small number of objects can be requested over session data channels before middleware memory is exhausted. The only way that objects are removed from this cache is when a session is terminated. At that time, the middleware cleans up all resources used by the session including the session database.

Given the current implementation, performance gains are certainly possible. Modifying this database to always read and write from disk might increase performance significantly. Even taking a slow disk into account, much performance could be achieved simply by avoiding memory paging once many objects exist in the cache.

2.9 Security



The security model of the iMASH system is a two tiered approach based on

both authentication of devices and authentication of the users of those devices. The device layer authentication is done when a device connects to the iMASH service. The device and the middleware perform a WTLS type 3 handshake when the connection takes place. In this handshake, certificates are exchanged. The middleware verifies that the device has a valid certificate to use the iMASH service and the device verifies that it is talking to a valid middleware server. Once this mutual authentication is done, the device layer channels are set up and the device is registered with the middleware.

The user level authentication is done from a registered device that has undergone the first check. When a iMASH session is started, the user must login with a username and password to authenticate that session. Once the session is authenticated, it is assumed to be mobile and can move to any registered device connected to any middleware server.

The WTLS authentication that is done to register a device uses certificates. Our security model assumes that there is a trusted certificate authority that can be used to verify devices and issue new certificates to devices that wish to join the iMASH service.

The trust model for iMASH follows from the two authentication scheme. Middleware servers implicitly trust each other. On boot up, middleware servers contact each other and exchange certificates to verify each other. Once this is done, it is assumed that a middleware server will not become a rogue server. Also, once devices connect to a middleware server, they trust all middleware servers. Users and their sessions don't necessarily trust the devices, but trust the middleware servers. The users are able to trust the devices that the sessions are on due to derived trust in the middleware. The interesting scenario is during handoff. A user gains trust in a remote node that a session is moving to because it has reg-

istered with the middleware and provided a valid certificate. This trust model does not prevent a valid user from compromising their own sessions, or a valid device from compromising those sessions that move through it, but it does prevent unauthorized devices or users from posing a problem. Also, in the case of an insider attack, the damage is limited. A user can only compromise their own sessions and a device can only compromise the sessions that run there. More detail about the security model can be found in [SKP02].

The device layer control is done by the device connection manager and the user level control is done by the access control module.

2.9.1 Access control

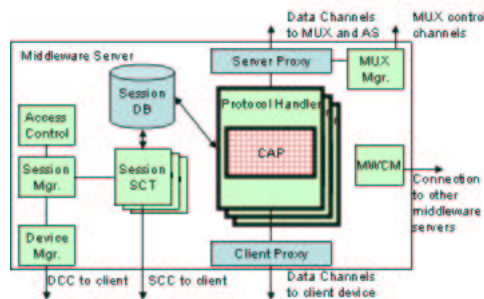
Access control is a middleware component that verifies if a user is authorized to use the middleware service. Currently access control is only done on the user/password granularity. This component is really just a wrapper that allows iMASH to tie in to the access control done on the operating system. Since the middleware currently runs on Linux workstations, this external library uses the checks */etc/passwd* and */etc/shadow* to determine if the user has permission to use the machine.

2.9.2 Future Work

There are several avenues for future work in access control for the middleware. The first being group associations. There should be an iMASH group, such that only members of the iMASH group can use the middleware service, thus differentiating valid system users from iMASH users. Also, assigning devices and users to particular subgroups provides for interesting extensions. A user and session might only be allowed to exist on a device that belongs to the same group. This

would allow more complex session/device relationships than the current model of any session can move to any device. This can further be extended to the idea that a session can transfer ownership. Currently a session is associated with a particular user for its lifetime. Perhaps the system could allow a session to transfer ownership between users of the same group. In this transfer, the session would inherit the capabilities of the new user and lose those of the old. However, allowing this ability opens up security “Pandora’s box”. Several questions arise, such as: What happens if a user has a secure authenticated connection to an application server, such as a stock broker? Should iMASH allow a iMASH user transfer while maintaining an authenticated AS connection or not. Should client Application data be cleansed when transferring between users, otherwise making a non-user-transferable tag to client data objects? Security questions like these have extensive ramifications and should be dealt with carefully. The iMASH architecture should not reduce security by adding mobility.

2.10 Protocol Handler and the Content Adaptation Pipeline



The Protocol Handler (PH) and the Content Adaptation Pipeline (CAP) together form the module that transcodes data to given device characteristics. That is, they take an object o and apply a function $f(o)$ on that object to form a new object o' that is then sent on to the client device.

The general use of these components is to distill data for either device or network constraints. If a device has a maximum screen size of 320x240 pixels, or only has a greyscale monitor, there is no point in sending a huge full color image to the device. The device may not be able to display the image or simply may not have the memory to handle it. The job of the protocol handler and CAP is to reduce the image (or any other type of object) to something that the device can handle. Mobile devices also tend to have poor network connections. Thus, the Protocol Handler/CAP pair may reduce the fidelity of an object simply to save bandwidth.

The decisions of what objects to content adapt and what adaptations can take place are all dependent on the session profile for an application. This session profile is a combination of the device profile, which contains specifics about a device and the user profile which has user or application preferences for how data objects should be transcoded.

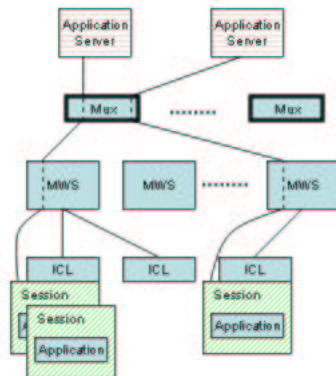
The Protocol Handler and CAP are designed to work together. The protocol handler deals with application protocols and the CAP handles the actual transcoding of objects. For example, with a web connection, the protocol handler parses the HTTP on the way down from the server. It then parses out embedded images and objects and send them to the CAP. The CAP has the job of actually adapting the objects. It could reduce the image size or reduce the colors. The Protocol Handler then re-inserts the objects back into the Protocol Handler to be sent to the client.

The Protocol Handler is also not constrained to modifying only the downstream messages. It has control of both the up and downstream data channels to the application server. This is necessary in some cases because the upstream channel needs to be adapted as well. An example of this is encryption. If a client

device is not powerful enough to do encryption on a data channel, the Protocol Handler could do it on behalf of the device. However, in this case, both directions of data would have to be adapted to achieve transparency to the application server.

The exact details of how the protocol handler and CAP make decisions and how they transcode objects is beyond the scope of this paper. For information regarding the CAP see [PZB02b].

2.11 Mux



2.11.1 Architecture

The mux is a lightweight component in the iMASH system that is designed to hide the details of handoff from the application servers. It simply acts as one extra hop for a data channel on its way to an application server. A middleware opens a data channel to the mux and the mux makes the connection out to the application server. When a middleware handoff does take place, the mux is able to hide the transfer of data flow from the AS.

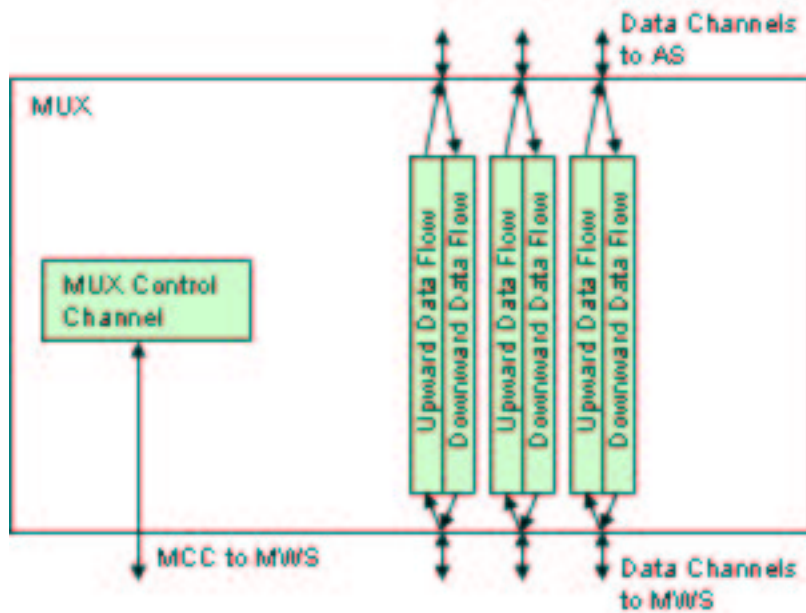


Figure 2.8: Mux Architecture

2.11.2 Implementation

The mux has a simple design that has similarities to the middleware itself. It has a control component that maintains a Mux Control Channel (MCC) to each middleware server and a pair of threads for each data channel that is flowing through it. The two operations that the mux is capable of is data channel creation and data channel handoff.

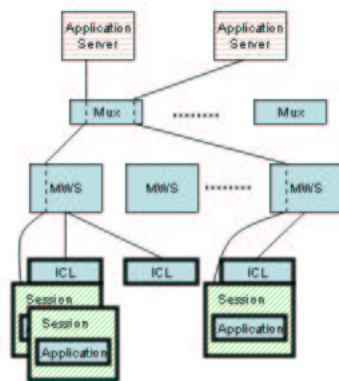
Data channel creation begins by one middleware sending a *data channel request* message to the mux. In response to this it will create an id for the new data channel and then connect out to the application server. The mux will then connect back to the middleware, completing the data channel connection. If the data channel creation has gone well up to this point, the mux will send a *ack data channel request* message to the middleware containing the mux data channel id. If something has gone wrong, an error will be returned. At this point, the mux

creates two threads to pump the data through and waits for either a data channel termination (implied by one entity closing the channel) or a handoff message.

If the mux receives a handoff message over its control channel to a middleware, it begins transferring the data channel to the new middleware. This request may come from any middleware that has the correct data channel id. These requests are trusted because they come from authenticated middleware over the encrypted MCC.

The handoff process is similar to data channel handoff in the middleware. The data channels are first suspended and then the mux connects back to the new middleware. The data flow is then transferred to the new connection and the middleware sends a *ack handoff to MWS* message to the new middleware. If any errors occur, the mux sends an error message to the middleware instead.

2.12 iMASH Client Layer



The iMASH client layer is a lightweight layer that runs on all devices connected to the iMASH service. It is intended to run on all ranges of devices from powerful workstations to current low end devices such as phones and PDAs. The main job of this layer is to access iMASH services in the middleware layer. This

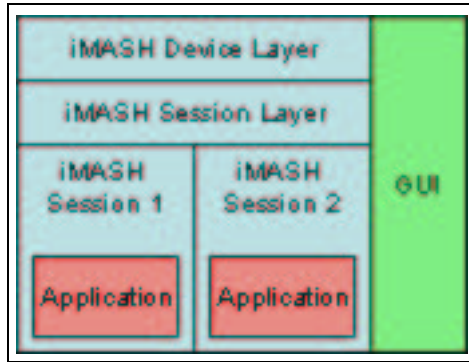


Figure 2.9: iMASH Client Layer Architecture

involves communication with the middleware to perform tasks and proxying of network connections through the middleware service to enable handoff.

The client layer is broken up into three main parts, a device layer, a session layer, and a GUI for the user to execute iMASH commands. There is one device layer and GUI per device, but there may be several independent iMASH sessions utilizing the session layer (see Figure 2.9). Each application running on a device that utilizes iMASH services runs in its own session and may move independently of other sessions for the purposes of handoff.

2.12.1 Device Layer

The device layer is responsible for registering a specific device to the middleware service and carrying out tasks relevant to a specific device. It interacts with the GUI for instructions from the user and has the ability to create new sessions on a device and accepts sessions handed off from another device.

When the iMASH Client Layer is started on a device, the device layer is the first piece to come up. It begins by registering with the middleware service. It does this by making a device control channel (DCC) to an available middleware server. Currently, this is done by finding the middleware server in a configuration

file, but this could easily be extended to using a discovery service to find an available middleware server.

Once the DCC is connected, the the device must authenticate itself to the middleware server. This is done through a WTLS type 3 handshake in which certificates are exchanged and both the middleware server and the device mutually authenticate each other². This process also creates a secure channel that uses strong encryption so all future communication is secure. The details of this security was described in section 2.9.

Once the device layer has authenticated itself, it looks for a device profile for the device. The device profile is an XML document that describes the characteristics of the device. It allows the middleware to make intelligent choices about transcoding data for each device. If this profile exists, it is sent to the middleware. The device layer also starts up a daemon process to listen for middleware messages over the DCC. This is needed because notifications of middleware handoff come over the DCC. Once the registration is complete, the new device becomes a candidate for handoff from sessions started elsewhere.

2.12.2 Session Layer

The session layer is above the device layer and utilizes the services provided by the device layer. It's main task is to handle an application running on the iMASH infrastructure. There can be multiple session layers on a given device and each application is housed in a separate session.

The session layer provides an API to the iMASH application that allows it to access iMASH services. These include network connections routed through the iMASH service and storing and retrieving application state for handoff.

²In the current iMASH implementation, certificates are not used.

A new session layer is instantiated on a device when either it arrives from handoff or a user starts a new application via the GUI. This causes the device to contact the middleware through the device layer DCC, creating a session control channel (SCC). Once the SCC is created, session control traffic can begin flowing. This traffic includes messages to negotiate session data channels (SDCs), checkpoint and restore state and instantiate handoff.

Session data channels are also considered to be part of the session layer. When the application requests a network connection to a given server, a corresponding session data channel is created. This connection is then routed through the middleware service. Data flowing through the middleware can then be transcoded as necessary for device and network characteristics.

The session is also the unit of mobility. When a application handoff occurs, the whole session is transferred from one device to another. Thus, all iMASH client state, the application, and the data channels are all transferred to the new device, which then can restore state and begin receiving network data through established connections. The data however may be transcoded to fit the parameters of the new device and network characteristics.

2.12.3 GUI

The client GUI is simply a means for the user to access iMASH functionality. The user instantiates all session related actions, such as handoff, session suspend and resume. This GUI is a lightweight component that accesses the functionality that is built into the session and device layer. The session layer functionality is composed of handoff, suspend and resume. The device layer is responsible for instantiating a new application, or rather creating a new session, which then starts the application. The design intent is that the GUI could be changed

depending on the form factor of the device, with the rest of the code remaining static. However, currently, only one implementation of the GUI exists.

CHAPTER 3

Protocols

This section contains detailed information on all the steps taken by various parts of the architecture for each of the protocols. The key shown in Figure 3.1 details how to read the subsequent protocol figures. The protocols described are: 1. Device connects to MW; 2. Session Creation; 3. Data Channel Creation; 4. Session Handoff; 5. Middleware Server Handoff; 6. Suspend; 7. Pull (Restore); 8. Get Session Object; and 9. Reconnect Data Channel.

3.1 Device connects to MW

When the iMASH software on the client device starts up, it attempts to contact the middleware immediately. It first reads the middleware address and port number from the *icl.conf* file. Using that information, it makes a connection to the middleware with a *WTLSSocket*.

Upon connection, the WTLS server and client sockets begin a WTLS type 3 authentication.¹ Type 3 authentication begins with mutual certificate exchange and verification for the client and server. The client device verifies that the MWS is not a rogue server and the MWS verifies that the device has a valid certificate

¹At the time of this writing, WTLS type 3 authentication is not being done (This is due to its removal when the WTLS security layer was ported from a C library to Java). Instead, WTLS type 2 authentication is being used. Type 2 does not do the certificate exchange and verify section of protocol. As a result of this, the pre-master secrets (used to calculate symmetric encryption keys) are not exchanged. Hard-coded values are used instead.

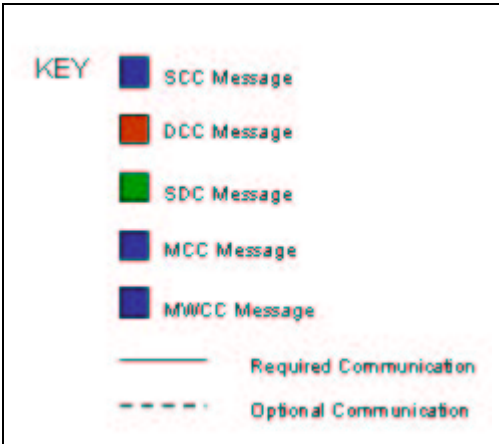


Figure 3.1: Protocol Figure Key

to utilize the iMASH service. After the certificate exchange portion, initialization vectors (IVs) and symmetric encryption keys are generated according to the WTLS specification[WAP]. The WTLS specification supports many symmetric ciphers, however our current implementation only utilizes DES and 3DES.

Once the WTLS handshake is done, the device is fully authorized to use the iMASH service. If this handshake fails, the connection is dropped by the middleware. When the client completes the handshake, it checks for a device profile. The profile path is stored in the configuration file. If the profile exists, the client builds a *Profile Command* message and sends it to the middleware. The middleware receives this message over the DCC and stores the profile in a hash table along with other information associated with the device.

At this point, the iMASH client layer initiates the GUI and waits for the user to start an application. This then begins the process to start a new session discussed next.

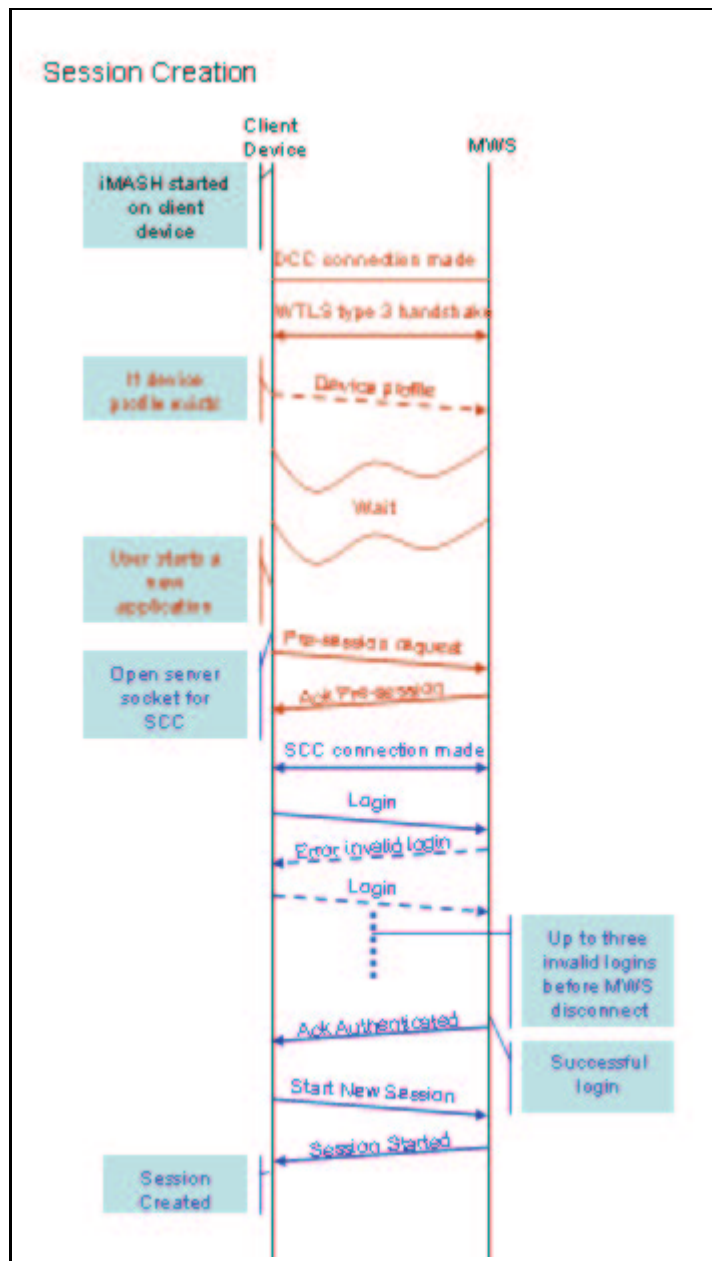


Figure 3.2: Session Creation Protocol

3.2 Session Creation

The process (Figure 3.2) to start a new session begins when the user starts a new application via the iMASH client layer. This is currently done via the iMASH client GUI, which initiates the *newSession* call to the client layer. This in turn creates the session on the client end.

The first step is for the client to create a session control channel (SCC) with the middleware. The client creates a server socket to receive the SCC connection and then sends a pre-session request message to the middleware over the DCC. The middleware responds to this message with an *ack pre-session* message back to the client. This acknowledgment contains encryption keys and parameters that will be used on the nascent SCC.

The middleware then connects back to the client using the port information contained in the pre-session request message. Once this connection is made, both parties may begin sending encrypted messages over the new channel using the encryption parameters decided by the middleware.

The next stage of session creation is for the user to authenticate herself. This is done via username and password. The client software either uses a pre-cached name/password or prompts the user. This information is assembled into a *LOGIN* message and sent to the middleware. The middleware then verifies that the username and password are correct (This is currently done via JNI and a call to a native Linux library that checks the user/password in the password files.). If the user is valid, the middleware returns an *ACK_AUTHENTICATED* message to the client. Otherwise it returns an invalid user message to the client.

If the client receives an invalid user error, it will prompt the user to try again. This will repeat the login phase for a number of times. Once the maximum

number of tries has occurred, the middleware will drop the connection.

Upon receiving a positive login response from the MW, the client session layer sends *START_NEW_SESSION* message to the MW. The middleware then formalizes the session in the middleware by creating a session ID. The session ID is an MD5 hash of the username, and current time. The session is also added to the session database, using the ID as a key. The middleware then responds to the client with a *ACK_SESSION_STARTED* message that contains the session ID. When the client receives this acknowledgment, it loads the application and initiates it with the *applicationInit* function defined in the *ImashApplication* interface.

Once this process has completed successfully, the application and user are have all the abilities associated with a session. The application may create data channels and any type of handoff may take place.

3.3 Data Channel Creation

An application opens a data channel to an application server by calling either *openConnection* or *openDatagramConnection* from the *ImashClientInterface*.² This service call will return a *Socket* or *Datagram socket* respectfully. In order to provide this service, the iMASH client layer, middleware, and mux communicate to create the connection that goes from AS to MUX to middleware to the Device.

This protocol (Figure 3.3) begins with the client layer opening a server socket to accept a connection for the nascent data channel. The client layer then sends a *dataChannelRequest* message to the middleware containing the server socket

²This section will describe the steps for a TCP stream connection, but the steps are essentially the same for a UDP datagram connection.

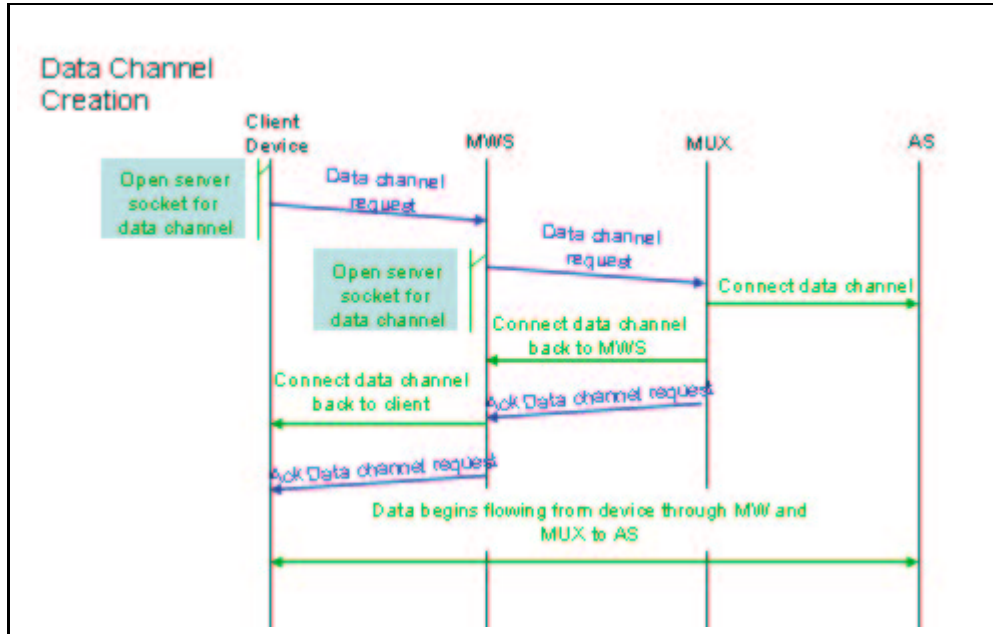


Figure 3.3: Data Channel Creation Protocol

port.

The session control thread receives this message and creates a unique channel ID for the data channel. (This number is used for all future references to the channel by the middleware. Then, the middleware requests a data channel from the server proxy. It calls *createDataChannel* with the host and port of the application server, the unique channel number, and the protocol, in this case. Server Proxy then checks to see if the channel exists and if not, creates a stream data channel or datagram data channel depending on the protocol.

Upon instantiation, the stream data channel asks the mux manager for the message handler to send messages to the mux over the MCC. If the mux manager does not already have a connection to a suitable mux, it makes that connection and then returns the message handler. The data channel then opens a local server socket and sends a *data channel request* message to the mux. The request

contains the AS address, AS port, protocol, and the local server socket port.

The mux receives the data channel request and uses the information to connect to the application server. The mux also connects the data channel back to the middleware server that issued the request. Then, the mux sends an ack (or error) back over the MCC.

Once the connection is created on the server side, the middleware calls the *createDataChannel* service from the Client Proxy. The client proxy then connects back to the client on the port that it created at the beginning of the protocol. After sending the ack, the session control thread starts the protocol handler for the data channel. This allows data to start flowing through the middleware and also transcode data objects as necessary, based on the client and device profile. The middleware-mux and client-middleware hops of the data channel also use strong WTLS encryption from the beginning. This gives protection on the last hop to the device, which is likely to be the most vulnerable for the wireless devices iMASH attempts to address.

3.4 Session Handoff – CASH or FASH

The CASH and FASH protocols, shown in Figures 3.4, 3.5, 3.6, and 3.7 describe the process for handing a an active session from one device to another. The CASH case occurs when both devices are attached to the same middleware and the FASH case occurs when the two are resident on different middleware servers.

The process begins with the user requesting a handoff through the iMASH client layer GUI. This causes the ICL Session layer to send a *query available devices* message to the middleware server. This message is received by the session control thread over the SCC. The SCT then forwards this query to all other

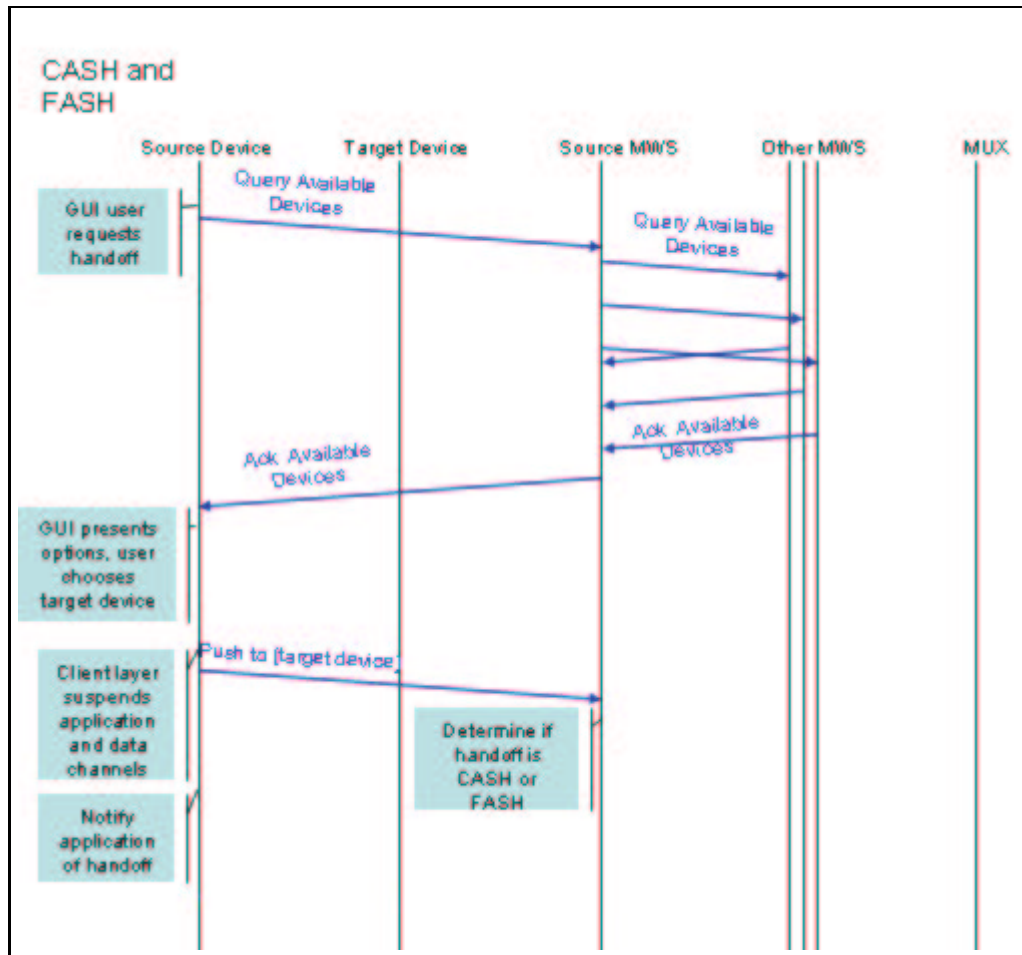


Figure 3.4: Client Handoff Begin Protocol

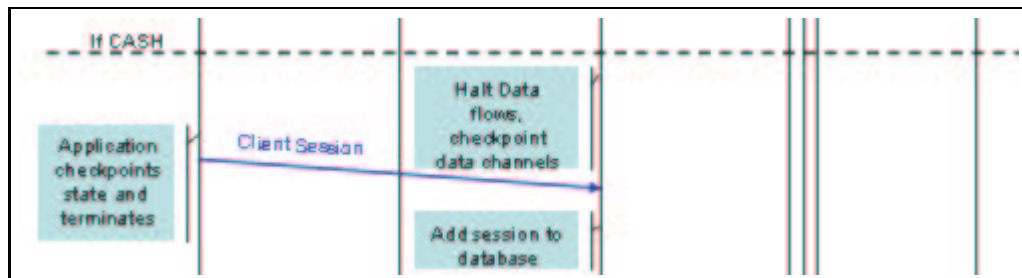


Figure 3.5: CASH Protocol

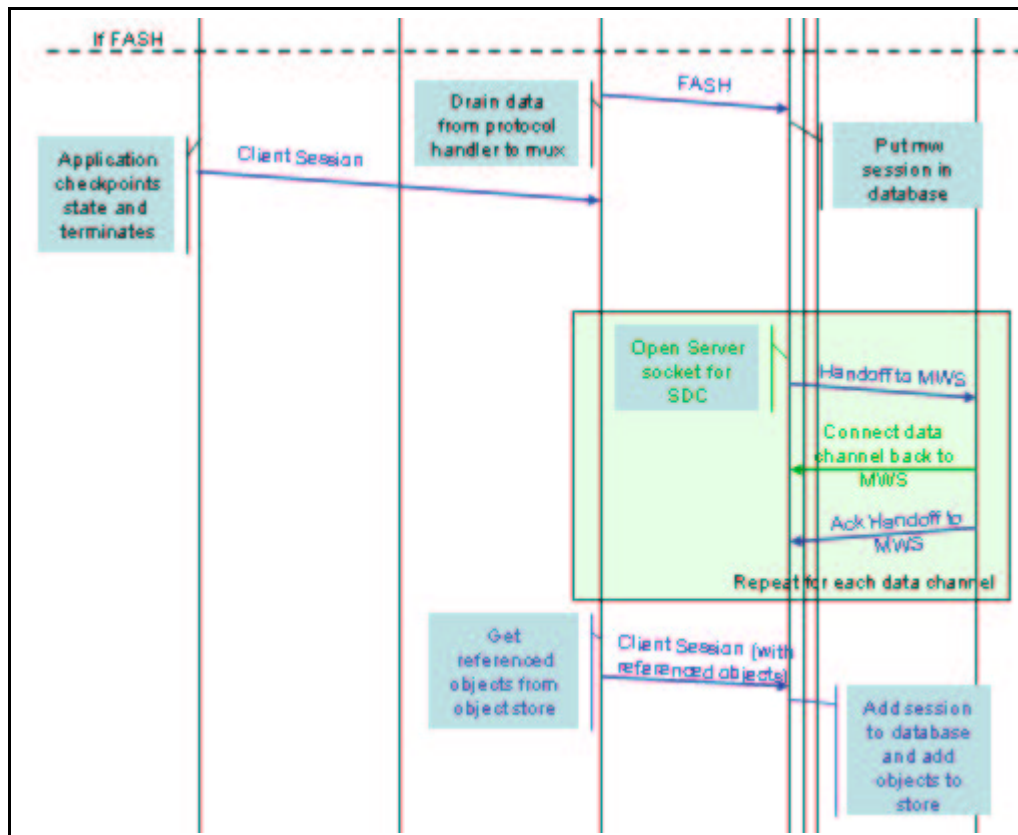


Figure 3.6: FASH Protocol

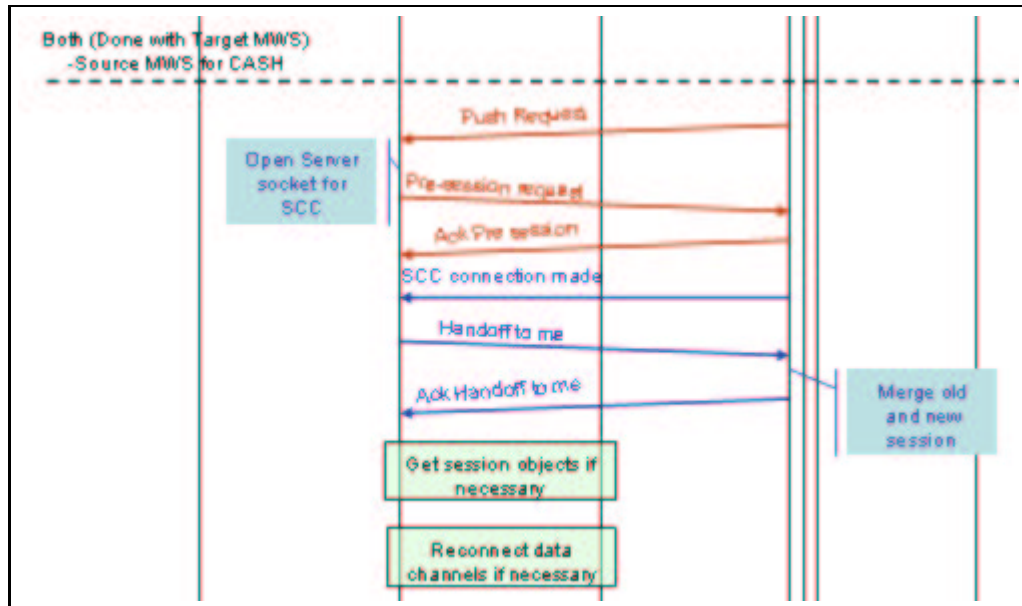


Figure 3.7: Client Handoff Finish Protocol

middleware servers using the middleware communication module. The request is broadcast to all middleware servers, which reply with a *ack available devices* message containing all the clients that have an active DCC. The source middleware server then checks its own connected devices and creates a master list of all connected devices. This information is then sent back to the client device in a *ack available devices* message.³

Upon receiving the reply, the GUI presents the user with possible target devices for handoff and the user chooses one. The ICL session layer then sends a *push to* message to the middleware server with the chosen device. This message is sent to the middleware session control thread that then determines if the target device is on the current middleware server or remote.

The local handoff (CASH) case is fairly simple. The SCT halts the data flow

³In a production system, each middleware would do access control to see which devices the requesting user has access to, but the current implementation simply returns everything.

through the protocol handler for all data flows and then proceeds to checkpoint the data channels. In this process the protocol handler saves any state that it needs to be restarted with the current stream. The protocol handler then terminates for each data channel. Once this is complete, the SCT waits for the client to send the client session. This is the data structure that contains all application check-pointed data and client side iMASH data.

The remote (FASH) case requires a bit more work by the middleware. First, it stops and checkpoints the data channels as in the CASH case. At this point, it notifies the target middleware of the handoff by sending a *FASH* message containing preliminary information about the handoff. It then drains all remaining data from the mux. This has the effect of blocking until all data channels have been transferred to the target middleware.

When the target middleware receives the *FASH* message, it stores the middleware session state in the session database and starts a session control thread for the session. The new SCT then goes through the process of handing off all the session data channels from the mux to the current middleware. This is done by the target MWS sending a *handoff to MWS* message to the mux for each data channel containing the data channel id. The mux then suspends the data channel, initiates a connection to the target MWS and restarts the data flow. After the data channel is connected it also sends a *ack handoff to MWS* message to the target MWS.

After sending the *FASH* message, the source MWS waits for the client session from the client. Once received, it examines the check-pointed state for referenced objects in the middleware cache. If any of these exist, it attaches those objects to the message. This message is then forwarded to the target middleware, which adds the objects to its object store and the client session to the session database.

The source client's role in handoff is the same for the CASH and FASH case. Once the *push to* message is sent to the source middleware, it signals the application that a handoff is taking place. It is then the application's responsibility to checkpoint state and terminate. Once the state has been check-pointed, the ICL session layer sends a *client session* message to the middleware that will get forwarded to the target middleware for a FASH.

The remainder of handoff, shown in Figure 3.7 is the same for both CASH and FASH. The target middleware (this is the same as the source for a CASH) sends a *push request* message to the target device over the DCC. This is received by the MWS Listener daemon resident in the iMASH device layer on the target device. The device then may choose to accept or reject the handoff.⁴ If it rejects, it replies with a *push deny* message over the DCC, otherwise, it begins setting up a new session.

Creating a new session from a handoff is similar to creating a new session. The device sends a *pre-session request* message to the MWS over the DCC. The middleware replies with a *ack pre-session* message containing encryption information and connects back to a port the device had opened. Once the new SCC is created, the protocol differs from the standard new session creation. Instead of sending a *login* message, the device sends a *handoff to me* message containing the session id. (The session id was sent to the device with the *push request* message) The middleware SCT receives this message and looks up the session in the session database. If it exists, the middleware sends the client session to the device in a *ack handoff to me* message. The middleware also merges the old session data into the new session.

When the ICL session layer receives the *ack handoff to me* message, it begins

⁴Currently, the device always accepts a handoff.

restarting the application on the new device. It first retrieves the application name from the client session and then looks for it on disk. If the application is found, the application is instantiated and then given tags⁵ for its saved state. Once the application is running, it may retrieve state using the get session object protocol or reconnect data channels using that protocol.

3.5 Middleware Handoff

The MASH protocol, shown in Figures 3.8 and 3.9 begins with the user selecting a middleware handoff from the client GUI. This in turn causes the ICL session layer to send a *discover MW* message. The session control thread receives this message and queries the middleware communication module to find out which middleware it is connected to. This information gets packed into a *ack discover MW* message and sent back to the client. The middleware server choices are then presented to the user.

Once the user chooses a target middleware server, the ICL session layer prepares for the middleware handoff. The iMASH client layer suspends all data flowing through the data channels. Then it notifies the application that a middleware server handoff is taking place. The session layer then opens up a new server socket to accept the new SCC connection and sends a *Change MW* message to the current middleware server. This causes the middleware server to check-point the data flows and middleware session as it is done in the FASH handoff case. The source middleware server then sends a *MASH* message to the target middleware server.

When the target middleware server receives the *MASH* message, it goes

⁵When an application saves a piece of state, it gives it a name known as a tag.

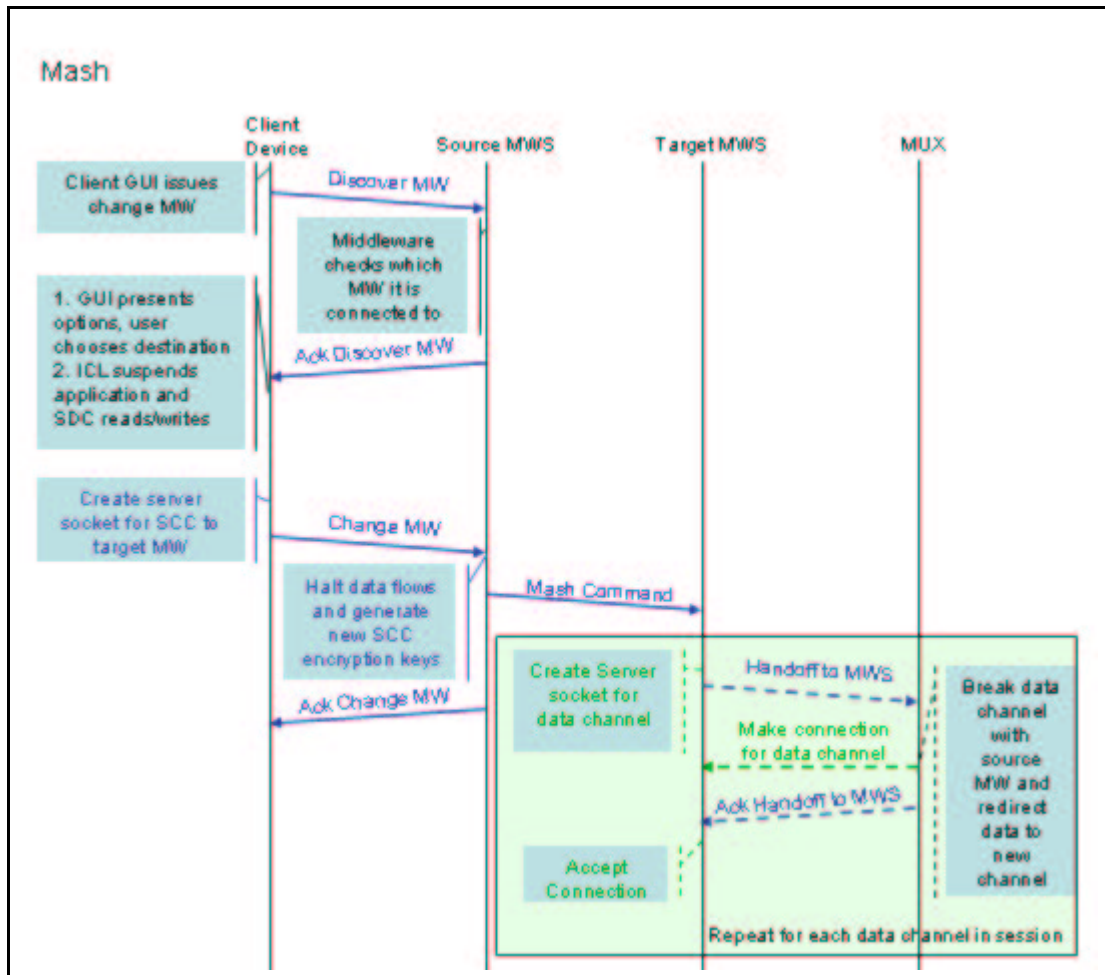


Figure 3.8: MASH Protocol part 1

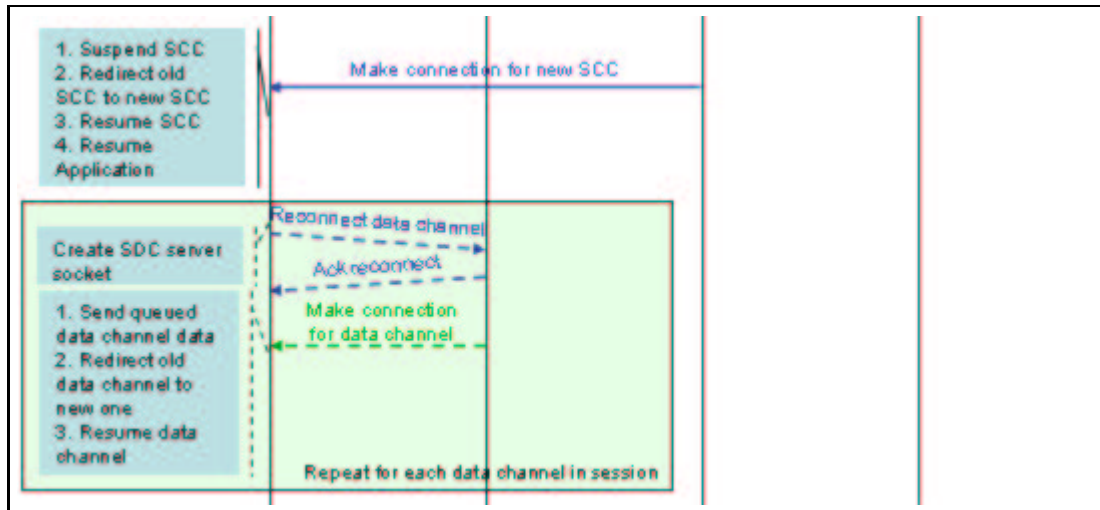


Figure 3.9: MASH Protocol part 2

through a similar process as in a FASH. The middleware server must transfer all the data channels from the source middleware server by sending *handoff to MWS* messages to the mux. Once all the data channels have been transferred, the target middleware server initiates a TCP connection to the client for the new SCC.

The client transfers its control channel to the new SCC and then begins the process of reconnecting all the data channels. Once all the data channels have been reconnected, the data channels are resumed and the application is notified that the middleware handoff is complete.

3.6 Suspend

A user may suspend a session at any time with the iMASH client GUI. This is useful if a user needs to stop using a device because of mobility or power reasons and does not have a current handoff target. This feature causes iMASH to initiate the first half of a handoff without completing it. Session state is saved at the

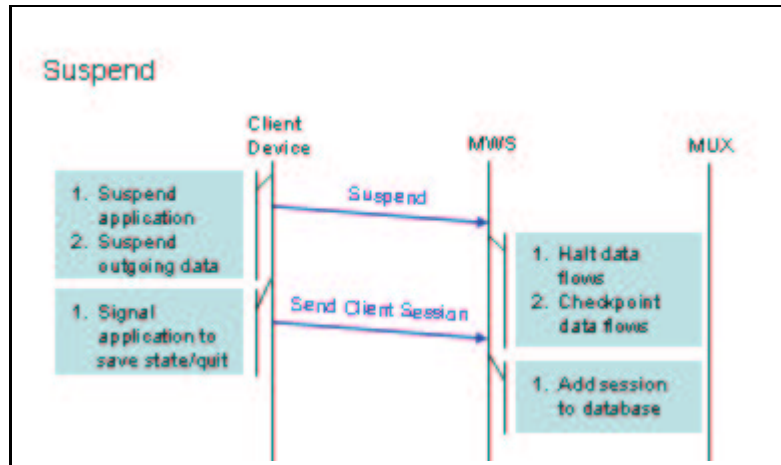


Figure 3.10: Suspend Protocol

middleware until the session is pulled to a device at a later time. Also, all active network data channels will remain open as in normal handoff⁶.

The protocol to suspend an application is trivial. Once the user has indicated that a suspend should take place, session data channels are suspended. Then a *suspend* command is sent to the middleware via the SCC. Upon receiving the suspend, the middleware will stop the data flows through the middleware and wait for the check-pointed session data. After sending the *suspend* command, the iMASH session layer signals the application that a handoff is taking place. The application then checkpoints its state and quits. This check-pointed state is then sent to the middleware in a *client session* message. When it arrives at the middleware, the state is added to the session database and the session is halted.

This session then remains at the middleware until it is pulled to another device. In the future, suspended sessions should be purged from the middleware after some point (to avoid forgotten sessions occupying middleware resources).

⁶No data is transmitted over the data channels while a session is suspended. Thus, an application server may close a connection that it sees as idle.

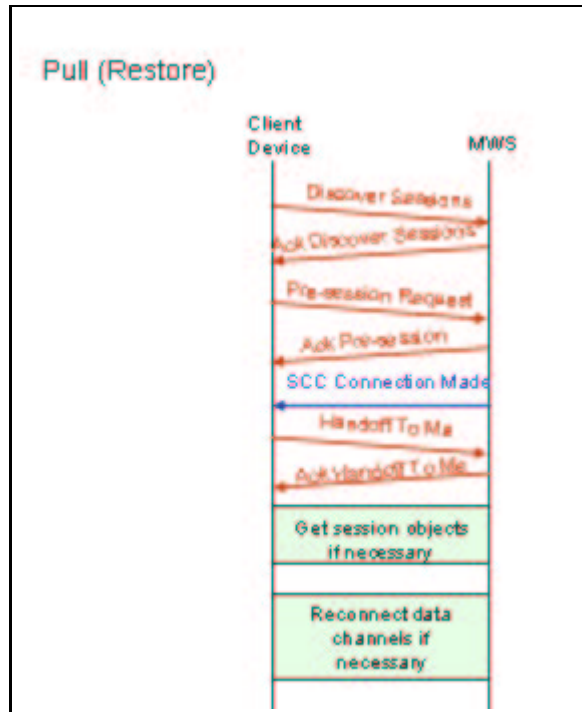


Figure 3.11: Pull (Restore) Protocol

3.7 Pull (Restore)

From a device that has successfully logged into the system, a user may pull suspended sessions to the local device. This is useful if a user had suspended a session at one location before leaving. Sometime later, the user arrives at another iMASH enabled device. From there, they can pull the session to the new device and resume work.

This protocol assumes that the device has already authenticated and has a DCC to the middleware.

The user elects to pull a session to the current device using the GUI menu. The GUI will then prompt the user for a username and password for the session that will be pulled. Using the username acquired from the user, the iMASH client

Device Layer sends a *discover sessions* message over the DCC to the Middleware. The middleware then replies with an *ack discover sessions* message containing a list of that user's sessions. This list contains both the application name and the session ID.⁷

The user then selects a application out of the list to be resumed on the current node.

Once the user has selected a session, it is resumed in the same way that sessions are resumed after handoff. The device sends a *pre-session request* message, and receives a *ack pre-session* message. Then, the SCC connection is made and the device sends a *handoff to me* message containing the session ID. Upon receiving the acknowledgment, the application is started and may resume data channels in the same way as after handoff.

3.8 Get Session Object

This protocol, shown in figure 3.12 is used by the application to restore state after a device handoff. Once the handoff has completed and a SCC channel has been established, the application may request state that had been check-pointed before handoff. This is done by the application making calls to the iMASH Client Interface API layer.

⁷There are two major bugs in this implementation. The first is a security bug. The user is never authenticated. At no time does is the user's name and password checked. The GUI asks for a user, but any username could be put in. The problem is that the discover sessions command should only be allowed over the SCC *after* the user has been authenticated. It was done this way to make it similar to handoff, but in that case, an authenticated user requests the handoff, where in this case, an anonymous user is given the same power. The current implication is that any user can resume any suspended session.

The second flaw in this protocol is that sessions are only queried on the local middleware. If a user was to suspend a session on one middleware and then try to pull it from a device connected to another middleware, it would fail.

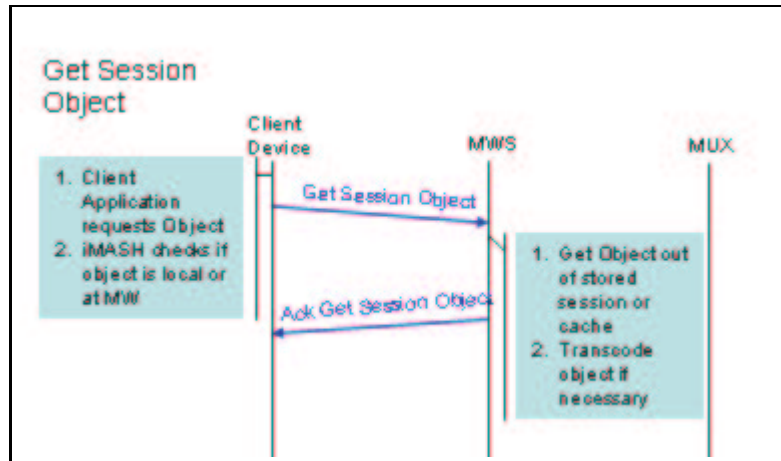


Figure 3.12: Get Session Object Protocol

The application asks for one object at a time by making an API call with an object's tag. (When each object is check-pointed, it is given a name by the application so it can be retrieved later.) The iMASH client layer then checks to see if the object is local. This check is necessary because during handoff, some objects are sent to the new device automatically, and others are kept at the middleware.

If the object is local, it is simply returned to the application. If this is not the case, the client layer sends a *get session object* message to the middleware over the SCC. The message contains the object's tag, which the middleware uses to locate the object in the session object store. The middleware then makes a determination if this object should be transcoded. Currently, any object that was requested from an application server is subject to transcoding.

This object is then sent through the CAP to create an object more suitable to the current device. Once transcoded, the middleware sends the object back to the client device over the SCC in a *ack get session object* message. If the middleware could not find the object, then it sends an error back to the requesting device.

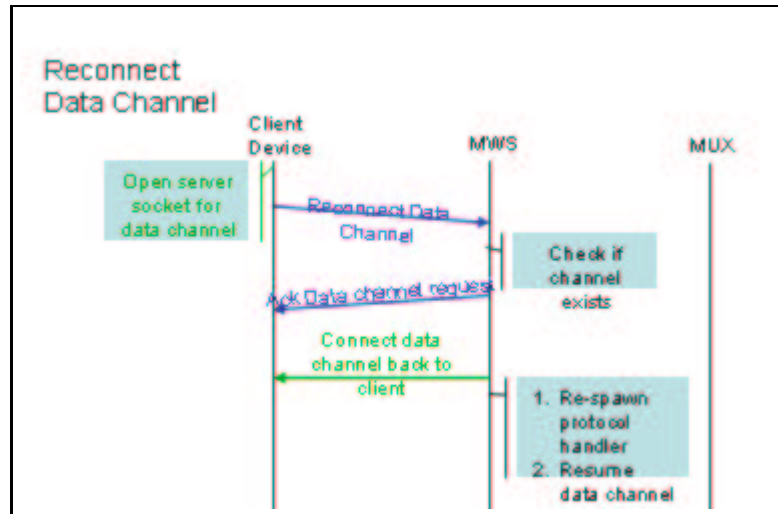


Figure 3.13: Reconnect Data Channel Protocol

When the iMASH client layer receives the object, it returns the object to the requesting application.

3.9 Reconnect Data Channel

The protocol (Figure 3.13) to reconnect a data channel is virtually the same as the data channel creation protocol. It begins by the application requesting a data channel be reconnected after a handoff. The application passes the data channel id to the iMASH client layer, which the client layer uses as an index to look up the data channel ID that was given to the channel by the middleware. The client layer then opens up a server socket to listen for the middleware to connect back. (Like data channel creation, the client asks for a connection and the middleware connects back to the client.) The client layer then send the *reconnect* message to the middleware.

This message is received by the session control thread for the given session on the middleware, and it begins the process of resuming the data flow to the

new client. First, encryption keys are created for the reconnected data channel. These are then sent back to the client in a *ack data channel request* message. The SCT then calls the client proxy to reconnect the data channel to the new client.

The client proxy first makes a connection to the target device and then utilizes the functionality provided by handoff sockets to transfer the stream from the old socket to the new. The SCT then resumes data flow by instantiating a new protocol handler for the stream and resuming the socket flow that was suspended at the beginning of handoff.

CHAPTER 4

iMASH Applications

The iMASH architecture is flexible by design, but it does put some constraints on application design. Those constraints include changes in invocation, subtle differences in the networking model, asynchronous signaling, and handling handoff appropriately.

4.1 General Design Constraints

iMASH applications must extend and implement the iMASH Application interface. This interface defines how a application is invoked in the iMASH architecture as well how the application is signaled when handoff is initiated by the user. Both of these interfaces are function calls into the application.

A application is invoked by iMASH calling the *applicationInit()* function defined in the interface. iMASH first creates a thread for the new application and then starts the application by calling *applicationInit()*. This function provides the application all the information necessary to communicate through the iMASH interface. First, it gives the application the *iMASH Client Interface*. This allows the application to create iMASH data channels and save state as necessary for handoff. It also gives the application object tags and a flag indicating if the application is newly spawn or initiated due to a handoff.

If the application is a new instance, it proceeds as normal. If it has been

initiated due to handoff, it uses the object tags to reconstruct its state prior to handoff using iMASH calls. Thus, an application is responsible for saving and reconstructing its state before and after handoff.

Another difference in the iMASH architecture is that an application is run as a child process of the iMASH Client Layer. Thus, an application needs to exhibit good behavior. If it makes an *exit()* call, it will kill the whole process, including the iMASH client layer. The architecture is designed this way mainly for simplicity. In a more robust iMASH implementation, the application could communicate to iMASH through IPC and iMASH could protect itself from application misbehavior.

4.2 Networking behavior

iMASH attempts to be completely transparent in its network redirection. However, there are subtle differences that may cause problems. iMASH may suspend and stop network data flow at any time in response to a handoff. Thus, when an application is sending data, it needs to be able to handle check-pointing partially sent, or partially received data objects. This causes the application to be either more robust to re-requesting data, or be smart enough to get part of an object before a handoff and part after. If an application server and client have a complex protocol or have real-time constraints on data delivery, this could cause problems.

4.3 Asynchronous Signaling and Handoff Semantics

A user can request a handoff at any time and this signal will come to the application asynchronously by iMASH calling the *handoff* function within the applica-

tion. It is the application's job to check-point its state and receive all remaining socket data before this function returns.

The application may also send data, but it does not actually get sent. Before calling the handoff function, iMASH suspends the outgoing data. Any data that the application sends is actually buffered until after the handoff and sent then. This is done to ensure that no data is lost during handoff, but requires thought on application design to handle this behavior.

Once the application returns control from the handoff call, iMASH provides no guarantee on socket connectivity. iMASH will close sockets at will and send check-pointed state to the new device. The application is then left to exit on its own. It may quit immediately, or continue for some length of time. This may be useful if the application is playing a video and wants to empty its buffer before quitting. This decision is application dependent, but if an application chooses to remain active, it is possible that the handoff will complete before the application terminates. In this instance, the same application would be running on two different devices in the same semantic session.

4.4 Save pointing and Restoring State

The iMASH client interface provides three ways to save and restore state in iMASH. Each type of state is handled differently by the iMASH infrastructure. An application has the choice of saving Application Objects (AOs), Reference Objects (ROs), or G-Functions.

4.4.1 Application Objects

An Application Object (AO) is any piece of data that the application wishes to save. Any byte array or Serializable Java object can be check pointed by iMASH architecture as an AO.

AOs are treated as opaque objects by the iMASH architecture. That is, they are transferred from one device to the other in handoff without modification. AOs are also pushed from the source to the middleware and then from the middleware to the target device during handoff. Any object that is created by the application software must be classified as an AO to be transferred by iMASH.

4.4.2 Reference Objects

Reference Objects (RO)s are another way for an application to save some pieces of state. Any object that was retrieved from the network through the iMASH infrastructure is a candidate to be an RO.¹ An example RO is an image sent over the network to a web browser.

Due to the nature of iMASH content adaptation, any object that an application retrieved from the network may have been content adapted, meaning it had some F-Function $f(o)$ applied by the protocol handler and CAP. If such an object is check pointed as a RO, it may be re-adapted to fit the characteristics of the new device after handoff. Also, ROs are not sent from the client device at hand-off, since they already exist in the middleware object store. This could provide significant savings for a bandwidth constrained device. Also, since the objects are re-adapted, handing off between devices with vastly different characteristics should not be a problem.

¹Any object that can be saved as a RO may also be saved as a AO. However, iMASH treats the two objects differently.

4.4.3 G-Functions

G-Functions are the third way to save state and are meant to be used with reference objects. If an object is retrieved from the network, it can be saved as a RO and then sent re-adapted to the new client device. This would be ideal if the object was not modified. However, applications frequently modify data. That is where G-Functions fit in. These functions are applied to the object in question prior to content adaptation. The object is then adapted and sent to the new client. That is, if an object o was originally content adapted and a device received $f(o)$, the application can apply a G-Function $g(o)$ to the object on handoff. This G-Function will occur on the original object before the object is content adapted to the new device. Thus, on handoff, the second device would receive $f'(g(o))$. With this understood, it is important that any G-Function would apply to the original object as well as the transcoded object. If this order of operations is not semantically correct, an application should convert the RO into an AO before applying such a function. In that way, the second device could receive $f'(g(f(o)))$ if that makes more sense in a given situation. This could happen if a content adaptation $f(o)$ changed the basic type of the object. A simple example of this is a speech to text engine. An operation g to make the text bold or italic might not have a representation in the original object o that was speech.

To use G-Functions, extra middleware software needs to be in place. First of all, there must be a referenced piece of software at the middleware to do the G-Function in question. Thus, each application that uses such functionality will need to provide this to the middleware.

G-Functions may be applied to ROs or other G-Functions. In this way, G-Functions may be chained together to provide complex transformations to an

object prior to sending it to the target device. These G-Functions are applied to the object at the time it is requested by the target device when application state is being restored.

CHAPTER 5

Results

The iMASH architecture presented in the previous chapters has been demonstrated to be robust and adaptable to a range of applications, while providing acceptable performance.

5.1 Applications

Many applications have been developed or ported to the iMASH architecture. Many of these have been toy applications to demonstrate certain features of the architecture, but more mature applications have also been adapted.

The first application developed was a simple image viewer called mini-browser that requested images over the network from a simple server application. It was mainly used to demonstrate the iMASH transcoding abilities through handing off the application between laptop computers and a Compaq IPAQ handheld.

The next application was designed to show the streaming handoff capabilities of the iMASH middleware servers. A java wrapper was created around the closed source MpegTV video player. This wrapper handled application checkpointing and restoring for handoff. This application was able to handoff a streaming mpeg stream between two desktop or laptop computers which had the MpegTV software. The iMASH video player wrapper was then extended

to handle multiple MPEG players¹. Thus, the application could adapt to the environment available on the device and use whatever video output application was available. This application also gave credibility to the argument that an application state could be handed off between compatible applications.

The next two applications were tested the benefits iMASH could bestow on real applications. A open source java telnet/secure shell application called JTA was retrofitted to the iMASH architecture and a wrapper was created for Galleon, a web browser based on Mozilla. JTA required a fair amount of work, because the application required a lot of state to be check-pointed prior to handoff as well as new functionality to be added to handle dynamic environment changing that occurs with handoff. Galleon on the other hand adapted to the iMASH architecture almost seamlessly. Galleon already had a checkpoint and restore capability designed for crash recovery. Every time a user opens a web page, the state is saved to a crash recovery file. The iMASH wrapper simply had to transfer this state to the new device and restart the Galleon application.

5.2 Extensibility

The current iMASH architecture is not tied to any one application. The application framework is simple, and requires only minimal work from the application designer. The main difficulty is only that an application can save and restore it's state using iMASH services. The state transcoding at the middleware can also be extended. As described in the various CAP papers [PZB02b, PZB02a], new types of content can be adapted by writing new adapter plugins for the middleware servers. Along the same lines, if an application runs a new protocol, new

¹The MpegTV software was not available for the iPAQ handheld used in our tests, thus the mpg123 application had to be used instead

protocol handler modules can be written to handle any type of new data.

5.3 Performance

iMASH performance can be measured by many metrics. Handoff time, data retrieval overhead, and transcoding time are all important to the overall performance of the system. Many experiments have been run to test the iMASH system, the following experiments originally appeared here [BBC03].

5.3.1 Data retrieval overhead

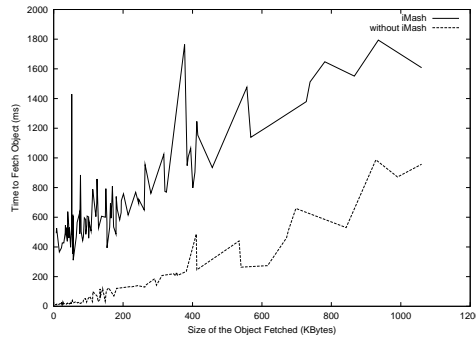


Figure 5.1: Average client latency experienced on object request (Y-axis), sorted by object size (X-axis), for iMASH and non-iMASH environments.

In this experiment the mini-browser application tested the overhead of the iMASH system. In the first case, the application retrieved a random set of images with varying size from the application through the iMASH service. In the second case, the application retrieved the same set of images communicating directly with the application server. Figure 5.1 shows the results of this experiment. As you can see, the latency produced was around .5 seconds for small images and upward to 1 second for bigger images. This is accounted for by the double proxying nature of sending the images through the middleware server and mux

components of the iMASH service.

5.3.2 Handoff performance

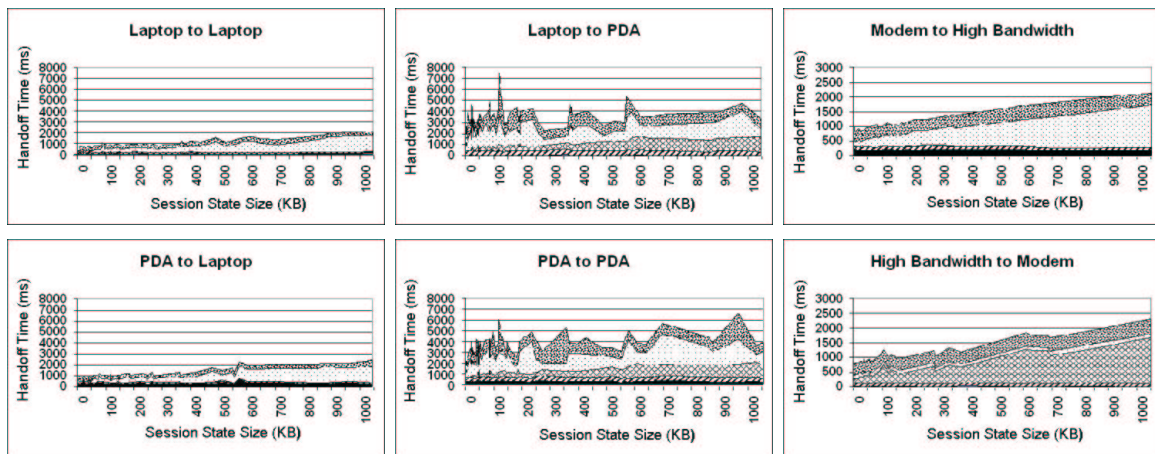
In this experiment, the latency for a CASH handoff is examined varying various parameters including hosts, network bandwidth and session size. The laptops in the experiments are Dell Inspiron 4000 and 8000 with 800Mhz CPUs running Linux and the PDA is a Compaq IPAQ with a 206 Mhz CPU also running Linux.

Figure 5.2 shows the results of these experiments. The latency in these experiments shows that a CASH takes anywhere from 0.5 to 7 seconds and is broken up into several components including state transfer, image transcoding, and resuming the application on the target device. What we see is that handoff is proportionate to the state size being transferred and more time is required when the state must be transcoded mid-handoff. This is one reason that transferring a session takes longer from a Laptop to PDA than in the reverse case. More processing must also be done on the target end of a handoff to restart the Java Virtual Machine, which also adds latency when handing off to a slow device.

In addition to experiments with CASH handoff, experiments were also done with Full Handoff, FASH. In these experiments, the source and target client were not connected to the same middleware server. Thus a middleware handoff also needed to take place. Figure 5.3 shows the results. As can be seen the handoff latency is similar to the CASH case, but with slightly longer handoff times. This time is accounted for by the latency that was incurred by transferring middleware state from one MWS to the other.

These experiments on handoff latency show that handoff takes place in a reasonable amount of time. Handoffs are user initiated and they take place in a matter of a few seconds. This is at least as fast as a user is able to stop working

on one device and begin to work on the next. The delay is only noticeable when either a lot of transcoding must be done mid-handoff, or the application has a large amount of state to transfer. Both of these would be either the result of poor application design in a mobile environment or applications that are not suited for mobility.



(a)

(b)

Figure 5.2: Client latency experienced on CASH 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

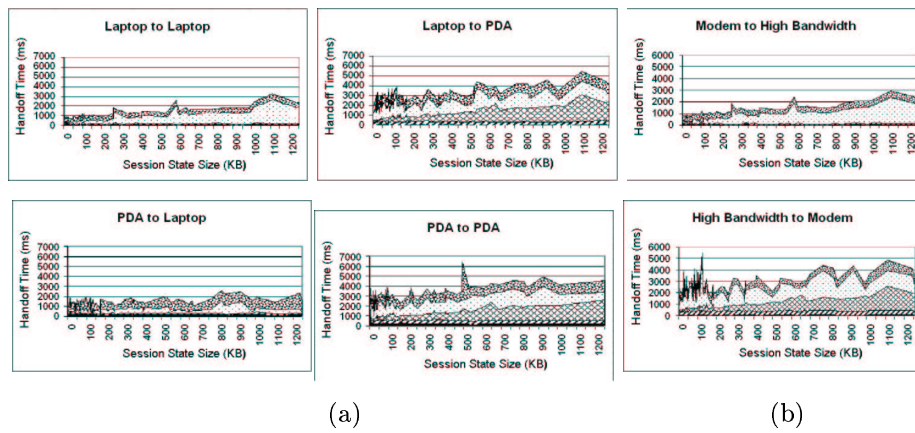


Figure 5.3: Client latency experienced on FASH, as a function of session state size. The X-axis units are bytes, ranging from $\approx 1\text{KB}$ to $\approx 1.3\text{MB}$; the Y-axis units are milliseconds. The upper curve represents total handoff latency.

CHAPTER 6

Conclusions

Pervasive, anytime, anywhere computing is a direction that the computer world is moving toward. The iMASH project has shown one aspect of this picture. Through iMASH, an application breaks free from the confines of a single device and is able to move with the user while continuously adapting to the environment. This mobile changing environment has certainly presented many challenges in the design of the iMASH system, but iMASH has addressed them. The main issue solved by iMASH is mobility, but iMASH also handles content adaptation, resource discovery, and security. The realm of pervasive computing is far from solved, but the iMASH project has shown what is possible.

This thesis has described the architecture designed to enable application mobility. It centers around the middleware servers, which provide the services required for mobile computing. All communication from client applications to remote servers is proxied through the iMASH architecture. This allows iMASH to freely redirect data without the application knowing about it. The architecture also provides a service to allow save-pointing and recovery of application state, which becomes necessary with application mobility. Applications are also provided with the handoff service, which moves an application from one device and restarts it on the second. The application recovers its state through iMASH and all open data channels are redirected to the new device.

The use of multiple devices within an application session led to new prob-

lems. Differing device constraints necessitated data transcoding. Data that flows through the iMASH system is custom transcoded on the fly for each device. When a handoff takes place, images for instance may be scaled down in size to fit on the display of a PDA.

Throughout the design of the system, security needed to be addressed. In order for an application to move from one device to another, a trust model was developed to include the devices and the middleware. Mutual authentication of middleware servers and devices addresses the trust issues and all iMASH data is encrypted using strong encryption provided by WTLS.

The results of our efforts are positive. The implementation of the system has shown that performance for all aspects of the system is good. Handoffs can be done in a matter of seconds and the overall overhead of the iMASH system is not that great for a single application.

The iMASH architecture has also shown itself to be robust and adaptable to a wide range of applications. These applications include a image viewer, a SSH application, and a full featured web browser. We have also run the iMASH client software and applications on the Windows, Linux, and Solaris platforms.

The mobile computing world is a environment with many challenges. iMASH has brought many of these challenges to light and presented solutions to those challenges. iMASH has made mobile computing a reality.

APPENDIX A

iMASH messages

This section contains a detailed list of all the messages sent between the various components of the iMASH architecture. All these messages are a derivative of the `ImashCommand` class. That class handles the message type, and id fields. It also has a general data and length field. For convenience this class has the ability to send and receive a copy of itself. On the receiving end, once the generic message is received, the receiver can check the type field and then create the specific message from that. Each subclass of `ImashCommand` is thus responsible for creating an instance of itself from a general `ImashCommand`.

- **ACK_AUTHENTICATED**

source/destination: Middleware to client device

channel: SCC

data: none

description: This message is sent in response to a successful *LOGIN* message.

- **ACK_AVAILABLE_DEVICES**

source/destination: middleware to client device or other middleware

channel: SCC, MWCC

data: Array of device names connected to the middleware server.

description: This message is sent in response to a *query available devices* message from a client device or another middleware. This is used in the push protocol by a client attempting to find a target device. First the client queries the middleware and then the middleware contacts all other MWSs and creates a list of suitable target devices for handoff. This data is then sent back to the client with the *ack available devices* message.

- **ACK_CHANGE_MW**

source/destination: middleware to client device

channel: SCC

data: none

description: This message is sent back to the client device in response to a *change middleware* message. Receipt of this message indicates that middleware handoff will take place.

- **ACK_DATA_CHANNEL**

source/destination: mux to middleware and middleware to client device

channel: MCC or SCC

data: Encryption Data, channel ID

description: This message is sent in response to a successful *data channel request* message. It contains the encryption primitives to be used once the connection is established. Thus encryption starts on the data channel from the beginning without a expensive handshake. The channel ID is a reference that the client or middleware will use when referencing the channel with the sending entity. Each entity keeps it's own references for data channels. That is done so there is no problem with numbering between

multiple client devices or multiple middlewares. When another entity wants to suspend, resume, handoff, etc a channel, it uses the reference to indicate which channel should be suspended, resumed, or handed off.

- **ACK_DISCOVER_MW**

source/destination: middleware to client device

channel: SCC

data: Array of available middleware addresses and ports.

description: This message is sent in response to a *discover middleware* message. The MWS checks which other middleware servers it is connected to and sends that back to the client. This is typically done before a middleware handoff.

- **ACK_DISCOVER_SESSIONS**

source/destination: middleware to client device

channel: DCC

data: Array of session IDs for sessions suspended by a given user.

description: This message is sent in response to a *discover sessions* message over the DCC. The middleware checks the session database and returns all suspended session IDs to the user with this message.

- **ACK_GET_SESSION_OBJECT**

source/destination: middleware to client device

channel: SCC

data: a session object initially stored by a client device.

description: Upon receiving a *get session object* message, the middleware will attempt to find the desired object in its object store. If the object is found, the middleware will apply all necessary g-functions and transcoding on the object and then send it to the client device. This is done through the *ack get session object* message.

- **ACK_HANDOFF_TO_ME**

source/destination: middleware to client device

channel: SCC

data: client session

description: This message is sent by the middleware to complete the handoff protocol. It will occur either in traditional handoff or resuming sessions from being suspended. The message contains the client session. This client session contains all the necessary information for the device to restart the application and restore itself to previous state.

- **ACK_HANDOFF_TO_MWS**

source/destination: mux to middleware

channel: MCC

data: mux port, data channel protocol

description: This message is sent in response to a *Handoff to MWS* message received at the mux. The port field is primarily used for UDP traffic and indicates which port the mux is sending from. This way the MW is able to ignore other packets.

- **ACK_PRE_SESSION_APPROVAL**

source/destination: middleware to client device

channel: DCC

data: encryption keys to be used to the new SCC

description: This message sends the encryption keys to the client device that will be used on the new SCC for the session. This is necessary so that keys do not need to be negotiated over the new channel. The key negotiation requires public key cryptography and is computationally and time expensive for low end devices. Since the DCC is a secure, encrypted channel, these keys can be transferred safely from the middleware to the client in this manner. This message is sent in response to a *PRE_SESSION_REQUEST* from the client to the middleware.

- **ACK_PUSH_DENY**

source/destination: client device to middleware

channel: DCC

data: session id

description: This message is sent by a device in response to a *push request* message. When the device decides for whatever reason that the session is not welcome, on the device, it will send this message back to the middleware instead of completing the push protocol¹.

- **ACK_SESSION_STARTED**

source/destination: middleware to client device

channel: SCC

data: none

¹This message is never sent. It is provided for completeness, but this aspect of the iMASH system was never realized

description: This message is sent by the middleware in response to a *start new session* message. It's purpose is to notify the client device that the middleware has finished creating a new session.

- **CHANGE_MW**

source/destination: client device to middleware

channel: SCC

data: Address of the target middleware and client port number (so the target middleware can connect back to the client).

description: This message is sent by the client to initiate a middleware handoff. After doing discovery, the client chooses a target middleware and sends this message to the current middleware. The target middleware will connect back to the client device to complete the MASH. There is no response to this message, other than the target middleware connecting back to the opened client port.

- **DATA_CHANNEL_REQUEST**

source/destination: client device to middleware, middleware to mux

channel: SCC (for client to middleware), MCC (for middleware to mux)

data: AS IP address, AS port, client port, protocol, encryption flag

description: This message is sent by the iMASH Client Layer when an application requests a data channel to an application server. The middleware will then set up the data channel and send another *data channel request* message to the mux. Once these two messages are acknowledged (with a *ack data channel* message), and the connections are made, the data channel will carry the application server data.

The AS IP address and port are self-explanatory fields. The client port is the port that the target entity will connect back to when the data channel is connected. That is, if a client requests a data channel from the middleware, the middleware will initiate a connection back to the client to complete the data channel. This was done to eliminate excess threading and waiting on both sides. The encryption flag indicates whether the application requests encryption on the client-middleware and middleware-mux links. The mux-application server link is always un-encrypted. If the application requires end-to-end security, it needs to provide it with the application server.

- **DISCOVER_MW**

source/destination: client device to middleware

channel: SCC

data: none

description: This message is sent by a client device to learn about new middleware servers. This message is sent at the beginning of the MASH protocol to find a target MW.

- **DISCOVER_SESSIONS**

source/destination: client device to middleware

channel: DCC

data: username associated with the sessions that the client devices is trying to discover.

description: This message is sent at the beginning of the pull/restore protocol. A user first asks the middleware what sessions are out there and then has them sent to the current device. The middleware will respond over the

DCC with a *ack discover sessions* containing the session IDs that the user has suspended.

- **ERROR**

source/destination: various

channel: various control channels

data: error code, text error message

description: This is a wrapper for all errors in the iMASH system. Also, all errors that do not have their own specific message will be sent with this type.

- **ERROR_INVALID_COMMAND**

source/destination: middleware to client device

channel: SCC

data: error string

description: This message is sent by the middleware whenever the client issues a command that is unexpected by the protocol or issues a command code that does not exist.

- **ERROR_INVALID_USER**

source/destination: middleware to client device

channel: SCC

data: error message

description: sent in response to a login message that has a bad username or password.

- **ERROR_NO_DATA_CHANNEL**

source/destination: mux to MWS

channel: MCC

data: error message

description: This message is sent by the mux to the middleware when it can't connect to the AS or can't connect the data channel back to the MWS.

- **ERROR_NO_SESSION**

source/destination: middleware to client device

channel: SCC

data: error string

description: This message is sent to the client device whenever the client attempts to do something that requires a session before a session exists.

- **ERROR_NOT_AUTHENTICATED**

source/destination: middleware to client device

channel: SCC

data: error string

description: This error is sent by the middleware when a client attempts to do something that requires authentication before it has provided authentication. This would happen if a client tried to request a data channel before sending the *login* message.

- **FASH**

source/destination: middleware to middleware

channel: MWCC

data: Middleware session data, target device for FASH

description: This message is sent by a middleware to initiate a handoff to a device connected to another middleware server. The message contains all the session data from the first middleware so the second can complete the handoff. The message also contains the name of the target device for the handoff. Upon receiving this message, the second middleware will begin transferring the data channels and contacting the target device to complete the handoff.

- **GET_SESSION_OBJECT**

source/destination: client device to middleware

channel: SCC

data: object tag

description: When an application stores an object as part of session state, it is stored with an object tag. When an application is restoring that state possibly after handoff, it requests the object with the tag. If the object is not in the local cache for the device, the client layer will send a *get session object* message to the middleware. The middleware will then find the object in its object store, do any necessary transcoding and send it to the target device.

- **HANDOFF_TO_ME**

source/destination: client device to middleware

channel: SCC

data: Session ID

description: This message is sent by a device to complete handoff. Once it has set up a pre-session with the middleware, it sends a *handoff to me* message over the nascent SCC to complete the handoff. The data for this message is the session ID which acts as a capability to transfer a session. If a device has a session ID, it is considered authorized to move a session. In response to this message, assuming that the session exists, the middleware will send an *ack handoff to me* message containing the client session data necessary to start the application on the new device.

- **HANDOFF_TO_MWS**

source/destination: middleware to mux

channel: MCC

data: Target MW IP address and port, data channel protocol, and the mux channel ID.

description: This message is used during handoff to move data channels from one middleware to the other. Currently, the target middleware sends this message, requesting that the mux redirect the data flow. The IP address and port are used to indicate the port that the target MWS is listening on to receive the connection.

- **LOGIN**

source/destination: client device to middleware

channel: SCC

data: username, password

description: This message is sent by the device to authenticate a user in a nascent session. In the normal flow, this message directly follows a

ACK_PRE_SESSION_APPROVAL message. The middleware will respond with an *ACK_AUTHENTICATED* message or an *ERROR_INVALID_USER* message.

- **MASH**

source/destination: middleware to middleware

channel: MWCC

data: MW session, client IP address and port, encryption keys.

description: Once a client device has prepared for a middleware handoff, the source middleware initiates the handoff by sending a *MASH* message to the target middleware. This message contains the necessary MW session information to move the session and necessary data for the target MW to reconnect the SCC. The target MW then connects SCC to the client at the specified address and port. The control channel traffic is immediately encrypted with using the created encryption keys.

- **PRE_SESSION_REQUEST**

source/destination: client device to middleware

channel: DCC

data: SCC port number

description: This message is used to initiate a new session on a client device. It is sent over the DCC and contains the port number for a newly created server socket on the client. The middleware uses this port number to create a connection back to the client. This new connection becomes the session control channel for the new session. The Middleware responds with an *ACK_PRE_SESSION_APPROVAL* message on success or an error message on failure.

- **PROFILE**

source/destination: client device to middleware

channel: DCC or SCC

data: XML profile.

description: This message is used to send either device profiles or user/session profiles to the middleware. Device profiles are sent over the DCC and session profiles are sent over the SCC. At the time of this writing, only device profiles are implemented and used by the CAP and Protocol Handler. There is no response to this message.

- **PUSH_REQUEST**

source/destination: middleware to target device in handoff

channel: SCC

data: session ID

description: As part of the handoff protocol, the middleware sends a *push request* message to the target device. The target device may then deny handoff by sending an error, or accept the session and use the session ID to complete the handoff procedure.

- **PUSH_TO**

source/destination: client device to middleware

channel: SCC

data: target device name

description: After completing the discovery protocol, a device initiates a handoff, either CASH or FASH, by sending a *push request* message to the

middleware. The device then checkpoints its state, sends it with a *client session* message, and ends gracefully. The push is completed by the middleware to the target device and the application is restarted.

- **QUERY_AVAILABLE_DEVICES**

source/destination: client to middleware or middleware to middleware.

channel: SCC, MWCC

data: user name associated with the session.

description: This message is sent by a client device and then forwarded by the middleware to other middleware to discover target devices for handoff. Once the MWS has consolidated the list, it replies with an *ack available devices* message.

- **RECONNECT**

source/destination: client device to middleware

channel: SCC

data: channel id, port number for middleware to connect back to

description: This message is sent by the client device to reconnect a data channel after handoff. The message contains the middleware's channel id for this SDC and the port for the middleware to connect back to. In response to this message, the middleware will send a *ack data channel request* message and connect the data channel back to the client.

- **SEND_CLIENT_SESSION**

source/destination: client device to middleware or middleware to middleware

channel: SCC or DCC

data: XML client session data, referenced objects, and the session id

description: This message is sent by the client device after it checkpoints its state. The client session contains all the state that the client is sending to the new device. This state is saved in XML format and then sent. The state gets sent to the new device with the *ack handoff to me* message.

When the middleware is sending this message to another middleware, it may also contain referenced objects. When the handoff includes a middleware change, the source middleware sends cached objects referenced in the client session to the target middleware. That way, when the target client device requests those referenced objects, they are available at the new middleware.

- **START_NEW_SESSION**

source/destination: client device to middleware

channel: SCC

data: none

description: This message is the last in a series of messages used to create a new session on the middleware and client device. It is sent after a client has completed the login protocol successfully. Its purpose is to differentiate between starting a new session and a user pulling a suspended session (which also requires login²). The middleware will respond to this message with a *ack session started* message.

- **SUSPEND**

²As described in the pull protocol, this was never implemented correctly. In the current state of iMASH, the pull protocol is not done here and the start new session message is always sent after a successful login

source/destination: client device to middleware

channel: SCC

data: None

description: This message is sent to the middleware to indicate that the user is suspending the session.

APPENDIX B

Acronyms

- *3DES* - Triple Data Encryption Standard
- *ACL* - Access Control List
- *AS* - Application Server
- *ASH* - Application Session Handoff
- *CAP* - Content Adaptation Pipeline
- *CASH* - Client only Application Session Handoff
- *CP* - Client Proxy
- *DCC* - Device Control Channel
- *DES* - Data Encryption Standard
- *ICI* - Imash Client Interface
- *ICL* - Imash Client Layer
- *IV* - Initialization Vector
- *FASH* - Full Application Session Handoff
- *GUI* - Graphical User Interface

- *MASH* - Middleware only Application Session Handoff
- *MCC* - Mux Control Channel
- *MH* - Message Handler
- *MW* - Middleware
- *MWS* - Middleware Server
- *MWCC* - Middleware Control Channel
- *MWCM* - Middleware Connector Module
- *PH* - Protocol Handler
- *SCC* - Session Control Channel
- *SD* - Session Database
- *SDC* - Session Data Channel
- *SCT* - Session Control Thread
- *SP* - Server Proxy

REFERENCES

- [BBC03] Rajive Bagrodia, Shantanu Bhattacharyya, Fred Cheng, Steven Gerding, Glenn Glazer, Richard Guy, Zhengrong Ji, Jinsong Lin, Thomas Phan, Erik Skow, Maneesh Varshney, and George Zorpas. “iMASH: Interactive Mobile Application Session Handoff.” In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, p. xxx, May 2003.
- [BPG02] Rajive Bagrodia, Thomas Phan, and Richard Guy. “A Scalable, Distributed Middleware Service Architecture to Support Mobile Internet Applications.” *Wireless Networks, The Journal of Mobile Communication, Computation, and Information (WINET)*, 2002.
- [LGG02] Jinsong Lin, Glenn Glazer, Richard Guy, and Rajive Bagrodia. “Fast Asynchronous Streaming Handoff.” In *IDMS/PROMS 2002 Joint International Workshop on Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems*, 2002.
- [PGG01] Thomas Phan, Richard Guy, Jing Gu, and Rajive Bagrodia. “A New TWIST on Mobile Computing: Two-Way Interactive Session Transfer.” In *Proceedings of the IEEE Workshop on Internet Applications (WIAPP 2001)*. IEEE, June 2001.
- [Pha02] Thomas Phan. *Utilising Application Session Handoff to Support the Pervasive Computing Vision in the Nascent Millennium*. PhD thesis, University of California, Los Angeles, May 2002.
- [PXG01] Thomas Phan, Kaixin Xu, Richard Guy, and Rajive Bagrodia. “Hand-off of Application Sessions Across Time and Space.” In *Proceedings of the 2001 IEEE International Conference on Communications (ICC 2001)*. IEEE, July 2001.
- [PZB02a] Thomas Phan, George Zorpas, and Rajive Bagrodia. “The Convergence of Heterogeneous Internet-connected Clients within iMASH.” *IEEE Wireless Communications Magazine*, 2002.
- [PZB02b] Thomas Phan, George Zorpas, and Rajive Bagrodia. “An Extensible and Scalable Content Adaptation Pipeline Architecture to Support Heterogeneous Clients.” In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002.

- [SKP02] Erik Skow, Jiejun Kong, Thomas Phan, Fred Cheng, Richard Guy, Rajive Bagrodia, Mario Gerla, and Songwu Lu. “A Security Architecture for Application Session Handoff.” In *Proceedings of the IEEE International Conference on Communications*, 2002.
- [WAP] WAP Forum. “Wireless Transport Layer Security specification.” www1.wapforum.org/tech/documents/WAP-261-WTLS-20010406-a.pdf.