MobiGATE: A Mobile Gateway Proxy for the Active Deployment of Transport Entities

by Yongjie Zheng

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Philosophy

Department of Computing
The Hong Kong Polytechnic University
February 2005

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Yongjie Zheng

Abstract

Mobile computing environments exhibit operating conditions that differ greatly from their wired counterparts. In particular, the mobile application must be able to tolerate highly dynamic network conditions and the effects of various computing devices. This research aims to develop an adaptive middleware system that adapts data flows over wireless networks to enable overlying applications to operate effectively and optimally in wireless environments.

To achieve this goal, an adaptive middleware system, MobiGATE, has been designed to support robust and flexible composition of adaptable services, termed *streamlets* in this research study. Significantly, the principle of *separation-of-concerns* is adopted in the system to facilitate clear separation of streamlet coordination from the service computation codes. By this means, the communication codes are completely separated from streamlet objects and modeled as a new type of object *channel*. An application running in this system is then regarded as a number of streamlet instances connected by channel objects. This has resulted in the formulation of a two-layered MobiGATE execution platform that supports rapid deployment of service streamlets, while facilitating adaptive composition in reaction to changing environmental contexts.

To describe application compositions, a coordination language, MCL, is designed. The language adopts the *Multipurpose Internet Mail Extensions*, *MIME*, as the underlying type definition to represent messages and streamlet interfaces. With this type system, a fundamental type compatibility check is allowed in the composition activities. In addition, a semantic model in *Z* language is defined for MCL to analyze composition descriptions and detect possible composition errors, such as feedback loops, open circuit, and mutual exclusions. The applications running in the MobiGATE system can be analyzed based on the derived semantic model to ensure their consistency in the internal structures.

A complete design, implementation and evaluation of the system have been fulfilled successfully on a Java platform, in which common runtime operating system elements are abstracted as residing either in the coordination or in the computing sub-layers. Initial experimental results validate the flexibility of the coordination approach in promoting *separation-of-concerns* in the reconfiguration of services, while achieving low computation and delay overheads. The system has proved to be efficient and robust in adapting to dynamic wireless conditions and can be improved by some recommendation work in future.

Acknowledgements

Many thanks to my supervisor, Dr. Alvin T. S. Chan, for letting me further my master study with him, for giving me all the directions I needed to select and extend my topic, and for teaching me about research, writing, and mobile computing. It’s my great happiness to work under his guidance in the past two years.

I would also like to thank Dr. Rocky K. C. Chang and Dr. H. V. Leong. Their courses have greatly enriched my knowledge in the field of internetworking protocols and distributed computing. Special thank also goes to Mrs. Elaine Anson for helping me proofread the thesis and teaching me technical writing techniques. Others I would like to thank include Chuang Siu Nam and Steve W. K. Poon who both provided me with helpful advice in my past two years of study.

Finally, I extend the most sincere gratitude to my family. It is their encouragement and support that have enabled me to follow academic pursuits. Thank you!

Table of Contents

List of Figures

1 Introduction

This chapter highlights the design issues encompassing the mobile computing operating over a wireless environment. Based on the desire to address these issues and challenges, the motivation and general objectives of this research study are specified. The organization of the thesis is given at the end of the chapter as further guidance to the reader.

1.1 Background and Motivation

Mobile computing is a rapidly emerging technology providing the ability to compute, communicate, and collaborate anywhere and anytime. With the current deployment of wireless communication services and advances in mobile computing devices, a large and ever increasing number of mobile computers and Personal Digital Assistants (PDA) are able to exchange data and synchronize with other computing devices across wireless links. However, mobile computing environments exhibit operating conditions that differ greatly from their wired counterparts. According to [Badrinath00, Satyanarayanan95], mobile computing is characterized by four main constraints:

- *Mobile elements are resource-poor relative to static elements.*
  For a given cost and level of technology, considerations of weight, power, size and ergonomics will exact a penalty in computational resources such as processor speed, memory size, and disk capacity. While mobile elements will improve in absolute ability, they will always be resource-poor relative to static elements.

- *Mobile connectivity is highly variable in performance and reliability.*
  Some buildings may offer reliable, high-bandwidth wireless connectivity while others may offer only low-bandwidth connectivity. Outdoors, a mobile client may have to rely on a low-bandwidth wireless network with potential gaps in coverage. This is in sharp contrast to the wired counterpart, where resources are abundant and highly stable.

- *Existing mobile devices are heterogenous.*

  Cell phones, personal digital assistants, palmtop computers, digital pagers, digital cameras and portable computers all have different capabilities and different requirements. Part of the difficulty of communications in the mobile environment is not just to deliver data over challenging network conditions, but to deliver such data in formats suitable for the client devices.

- *Mobile elements rely on a finite energy source.*

  While battery technology will undoubtedly improve over time, the need to be sensitive to power consumption will not diminish. Power consumption concerns must span many levels of hardware and software to be fully effective.

Another concern is the fact that the existing network protocols that have enabled the Internet revolution are not perfectly suited to the mobile computing environment. TCP, for example, does not work well on many wireless links, and often behaves poorly over satellite links owing to long latencies. Researchers have proposed modifications to existing protocols [Bakre97, Balakrishnan95, Caceres95] to handle such problems, but the understanding of networks is insufficient to allow a design of protocols that behave well in the face of all probable network conditions. Even if such protocols could be developed, the challenge of converting the enormous installed base of today's network infrastructure would have to be addressed. The Internet is distributed, decentralized and vast, and the simple solution of complete replacement of that existing infrastructure is daunting to say the least.

However, it is important to realize that even if new protocols could be successfully deployed, problems would still remain. The real goal of adaptive systems is to provide good end-to-end service, where the end points are located in applications [Badrinath00]. No adaptive solution at the network level alone can solve the entire problem without considering the needs of applications and their users.

Therefore, for mobile applications to operate effectively and optimally, the communication-related software at the application-level must be able to adapt to those mobile constraints at runtime [Chan03, Katz94]. By this it is meant that systems must be location and situation-aware, and must take advantage of this information to dynamically configure themselves in an appropriate fashion. The challenges that must be faced span a wide range of considerations and technical expertise. These include the architecture of the communications and information service infrastructure (base stations, network protocols, servers) necessary to support mobile communications, various preferences of different applications, the correctness and consistency requirement for dynamic reconfigurations, and the collection of context information.

1.2 Research Field

Basic information regarding the two fields in this study, infrastructural proxy-based adaptation and the concept of coordination, are given below.

1.2.1 The Approach: Infrastructural Proxy Services

One way to meet the above-mentioned challenges in wireless domain is by using a proxy-based gateway approach to adaptation, in which augmented network services, placed between mobile clients and gateway servers, perform aggressive computation and storage on behalf of clients [Fox98a, McKinley03]. In such architectures, adaptable applications are built from interconnected building blocks and deployed at proxy stations. Each building block, or service entity, specializes in a specific task in processing the data flow. For example, the task could involve the scaling/dithering of images in a particular format, or conversion between specific data formats, or even suitable caching to minimize the traffic transiting across a wireless network. The development of mobile applications may extend beyond the end-host process to include the composition of service entities to adapt to variations in networks and client resources.

As shown in Figure 1-1, the infrastructural proxy system mainly consists of following two network components residing between the wireless end-points:

1) A wired-side gateway called the server proxy, commonly deployed at the edge of a wired network.
2) A peer client-side proxy called the client proxy, deployed within the mobile host (MH).

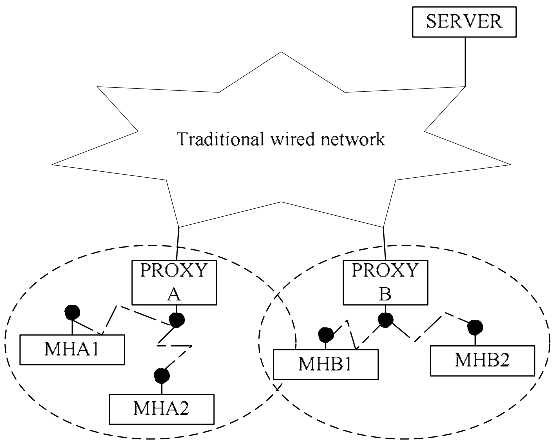


Figure 1-1: Infrastructure proxy system

The infrastructural proxy architecture supports augmented wireless network services by allowing adaptation-based service entities to be deployed at both server and client proxies to shield clients from all kinds of variances. Significantly, the architecture inherits the principle of interoperability, in which innovative and exciting services can be rapidly deployed within the existing networking environment, without causing changes to the infrastructure. The kinds of service entities that may be applied to adapt the flow of data include transformation (such as filtering, format conversion), aggregation (collecting and collating data from various sources), caching (both original and transformed content), and customization (maintenance of a per-user preferences database). Studies in this area have focused primarily on applying fixed specific service entities to the gateway proxy to introduce specific adaptation to data flowing across the wireless environment. A service entity based on image transcoding is applied to convert images on-the-fly to reduce the bandwidth requirement and have the images displayed on a display-constrained device, such as a PDA [Fox98a]. Similarly, experiments deployed based on the architecture of the gateway proxy have been conducted on text-compression, XML streaming [Chan04], and caching service entities.

A common approach to implementing the adaptation of services at the gateway proxy is a static interaction of service entities by explicitly invoking procedures on a named interface. The result is that the system integration code becomes entangled with the application-specific codes. Any replacement or modification of a service entity requires updating of not only the code for the new service entity to be integrated into the system, but also the code of those entities that have a direct relation with the old service entity. The tight coupling of service entities, in terms of the strong coordination dependency, translates into the need for manual modifications, when the transport service entities are deployed into a new environment. In a wireless network, which exhibits highly dynamic network conditions, the adaptation of service entities in the form of dynamic composition and reconfiguration is considered the norm rather than the exception.

1.2.2 Coordination: Separation of Concerns

Coordination models are a class of model recently developed to describe concurrent and distributed computations. In the area of Programming Languages, coordination is defined as the process of building programs by gluing active pieces together. A coordination model can therefore be regarded as the glue that binds separate activities into an ensemble [Malone94, Papadopoulos96]. A coordination language is the linguistic form of a coordination model. Coordination languages offer facilities for controlling the synchronization, communication, creation, and termination of computational activities.

The most prominent advantage of applying the coordination theory is that there is a complete separation of coordination from computational concerns. This separation is usually achieved by defining a new coordination language to describe the architecture of the composition. In particular, the coordination system generally consists of two kinds of processes: computation and coordination. Computational processes are treated as black boxes, while processes communicate with their environment by means of clearly defined interfaces, usually referred to as *input* or *output* ports. Producer-consumer relationships are formed by setting up channel connections between the producer output ports and the consumer input ports.

Consider the following simple example of a concurrent application where the two active entities (i.e., processes) *p* and *q* must cooperate by exchanging messages in their computations. The source code for this concurrent application looks like the code presented in Figure 1-2. The most notable point in the code is that it simultaneously gives both a description of the computation by *p* and *q*, and a description of their cooperation. The communication concerns are mixed and interspersed with computation. Thus, in the final source code of the application, no isolated piece of code can be considered as the realization of the cooperation model and reused in other applications.

```
process p:                        process q:

compute m1                        receive m1
send m1 to q                      let z be the sender of m1
compute m2                        receive m2
send m2 to q                      compute m using m1 and m2
do other things                   send m to z
receive m
do other computation using m
```

Figure 1-2: A traditional cooperation example

Let us reconsider the above example, and see how it can be implemented in the coordination model. This time the code consists of three processes: revised *p*, revised *q*, and a coordinator process *c* that is responsible for facilitating the communication of *p* and *q*. The source code for this version of the application looks something like the code presented as follows:

process p:	process q:	process c:
compute m1	read m1 from input port i1	...
wire m1 to output port o1	read m2 from input port i2	create the channel p.o1$\rightarrow$ q.i1
compute m2	compute m using m1 and m2	create the channel p.o2$\rightarrow$ q.i2
write m2 to output port o2	write m to output port o1	create the channel q.o1$\rightarrow$ p.i1
do other things		...
read m from input port i1		
do other computation using m		

Figure 1-3: The application example in the coordination model

In this example, the pattern of cooperation between the processes *p* and *q* is simple and static. Communication concerns are moved out of *p* and *q* and into *c*. However, the processes *p* and *q* are now oblivious of the source of their input, or the destination of their output. They know nothing about the pattern of cooperation in this application; they can just as easily be incorporated in any other application, and will do their job provided that they receive the right input at the right time. The process *c*, in turn, knows nothing about the details of the tasks performed by *p* and *q*. Its only concern is to ensure that they are created and connected correctly.

From the above example, it can be seen that removing the communication concerns from the computational processes enhances the modularity and the re-usability of the resulting software. The coordinator processes are generic and reusable as they know nothing about and have absolved nothing of the tasks performed by the processes they coordinate and are therefore unimpeded in these processes.

1.3 Objective

The main objective of this study is to develop an adaptive software system, which adapts data flows over dynamic wireless network conditions and various mobile devices.

To achieve this objective, the coordination theory is used in the design of a middleware system *MobiGATE*, to support the service composition and system reconfiguration at infrastructural proxies of the wireless domain. This middleware is expected to be context-aware, reconfigurable, robust, and most of all, efficient in processing incoming data flows. Specific principles on the design of this middleware are given in Section 3.1.

Concurrent with the above is the syntax and semantic definition of a coordination language *MCL* to describe the composition of applications running in the middleware system. In addition to all of those common properties described in Section 1.2.2 that are shared by existing coordination languages, this newly designed language possesses its own type system, the function of compatibility check in the composition activities, and the ability of conducting correctness verifications of language descriptions.

1.4 Organization of the Thesis

Chapter 1 introduces the background and motivation of this study. Based on the background described, the objective of this study is identified.

Chapter 2 describes the work related to this study. Several typical adaptation systems: *TranSend*, *Odyssey*, and *RAPIDware*, and a comparison with the MobiGATE system are described. Some well-known coordination languages are introduced and compared with MCL along some important dimensions, such as coordination unit, computational language, application domain. The specific characteristics of the MobiGATE system and its advantages over other similar works are highlighted.

Chapter 3 is devoted to the architecture of the MobiGATE system. Specific design principles and the working paradigm of the whole system are introduced in this chapter. The internal structure of the architecture is described from the server side to client side, with the emphasis on the function of those important components.

Chapters 4 and 5 focus on the coordination language MCL. In particular, Chapter 4 describes syntax designs of the language, ranging from its MIME type system, language elements definitions, to some specific refinement issues. As a coordination language, MCL is designed to provide the abstraction of service interfaces and the types of data associated with the messages, and checking compatibility in the composition activities. In addition, it can support the concept of recursive composition and streamlet sharing, which also differentiates MCL from other coordination languages. Chapter 5 introduces MCL's semantic model that is defined in the *Z* language. This semantic design is very important to conduct extensive analysis of MCL descriptions. This is not possible using syntax design alone.

Chapters 6 and 7 present the specific development work of the MobiGATE system and results of a series of experimental studies, which demonstrate the feasibility and validate the benefits of MobiGATE in providing adaptive mobile computing. Two operations that are most possible to bring overhead into the system are measured independently. A complete end-to-end application that fully exercises the system components of MobiGATE is set up to evaluate the system performance. The purpose for doing this is to demonstrate the use of MobiGATE while verifying the insignificant overheads incurred in runtime processing compared with the performance gained in service deployment and reconfiguration.

Finally, Chapter 8 offers conclusions to this study. It also points out some directions for the future research activities on this topic. Such work is necessary to make the MobiGATE architecture more complete, secure, and robust for deployment over a wide scale wireless and mobile environment. In the short term, as further experiences are gained in using MCL, it is aimed to further refine the language to enrich its syntax to capture mis-configuration and semantic assertions even during runtime.

2 Related Work

As previously stated, this study focuses on adaptive middleware and coordination languages. This chapter first gives a brief outline of the newly designed middleware system MobiGATE and its supporting coordination language MCL. A suite of protocols specific to mobile computing: *Mobile IP*, *snoop* protocol, and *Indirect TCP*, is then introduced. Based on this introduction, several typical adaptation systems and some well-known coordination languages are reviewed. The objective of this arrangement is to enable the reader to compare MobiGATE and MCL with this related work. Finally attention is drawn to the main areas of comparison acknowledged by the writer.

Before going into details of the related work, an overview of the newly designed MobiGATE and MCL is given as follows:

- The design and development of a **Mobi**le **G**ATEway proxy for the **A**ctive deployment of **T**ransport **E**ntities, or, MobiGATE (pronounced Mobi-Gate), is introduced in this research study. MobiGATE is a mobile middleware architecture that supports the robust and flexible composition of transport entities, known as *streamlets*. The flow of data traffic is subjected to processing by a chain of streamlets. Each streamlet encapsulates a service entity that adapts the flow of traffic across the wireless network. A major goal of the MobiGATE architecture is to provide an environment, where programmers can develop new mobile applications through combining some active service entities (streamlets), while the configuration structure of the application is completely separated from the computational activities of individual streamlets. This architecture has the advantage of supporting ease of dynamic reconfiguration and the re-usability of streamlets across applications.

- A coordination language called MobiGATE Coordination Language (MCL) is designed as part of this research. The language possesses some attractive characteristics to support the composition and reconfiguration of flexible streamlets in MobiGATE. Firstly, MCL supports the capture of flow types between streamlets and allows strong type compatibility checks in the composition activities. MCL employs *Multipurpose Internet Mail Extensions* (*MIME*) [Freed96] specifications to model streamlet interfaces and message types. MIME possesses a flexible format that easily accommodates well-known message types such as text, image, video, sound, or other application-specific data. In addition, MCL supports the notion of *recursive composition*. In other words, a composition of streamlets can itself be organized as a composite streamlet. The recursive structuring of streamlet compositions can be nested to an arbitrary level to promote modularization and re-usability. The more ambitious intention is to capture the semantics of MCL using a formal specification approach based on *Z* notation [Spivey89] to enable the analysis of the composition for consistency and to infer non-trivial properties of the language.

2.1 Mobile Networking

This section describes some basic protocols designed to serve the needs of burgeoning population of mobile computer users who wish to connect to the Internet and maintain communications as they move from place to place.

2.1.1 Mobile IP

Mobile IP [Perkins98] is a proposed standard protocol that builds on the Internet Protocol by making mobility transparent to applications and higher level protocols like TCP. It extends IP by allowing the mobile computer to effectively utilize two IP addresses: a fixed home address and a care-of address that change at each new point of attachment.

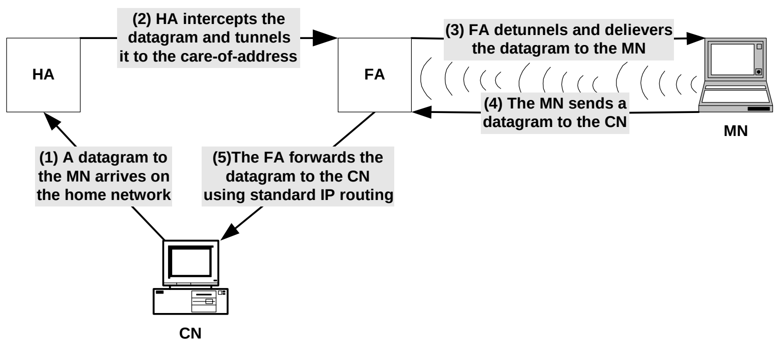

(2) HA intercepts the datagram and tunnels it to the care-of-address
HA
FA
(3) FA detunnels and delievers the datagram to the MN
(4) The MN sends a datagram to the CN
MN
(1) A datagram to the MN arrives on the home network
(5)The FA forwards the datagram to the CN using standard IP routing
CN

Figure 2-1: Mobile IP datagram flow

Mobile IP can be thought of as three major subsystems. First, a discovery mechanism is defined so that mobile computers can determine their new attachment points (new IP addresses) as they move from place to place within the Internet. Second, once the mobile computer knows the IP address at its new attachment point, it registers with an agent representing it at its home network. Lastly, mobile IP defines simple mechanisms to deliver datagrams to the mobile node when it is away from its home network.

The following gives a rough operation outline of the mobile IP protocol, making use of the above-mentioned operations. Figure 2-1 may be used to help envisage the roles played by the entities.

(1) Mobility agents make themselves known by sending agent advertisement messages. A newly arrived mobile node may optionally solicit an agent advertisement message.
(2) After receiving an agent advertisement, a mobile node determines whether it is on its home network or a foreign network. A mobile node works basically like any other node on its home network when it is at home.
(3) When a mobile node moves away from its home network, it obtains a care-of address on the foreign network, for instance, by soliciting or listening for agent advertisements, or contacting Dynamic Host Configuration Protocol (DHCP) or Point-to-Point Protocol (PPP).
(4) While away from home, the mobile node registers each new care-of address with its home agent (HA), possibly by way of a foreign agent (FA).
(5) Datagrams sent to the mobile node's home address are intercepted by its home agent, tunneled by its home agent to the care-of address, received at the tunnel endpoint (at either a foreign agent or the mobile node itself), and finally delivered to the mobile node.
(6) In the reverse direction, datagrams sent by the mobile node are generally delivered to their destination using standard IP routing mechanisms, not necessarily passing through the home agent.

2.1.2 Wireless TCP – Snoop

TCP is a reliable transport protocol tuned to perform well in traditional networks made up of links with low bit-error rates. Networks with higher bit-error rates, such as those with wireless links and mobile hosts, violate many of the assumptions made by TCP, causing degraded end-to-end performance. The snoop [Balakrishnan95] is a simple protocol that improves TCP performance in wireless networks. The snoop modifies network-layer software mainly at a base station and preserves end-to-end TCP semantics. The main idea of the protocol is to cache packets at the base station and perform local retransmissions across the wireless link.

The snoop protocol introduces a module, called the snoop agent, at the base station. The agent monitors every packet that passes through the TCP connection in both directions and maintains a cache of TCP segments sent across the link that have not yet been acknowledged by the receiver. A packet loss is detected by the arrival of a small number of duplicate acknowledgments from the receiver or by a local timeout. The snoop agent retransmits the lost packet if cached, and suppresses the duplicate acknowledgments. In the classification of the protocols, the snoop protocol is a link-layer protocol that takes advantage of the knowledge of the higher-layer transport protocol (TCP).

The main advantage of this approach is that it suppresses duplicate acknowledgments for TCP segments lost and retransmitted locally, thereby avoiding unnecessary fast retransmissions and congestion control invocations by the sender. Like other link-layer solutions, the snoop approach could also suffer from not being able to completely shield the sender from wireless losses [Balakrishnan97].

2.1.3 Wireless TCP – Indirect TCP

Indirect TCP [Bakre97] is a split-connection protocol that uses standard TCP for its connection over the wireless link. It splits each TCP connection between a sender and receiver into two separate connections at the base station - one TCP connection between the sender and the base station, and the other between the base station and the receiver. Like other split-connection proposals, Indirect TCP attempts to separate loss recovery over the wireless link from that across the wireline network, thereby shielding the original TCP sender from the wireless link.

The basic idea behind the indirect protocol model is as follows: whenever an interaction between two hosts on the internetwork, such as between a mobile host and a stationary host, involves communication over two drastically different kinds of media (i.e., wireless and wired), the protocol splits such an interaction into two separate interactions—one for each kind of communication medium. An indirect transport layer interaction between a Mobile Host (MH) and a Fixed Host (FH) consists of a fixed network protocol (i.e., TCP) used for communication between the FH and the Mobility Support Router (MSR); and a wireless protocol (i.e., wireless TCP) for communication between the MH and the MSR. The highest protocol layer at which indirection occurs is determined by the MH application—an indirect transport layer can be used in conjunction with end-to-end session and presentation layer. On the other hand, if presentation requirements are different over wireless and wired links, then an indirect presentation layer protocol can be used. Furthermore, application layer proxies running on MSRs that support MH applications are examples of application layer indirection.

Notice that even though the indirect model replaces an interaction between a mobile host (MH) and a fixed host (FH) with one interaction between the MH and its MSR and another between the MSR and the FH, the FH does not see the MSR as its communicating peer. It sees the MH itself as its actual peer host. The MSR fakes an image of the MH which is used to communicate with the fixed hosts. This image is handed over to a new MSR in case the MH engaged in an indirect interaction switches cells.

2.2 Adaptive Middleware in Mobile Computing

Middleware is necessary for distributed systems. It provides an abstract interface that gives an application developer a uniform view of low-level operating systems and networks. In the traditional systems, middleware is a means for gluing together application components that comply with certain interoperability requirements. However, in mobile computing one consequence of mobility is that the environment in which an application performs may also be changing dynamically. For example, different fault tolerance and security properties may be enforced in different execution environments. The mobile environment also introduces other complications: such as heterogeneity in the communicating devices. This has been discussed deeply in Section 1.1. As a consequence, in a wireless environment, middleware must be sufficiently flexible to enable adaptation to changes in the underlying operating systems and networks, as well as to changes in application requirements.

One general class of solutions to solving this problem is to allow various forms of network traffic adaptation. Such solutions allow hardware or software to alter the protocols or the data content being transmitted, to provide a better quality of service to users. Data flows over networks can be usefully adapted in many ways [Badrinath00]:

- The underlying protocol can be altered to handle difficult conditions. The Berkeley snoop protocol improves TCP over high error rate links [Balakrishnan95]; an adaptation mechanism can automatically initiate the snoop protocol and establish the necessary links to alleviate the poor traffic conditions over the wireless network. [Allman97].

- The data can be altered in a lossless way. Various systems allow data compression or encryption across links with poor connectivity, without any application involvement.

- Lossy adaptations can be used to obtain better compression of data over limited links by dropping inessential portions of the information, or sending a low-fidelity version. For example in TranSend, performance improvement by an order of magnitude is achieved through the effective application of lossy compression [Fox98a].

- Data can be automatically converted to formats better suited to the end systems or the intermediate networks. For example, the Top Gun Wingman browser [Fox98b] converts Web images into 2-bit grayscale bitmap displays before sending them to

Palm Pilots. This configuration has the effect of significantly reducing the bandwidth requirements, while adapting the images to better map to the small display size of handheld devices.

2.2.1 UC Berkeley TranSend

UC Berkeley's TranSend Web accelerator proxy [Fox96] was one of the earliest projects to explore adaptation proxies aggressively. TranSend intercepts HTTP requests from standard Web clients and applies data type specific lossy compression, when possible; for example, images can be scaled down or down sampled in the frequency domain, long HTML pages can be broken up into a series of short pages. TranSend's primary goal was to provide network adaptation for users of slow links.

TranSend supports a wireless vertical handoff mechanism. When a client equipped with multiple wireless interfaces switches between wireless networks, the client side vertical handoff software (which is completely independent of TranSend) generates a notification packet containing some essential characteristics (e.g., estimated expected throughput) of the new network. This packet is sent to a special UDP port on TranSend where the notification is processed and stored in a per-client profile. TranSend then processes future requests from that client in accordance with the new network type; for example, aggressive image down sampling is performed for clients connecting with an expected throughput of 15–25 Kb/s, whereas compression is much less aggressive (and in some cases disabled) for Wave LAN clients connecting at about 1 Mb/s.

The main problem with TranSend is that it cannot support peer-to-peer, collaborative services. Supporting such services is clearly important; doing so will allow direct support of peer-to-peer systems. Ways to reinforce the MobiGATE system with this important function are being investigated.

2.2.2 CMU Odyssey

Odyssey is a system built at Carnegie Mellon University to support challenging network applications on portable computers [Noble97]. Odyssey particularly focuses on resource management for multiple applications running on the same machine. Odyssey was designed primarily to run in wireless environments characterized by changing and frequently limited bandwidths, but the model was found to be sufficient in handling many other kinds of challenging resource management issues, such as battery power or cache space. The goal of the system is to provide all applications on the portable machine with the best quality of service consistent with available resources and the needs of other applications.

Odyssey is an application-aware approach to adaptation intended primarily to assist client/server interactions. The Odyssey system consists of a viceroy, an operating system entity in charge of managing the limited resources for multiple processes, a set of data type-specific wardens that handle the intercommunications between clients and servers, and applications that negotiate with Odyssey to receive the best level of service available. Applications request from Odyssey the resources they need, specifying the window of tolerance required for the desired operation. If resources within that window are currently available, the request is granted and the client application is connected to its server through the appropriate warden for the data type to be transmitted. Wardens can handle issues like caching or pre-fetching in manners specific to their data type, to make best use of the available resource. If resources within the requested window are not available, the application is then notified and can request a lower window of tolerance and corresponding level of service. As conditions change and previously satisfied requests can no longer be met (or, more happily, conditions improve dramatically), the viceroy uses upcalls, registered by the applications, to notify these applications that they must operate in a different window of tolerance, subsequently possibly causing them to alter their behavior.

One interesting aspect of Odyssey with regard to the adaptation framework is that much of the adaptation in this model is, in fact, done by the applications, which interact with Odyssey. For example, Odyssey itself does not decide that color video frames should be converted to black-and-white, but rather instructs the application that some action is required. The application itself decides how adaptation should occur, and typically instructs the server to make the adjustment. This aspect highlights a big difference between Odyssey and the MobiGATE system that completely shields the applications from the adaptation work.

2.2.3 MSU RAPIDware

The MSU RAPIDware [McKinley03] project addresses the design and implementation of middleware services for dynamic, heterogeneous environments. A major goal of the RAPIDware project is to develop adaptive mechanisms and programming abstractions that enable middleware frameworks to execute in an autonomous manner, instantiating and reconfiguring components at runtime, in response to the changing needs of client systems.

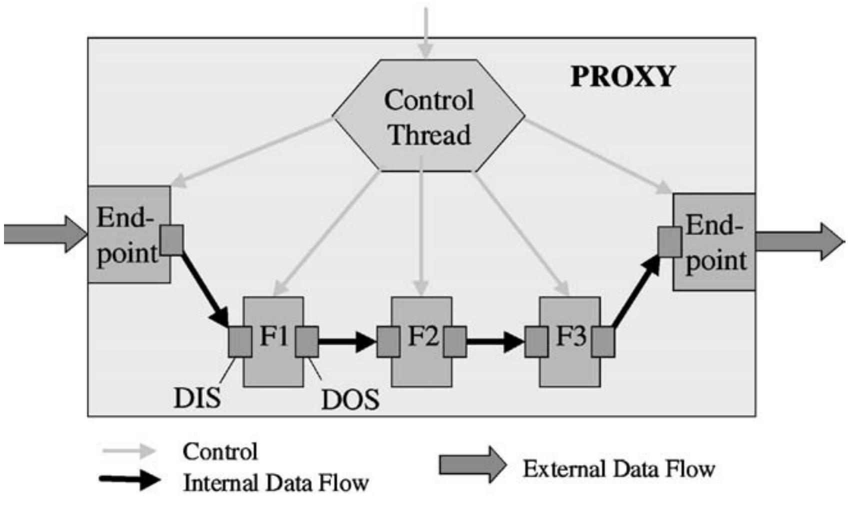


Figure 2-2: RAPIDware proxy

Figure 2-2 depicts an example of RAPIDware proxy and its configuration for processing a single data stream. The proxy receives and transmits the stream on *EndPoint* objects, which encapsulate the actual network connections. Each *EndPoint* has an associated thread that reads or writes data on the network, depending on the configuration of the *EndPoint*. A *ControlThread* object is responsible for managing the insertion, removal, and ordering of the filters associated with the stream. In this example, the proxy is comprised of three filters, *F1*, *F2*, and *F3*. The key support mechanisms are detachable stream objects, namely, *DetachableInputStream* (DIS) and *DetachableOoutputStream*

(DOS). The DIS and DOS are used for all communication among filters and between filters and *EndPoints*. DIS and DOS can then be stopped, disconnected, and reconnected, enabling the dynamic redirection and modification of data streams. The I/O stream abstraction provides a convenient way to separate adaptive behavior from the application and other parts of the middleware.

The RAPIDware system is similar to the MobiGATE system in many ways, such as the concept of filters and streams, the function of *ControlThread*, and the communication through a special object. However, owing to the definition of its DIS and DOS objects, RAPIDware can only support the linear composition of filters. Furthermore, it cannot check the "composability" of proxylets. Supporting the branch composition and consistency checks are two important advantages of MobiGATE over RAPIDware.

2.2.4 Comparison

Table 2-1 offers a comparison of the MobiGATE system and the above-introduced adaptive systems. The notable points are shown below:

- *Application Awareness* in these adaptive middleware systems can be application aware and application transparent, depending on whether the application is informed that adaptation is occurring and perhaps expected to provide an application-level response, or the system attempts to completely shield the application from this fact.

- *Adaptation Range* is the collection of applications supported by the system. Some systems provide general machinery to support a collection of unrelated applications, while others probably only support a specific application or narrowly-defined class of applications.

- *Adaptation Location* describes where the adaptation machinery resides. It can be in the client, in the server, in one or more intermediate proxies, or all of these.

- *Adaptation Compositions* refers to the possibility of composing adaptations in the adaptation machinery. In other words, it points out whether the adaptation can occur at multiple levels.

- *Mechanism* is the primary technology used in the adaptation. As far as the MobiGATE system is concerned, the separation of concerns (coordination theory) is the unique principle adopted in the design of the middleware system.

- *Description* is a general summary of the middleware system.

The comparative features below are discussed further in Chapter 3.

	TranSend	**Odyssey**	**RAPIDware**	**MobiGATE**
Application Awareness	Application transparent	Application aware	Application transparent	Application transparent
Adaptation Range	Application-specific	Application specific	General	General
Adaptation Location	Proxy	Client & Server	Proxy & Client	Proxy & Client
Adaptation Compositions	Partial	No	Partial	Yes
Mechanism	Data-type specific distillation	Resource management	Detachable stream objects	Separation of concerns
Description	Web acceleration through datatype-specific lossy compression	Application-aware adaptation by multiple applications using diverse data types	Web-based collaboration in heterogeneous wireless environments	Applying coordination theory in the service composition and system reconfigurations

Table 2-1: A comparison of adaptive systems

2.3 Coordination Models and Coordination Languages

With recent advances in the coordination theory, a number of coordination languages have become available, such as *PCL* [Sommerville96], *Conic* [Magee89], *Durra* [Barbacci93], and *Manifold* [Arbab96]. As introduced in Section 1.2.2, these languages share many common characteristics. In particular, the coordination system generally consists of two kinds of processes: computation and coordination. Computational processes are treated as black boxes, while processes communicate with their environment by means of clearly defined interfaces, usually referred to as *input* or *output* ports. Producer-consumer relationships are formed by setting up channel connections between the producer output ports and the consumer input ports.

While these existing coordination languages support primitive constructs to enable a connection to be established between coordinated processes in the form of a high-level architectural description, they lack the linguistic support to capture the input and output types associated with the ports. As a result, interconnected processes must be manually established to ensure compatibility of type when messages are exchanged between the respective input and output ports. However, the computing architecture that requires the coordination of process to be dynamically composed and reconfigured at runtime requires the intrinsic support of typed messages, which allow the programmer to capture the intended compatibility between input-output ports, and to exercise runtime safety checks.

The following subsections describe these existing languages that are designed to address the issue of coordination and architectural descriptions.

2.3.1 Proteus Configuration Language

Proteus Configuration Language (PCL) [Sommerville96] is a language designed to model the architecture of multiple versions of computer-based systems. Coordination in PCL is understood as a configuration; the unit of configuration is a family entity, representing a set of versions of a logical component or system. A family entity has various kinds of associated information, namely a classification section, an attribute section, an interface section, a parts section, a physical section specifying the entity name implementing the entity, and a relationship section that sets out the relationships between PCL entities.

Another major element of the configuration paradigm is the ports used to represent either provided or required service. A component may have a number of required and/or provided ports. Inter-component communication is facilitated indirectly by transmitting messages through bindings, where a binding is used to connect two ports. Communication can either be synchronous or asynchronous. In addition, port connections are effectively unlimited buffers. If component replacement is to take place, any outstanding messages not yet delivered to a to-be-replaced component are retained by the run-time system and eventually forwarded to the component's replacement.

Finally, PCL supports a clear distinction between the configuration component (namely PCL) and what is being configured (i.e., computational components written in any conventional programming language). Furthermore, components are context independent since inter-component interaction and communication is achieved only by means of indirect interfaces comprising ports connected by means of bindings. Thus, a separation is achieved between the functional description of individual component behaviors and a global view of the formed system as a set of processes with interconnections.

2.3.2 Conic

Conic [Magee89] is another language where coordination is viewed as configuration. A key idea in Conic is the concept of logical node. A logical node is the system configuration unit comprising sets of tasks that execute concurrently within a shared address space. Configured systems are constructed as sets of interconnected logical nodes; these sets are referred to as groups.

The programming subcomponent of Conic is based on the notion of task module types, which are self-contained, sequential tasks; these are used at run-time by the Conic system to generate respective module instances, which exchange messages and perform various activities. The modules' interface is defined in terms of strongly typed ports. An *exitport* denotes the interface at which message transactions can be initiated and provides a local name and type holder in place of the destination name and type. An *entryport* denotes the interface at which message transactions can be received and provides a local name and type holder in place of the source name and type. A link between an *exitport* and an *entryport* is realized by means of invoking the message passing facilities of the programming subcomponent. The system supports both unidirectional asynchronous and bi-directional synchronous communication. Since all references are to local objects, there is no direct naming of other modules or communication entities. Thus each programming module is oblivious to its environment, which renders it highly reusable, simplifies reconfiguration, and clearly separates the activities related to the latter from purely programming concerns.

Conic supports a limited form of dynamic reconfiguration. First of all, the set of tasks and group types from which a logical node type is constructed is fixed at node compile time. The number of task and group instances within a node is fixed at the time a node is created. Dynamic changes to link set-ups can be achieved by explicitly invoking a configuration manager through the unlink command. Another limitation of the dynamic reconfiguration functionality of Conic is related to the very nature of the links that are being established between *entryports* and *exitports*. In particular, these links are not viewed as (unbounded) buffer areas. Thus, when a link is severed between a pair of ports, the module instances involved in communication must stop exchanging messages, otherwise information may be lost and inconsistent states may result. Finally in Conic a user is constrained by using a single programming language (the Pascal like Conic programming subcomponent).

2.3.3 Durra

Durra [Barbacci93] is yet another architecture configuration language. A Durra application consists of a set of components (application tasks and communication channels) and a set of configurations specifying how the components are interrelated. Tasks are active components that initiate all message-passing operations, and channels are passive components that wait for and react to requests from the tasks. These tasks and channel implementations are linked to run-time support packages and configuration tables generated by the Durra compiler to form executable programs called clusters. The runtime support portion of a cluster is called the cluster manager, which is responsible for starting and terminating application processes and links, for passing messages between components, for monitoring reconfiguration conditions, and for carrying out reconfigurations.

The basic building blocks of Durra are the task description, which specifies the properties of an associated subprogram or subsystem, and channel description, which specifies the properties of a package implementing a communication facility. An application can be described by a compound description that contains components, structure, and reconfiguration sections.

The main concern of Durra is how to coordinate resources, such as load and execute programs, route data, and reconfigure application. As with all the other members in this family of coordination languages, it makes a clear distinction between application structure and behavior. Tasks implement the functionality of the application, whereas channels implement communication facilities. Thus it is tailored more to support rapid prototyping of distributed heterogeneous applications and test different configuration strategies, rather than as a means to actually implement these applications. Unrestricted dynamic creation of task instances is not possible.

2.3.4 Manifold

Manifold [Arbab96] is one of the latest developments in the evolution of control-driven or process-oriented coordination languages. As is the case in most of the other members of this family, Manifold coordinators are clearly distinguished from computational processes that can be written in any conventional programming language augmented with some communication primitives. Manifolds (Manifold coordinators) communicate by means of input/output ports, connected by means of streams. Evolution of a Manifold coordination topology is event-driven based on state transitions. More pertinently, a Manifold coordinator process is at any moment in time in a certain state where typically it has set up a network of coordinated processes communicating by sending and/or receiving data via stream connections established between respective input/output ports. Upon observing the raising of some event, the process in question breaks off the stream connections and evolves to some other predefined state, where a different network of coordinated processes is set up. Note that, unlike the case with other coordination languages featuring events, Manifold events are not parameterized and cannot be used to carry data — they are used purely for triggering state changes and causing the evolution of the coordinated apparatus.

One important advantage of Manifold is its support of recursive composition. This means that any coordinator can also be used as a higher-level or meta-coordinator, to build a sophisticated hierarchy of coordination protocols. Such higher-level coordinators are not possible in most other coordination languages and models. However, Manifold does not support type compatibility check, which translates to the inability to perform automatic checking for type compatibility and operation consistency in the event of adaptation and reconfiguration.

2.3.5 Comparison

	PCL	**Conic**	**Durra**	**Manifold**	**MCL**
Coordination Unit	Family entities	Logical nodes	Components	Processes	Streamlets
Computational Language	Conventional language	Pascal-like language	Ada	C, Fortran	Language Independent
Message Passing	Synchronous Asynchronous	Synchronous Asynchronous	Synchronous Asynchronous	Asynchronous	Synchronous Asynchronous
Dynamic Reconfiguration	Partial	Partial	Yes	Yes	Yes
Compatibility Checking	No	Partial	No	No	Yes
Recursive Composition	No	No	No	Yes	Yes
Formalization	No	No	No	No	Z-notation Formalization
Application Domain	Model system versions	A typical configuration language	Application prototyping	Component based development	Wireless proxy services composition

Table 2-2: A comparison of coordination languages

Table 2-2 offers a comparison of existing coordination languages and MCL along eight dimensions: *Coordination Unit* is the basic unit in terms of which the configuration is performed; *Computational Language* provides the name of the languages supported by the coordination language to program individual computational entities; *Message Passing* in these coordination models can be synchronous, asynchronous or both, depending on the underlying communication channels; *Dynamic Reconfiguration* describes the ability to dynamically change the composition structure and to create/destroy coordinated object instances at runtime; *Compatibility Checking* and *Recursive Composition* are as described above; *Formalization* is the ability to formalize the language by developing a semantic model; *Application Domain* refers to the application of languages in a domain for which it is designed. Discussions on the above comparative features are given in Chapters 4 and 5.

3 Introduction to MobiGATE Architecture

This chapter focuses on the design of the MobiGATE framework. It introduces basic design principles and the working paradigm of the whole system. Based on this introduction, an overview of the MobiGATE server and client is given with the emphasis on its internal structure.

As stated in Section 1.3, the main objective of the MobiGATE system is to adapt data flows over dynamic wireless network conditions and various mobile devices in the application level. Strictly following this goal, the design principles and whole working paradigms are introduced in following sections.

3.1 MobiGATE Design Principles

As an adaptive middleware in the mobile computing environment, the MobiGATE system is expected to be context-aware, reconfigurable, robust, and efficient in processing incoming data flows. The concept of separation of concerns forms the underlying and unifying principle in the provision of adaptive composition of services. This is regarded as one of the important contributions of this study. The core design principles of the MobiGATE system are summarized as follows:

- Firstly, the MobiGATE system should be context-aware. In other words, the system must possess the ability to collect contextual information, such as network bandwidth, transmission error rate, and client resources, and to adjust its own behavior appropriately. The principle of context-awareness fundamentally facilitates streamlets and streams to react adaptively to the operating conditions of the surroundings. One popular solution [Chan03, Fox96, Noble97] for this is to employ an entity called `Event Manager` responsible for receiving environment messages that will alter behavior of the system. These messages can originate from local operating system services and remote clients. The MobiGATE system extends this mechanism by allowing applications to choose and subscribe the context messages of interest, while filtering away those which are not necessary.

- In addition, the newly designed system must be reconfigurable. In this context, reconfigurable means that the composition structure of applications running in the middleware system can be changed dynamically in response to different conditions. More ambitiously, the system should ideally support the dynamic reconfiguration of each service entity bound to associated applications. For example, the behavior of a service entity may be changed or adapted by altering its meta-representation at runtime.

- Significantly, MobiGATE is a middleware supporting the separation of concerns, advocated by the coordination theory. Firstly, the system possesses the ability of composing adaptation services. Secondly, the communication codes are completely separated from those computational activities in the composition of adaptations. Each service entity should be completely independent of its running environment. The main difficulty, in this respect, lies in the abstraction of environmental dependencies from those service entities while at the same time maintaining an acceptable performance.

- In contrast to some existing adaptive middleware, such as TranSend introduced in Section 2.2, the MobiGATE system is expected to support peer-to-peer, collaborative adaptation services. To achieve this goal, MobiGATE needs a client-side system to reversely process data flows from the server for the purpose of adaptation, such as decompression and decryption. Because of the constrained resources and power of most mobile devices, this MobiGATE client system must follow a thin-client model, which means there cannot be as much workload as on the server side.

- As far as performance is concerned, the system should be efficient in processing data flows. With the increase in the number of running applications and mobile clients, an acceptable performance should still be obtained. It is also important to note that this system must be robust and maintain a relatively stable throughput most of the time. The aim is for all of these performance requirements to be satisfied with the development of several related technologies, such as *carrier resource* and *instance pooling*, which are introduced in following sections.

3.2 MobiGATE Working Paradigms

The MobiGATE system consists of two parts: MobiGATE server and MobiGATE client. The MobiGATE server, where adaptations of data flows are composed, resides in the intermediate proxy between the data sender and receiver. The MobiGATE client, in most cases, stands in the position of data receiver, responsible for processing received messages reversely.

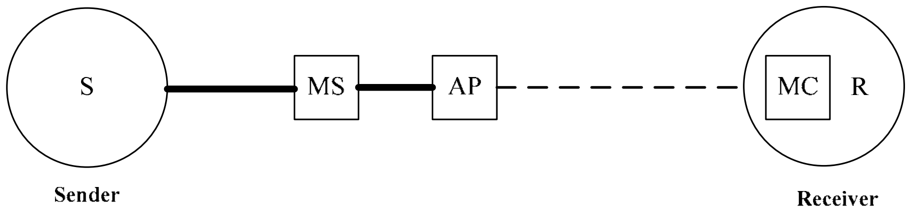


Figure 3-1: The working paradigm of MobiGATE system

Figure 3-1 shows a simple data flow with a single sender (S) and receiver (R). The data flows across various links and nodes in the network. The thick line represents the wired network and the dashed line suggests the wireless part. Access Point (labeled AP in the figure) is located at the edge of the wired network to support communications between the fixed sender and its mobile receiver. At some point in the network, the MobiGATE Server (MS) imposes various adaptation services on the data flow, which is then processed reversely by the MobiGATE Client (MC) at the receiver side.

To some extent, Figure 3-1 is a simplification of real world. It shows a simple data flow and it does not illustrate problems, such as delivery deadlines or security concerns, nor does it suggest the level of complexity possible in even a single network flow. But the figure captures the root of the problem. A stream of data flows from a source to a destination across a network, using links of different conditions. Altering the data flows in various ways could lead to better overall results, in terms of lowering bandwidth requirements, alleviating error condition, encoding secured data, generic compression, and transcoding. The aim of the MobiGATE system is not to provide specific services or configuration of services, but rather to provide a general platform to facilitate ease of deployment of services across the wireless links by providing core mechanisms and system services.

It is important to note that the MobiGATE server may reside in mobile nodes, while the MobiGATE client is placed at proxies in the wired network. This situation, upstream transmission (client-to-server*)*, happens when the data sender is a mobile device, while the receiver is a fixed node in the wired network. However, there is an inherent asymmetry in the wireless communications: the bandwidth in the downstream direction (server-to-client) is much greater than that in the upstream direction (client-to-server). For this reason, more and more mobile-aware applications have now adopted the push-based (downstream direction) data dissemination model [Barbara99]. In this thesis, the MobiGATE system primarily focuses on solving problems in downstream direction communications. As discussed, the architecture is sufficiently flexible to be used to address upstream communications as well.

3.3 MobiGATE Server

There exists in MobiGATE a clear distinction between the activities of coordination and computation. Figure 3-2 shows the architecture of MobiGATE server, which is organized into two executing planes. The `Streamlet Execution Plane` is responsible for scheduling streamlet instances for computation, while the `Stream Coordination Plane` is responsible for maintaining the interaction and relationship between the coordinated streamlets. The `Coordination Manager` maintains a configuration table for each instance of streamlet composition. The configuration table serves to contain meta-information on the composition of streamlets, message type constraints, port connections, and routing constraints. The table is derived from the compilation of the MCL script, which the `Coordination Manager` uses to control the stub generation and the channel objects and to facilitate the exchange of messages among the streamlets. In short, the coordination plane can be viewed as a routing plane, where coordination activities and interaction are abstracted from the streamlet codes. This leads to a highly reconfigurable system where interconnections and relationships between service entities can be composed dynamically in a non-intrusive way.

On another plane, the `Streamlet Manager` controls the execution of instances of a streamlet. During the setup process, the manager is required to locate the classes of streamlets and allocate necessary computational resources for execution. The `Event Manager` is responsible for generating system events in reaction to different conditions. Finally, there is a `Streamlet Directory`, where the streamlet providers can advertise their services. This directory provides code-level implementations of streamlets at runtime. Below, various components of the MobiGATE server architecture are described in detail.

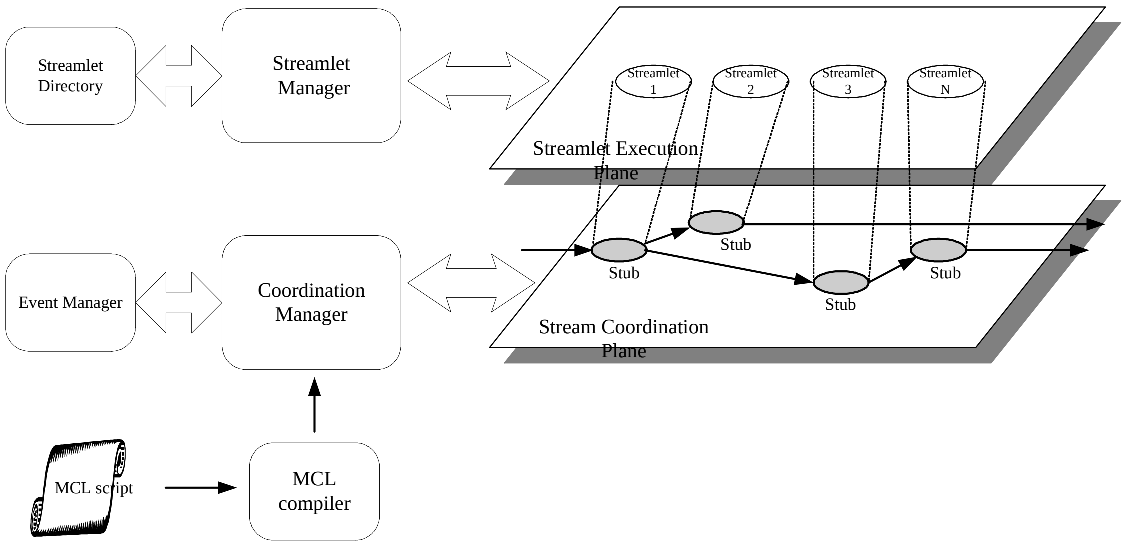


Figure 3-2: Architecture of MobiGATE server

3.3.1 Coordination Manager

The `Coordination Manager` controls the generation of stubs and channel objects and facilitates the message exchange among the streamlets. It maintains a configuration table for each running coordination stream, defining the specific message flow route in these streams. From the perspective of networking, the role of the `Coordination`

Manager is somewhat similar to that of a router, while the configuration table acts as the routing table. Another important function of the Coordination Manager is to filter events from the Event Manager and to broadcast them among coordination streams. This may invoke dynamic reconfiguration actions.

3.3.2 Stream Coordination Plane

The Stream Coordination Plane is the layer where coordination activities take place. In this plane, a stream object is modeled as streamlet stubs connected by channels, with the composition structure defined by the configuration table held by the Coordination Manager. Stubs do not contain any service logic. Instead, they implement whatever operations are necessary to forward requests to streamlet instances and receive results. The exchange of data among the stubs is currently done through channels. The channels transport data by using a frequently used method, *carrier resource*, where a *repository* or *carrier resource*, accessible to both producer and user stubs, is created. Producer stubs write the data to the shared carrier. User stubs read the data from the shared carrier. The carrier resources can be written only after they have been read by consumers.

3.3.3 Streamlet Manager

The Streamlet Manager manages the execution of various streamlets. It intercepts service requests from the Stream Coordination Plane, passes the incoming message to the corresponding streamlet instance for processing, and finally returns the result message. If the requested streamlet has not yet been initiated, the manager creates an instance for it from the Streamlet Directory; otherwise the manager directly delivers the message to the Streamlet Execution Plane.

3.3.4 Streamlet Execution Plane

All the computation activities take place in the Streamlet Execution Plane. In this plane, individual streamlets run independent of others and focus on imposing services on the incoming messages. Two kinds of streamlets, *Stateless* and *Stateful*, are distinguished depending on whether state information is to be kept for the requesting coordinator processes.

One of the fundamental benefits of using the MobiGATE architecture is that it is able to handle a heavy workload while maintaining a high level of performance. There is a relationship between the number of streams and the number of streamlets that are required to service them. As the stream population increases, that is, as the number of applications increases, the number of streamlets required increases correspondingly. At some time, the increase in the number of streamlets will have an impact on performance and diminish the throughput. MobiGATE explicitly supports a mechanism called *streamlet pooling* that makes it easier to manage large numbers of streamlets in the `Streamlet Execution Plane`.

The concept of pooling resources is not new. A commonly used technique is to pool database connections so that the business objects in the system can share access to the database. This mechanism reduces the number of database connections that are needed, which, in turn reduces the consumption of resources and increases throughput. The MobiGATE `Streamlet Execution Plane` also applies resource pooling to streamlets; this technique is called *streamlet pooling*. Streamlet pooling reduces the number of streamlet instances, and therefore, the resources needed to service requests from the `Stream Coordination Plane`. It is also less expensive to reuse pooled streamlet instances than to frequently create and destroy instances.

Streamlet pooling is applicable to streamlets that are considered *Stateless*. In other words, since *Stateless* streamlets are never associated with a specific stream, there is no fundamental reason to keep a separate copy of each streamlet for each stream instance. Thus, the system can keep a much smaller number of streamlets, reusing each streamlet instance to service the different requests. By this means the resources actually needed to service all the requests are greatly reduced.

3.3.5 Event Manager

The `Event Manager` is responsible for generating system events in reaction to different conditions. These events may be caused by client requests, changes to the system environment, or by exceptions in streamlet executions. Coordinating the publication of events is fundamental to the realization of adaptive processing in a mobile middleware system, such as MobiGATE.

3.3.6 MCL Complier

The `MCL Compiler` controls the compilation of the MCL coordination script and generates the necessary configuration tables to define the message flow routes in coordination streams. It is also responsible for any compile-time validation work such as compatibility checks. Incompatible connections in the script are returned by the compiler with a detailed error message.

3.3.7 Streamlet Directory

The `Streamlet Directory` serves as the repository where streamlet providers can advertise their services. In addition, it serves as a central storage for streamlet codes in which the `Streamlet Manager` may locate the relevant streamlets and create instances for execution. Note that it is possible for a streamlet itself to be represented as an MCL coordination script. This defines a recursive composition of other native streamlets.

3.4 MobiGATE Client

Figure 3-3 depicts the operational flow and architecture of the MobiGATE client. In contrast to the server, the MobiGATE client system has no concept of channel or coordination. All the composition information is already recorded in the incoming message header. The system at the client side needs simply to read the message header and distribute the message to corresponding client streamlets for reverse processing. The resultant messages are then sent to higher layered applications. This asymmetry mechanism has greatly liberated MobiGATE client systems from heavy coordination logic, and translates into a much lower consumption of computing resources and energy on the client side. The details of the comprising components are given below.

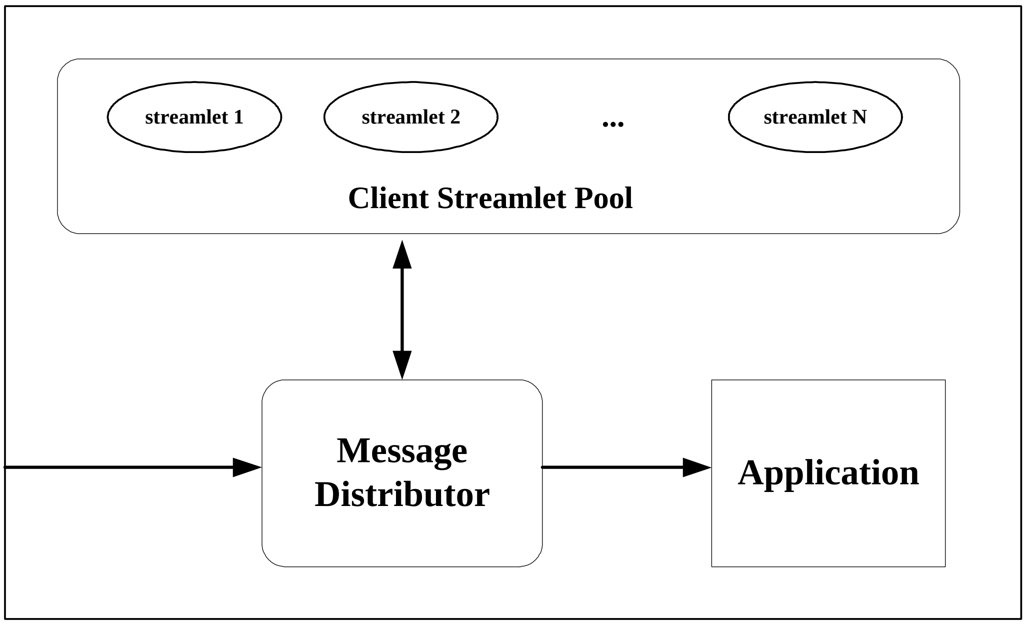


Figure 3-3: Architecture of MobiGATE client

3.4.1 Message Distributor

The main task of the `Message Distributor` is to parse the incoming MIME messages and distribute them to each corresponding client streamlet for reverse processing. An important characteristic of the `Message Distributor` is that it can support multiple threads at runtime. This is similar to the characteristics of the *servlet* in the J2EE architecture. Whenever a new message arrives, the system tries to find an available `Message Distributor` thread to parse the message. If this fails, the system creates a new thread to service the incoming message.

3.4.2 Client Streamlet Pool

The function of the `Client Streamlet Pool` is quite similar to that of the `Streamlet Directory` at the server side. The difference is that here the system maintains peer streamlets, instead of original streamlets maintained at the server side. In addition, the `Client Streamlet Pool` is also responsible for creating and destroying client streamlet instances to service the incoming messages forwarded by the `Message Distributor`.

4 MobiGATE Coordination Language

This chapter describes the MobiGATE Coordination Language (MCL) used to compose applications running in the MobiGATE system. The syntax design of the language is introduced in detail, including the type system, language elements, and the important design characteristics that differentiate MCL from other coordination languages. A case example using MCL to compose applications out of existing services, is also given to demonstrate the effectiveness of this newly designed language. Based on the syntax design, the formalization of the language with a semantic model is then introduced in Chapter 5.

4.1 Message and Port Typing

The type system in programming languages defines the type of data and structural representation of information to be processed. The typed information represents the characteristics of the data intended by the developer of the program and is correspondingly treated as such during compilation and execution. In MobiGATE, the typed messages exchanged between streamlets and the definition of port types is viewed as fundamental in enabling the flexible and robust composition of streamlets. Significantly, it allows the developer to concisely capture the intended message types, bound to the streamlet ports. Runtime checking, in the form of matching the message types to the streamlet ports, can be exercised to ensure consistency during operations. In this project, the adoption of the *Multipurpose Internet Mail Extensions* (*MIME*) 1.0 Internet standard is proposed as the underlying type definition, to represent messages and declarations of port type. As such, messages, exchanged in the system, are formatted based on MIME. This assumption is reasonable and valid considering the fact that MIME has evolved to become the *de facto* formatting standard for many network services, including email, news and the World Wide Web.

Figure 4-1 shows a graphical representation of the MCL type system. A fundamental property is that, each given type has multiple associated direct subtypes or supertypes. This is useful in facilitating the checking process for type compatibility of activities of which the architecture is composed. Another interesting property of the defined type system comes from the extensible nature of the MIME type media system, meaning that it is not difficult to introduce a new message type into the system.

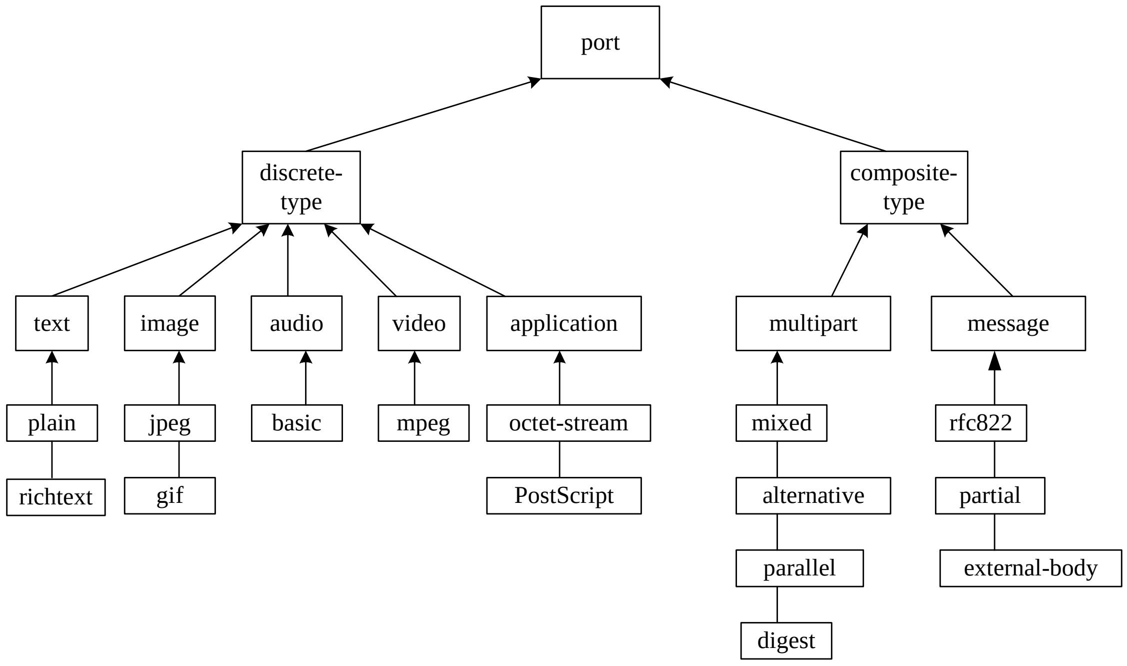


Figure 4-1: Graphical representation of a type system

Based on the MIME type system, the *Backus Normal Form* (*BNF*) notation of a type declaration in MCL can be defined as shown in Figure 4-2. Note that this definition is generated from a simplification of a standard MIME Content-Type header field definition with some modifications.

```
type-declaration ::= type “/” subtype | intermediate
                           ; Matching of media type and subtype
                           ; is ALWAYS case-insensitive
intermediate ::= “port” | discrete-type | composite-type | type
type ::= discrete-type | composite-type
discrete-type ::= “text” | “image” | “audio” | “video” | “application”
composite-type ::= “multipart” | “message”
subtype ::= <A publicly-defined extension token. Tokens of this form
                 must be registered with IANA as specified in RFC 2048>
```

Figure 4-2: BNF notation of the type declaration

4.2 MCL Language Elements

MCL is an underlying declarative language for describing dynamically changing networks of active concurrent processes. It is comprised of several important abstractions including streamlets, channels, and streams. Collectively, the abstractions, labeled constructs, constrained typing, and definitions form the building blocks for describing the composition of the streamlets and their architectural description. The important elements representing the core abstractions are described in the following sections.

4.2.1 Streamlet

Streamlets in this study represent the main functional elements of an application and work as coordination units, as listed in Table 2-2. They own a set of ports, through which they interconnect with the rest of the system. Interconnections among streamlets are explicitly represented as separate language elements, called channels. Streamlets must always connect to one another through channels. As a consequence, every streamlet port must be connected to a compatible channel port based on the definition of MIME type.

Within the context of a streamlet, ports play the role of placeholders. This means they will not be affected by the computation of the streamlet. Streamlets read/write messages from/to their associated input/output ports by using read/write primitives. They do not need to have explicit knowledge of the real source/destination of messages. The separation and externalization of the interconnections of the streamlets promote their independence and reusability. In MCL the notation *p.i* is used to refer to port *i* of a streamlet instance *p*.

```
streamlet-definition ::= “streamlet” streamlet_name description
streamlet_name ::= token
                          ;is ALWAYS case-insensitive
description ::=  “{”  ports attributes “}”
ports ::=  “port”  “{”
                                   port_declaration
                            “}”
attributes ::=  “attribute”  “{”
                                            streamlet type
                                            implementation
                                            description
                                       “}”
port_declaration ::= dir port_name “:” type-declaration “;”
dir ::=  “in” | “out”
port_name ::= token
                  ; is ALWAYS case-insensitive
streamlet type ::=  “type”  “=”  “STATELESS” | “STATEFUL”  “;”
implementation ::=  “library”  “=” value “;”
description ::=  “description”  “=” value “;”
value ::= quoted-string
token ::= *(<any (US-ASCII) CHAR except SPACE, CTLs, or tspecials>)
tspecials ::=  “(” | “)” | “<” | “>” | “@” | “,” | “;” |
                   “:” | “\” |<” > “/” | “[” | “]” | “?” | “=”
```

Figure 4-3: BNF notation of the streamlet definition

Streamlets are defined as sets of ports and attributes, which describe streamlets’ core functions and capabilities to interconnect with the rest of the system, as shown in Figure 4-3. The Establishment of the type of an input/output port is required as part of the port declaration. Notice that as identification, each streamlet may have more than one input/output port, each of which is associated with the name of a specific port. The attribute declaration describes three important properties:

- *Type*. Type indicates whether the streamlet needs to keep information on corresponding application states. Based on this attribute, streamlets are distinguished as *Stateless* or *Stateful*.

- *Library*. The library connects streamlets with code-level components that implement their intended functionality. Examples of code-level components include executable programs, and source code models.

- *Description*. Description provides some general descriptive information about streamlets.

In addition, a distinction is made between the descriptions of streamlets and their instances in MCL. In this study, a streamlet is defined as an instance and a streamlet definition is its description. Streamlets (or streamlet instances) can be created from a definition using the *new-streamlet* primitive or destroyed using the *remove-streamlet* primitive.

4.2.2 Channel

Channels describe relationships of interconnection and constraints among streamlets. Traditional programming languages do not support a distinct abstraction for representing such relationships, and implicitly encode support for component interconnections inside their abstractions for components. In contrast, all streamlet interconnections, in MCL, are explicitly represented, using channels. Channels, like streamlets, own ports. These ports must be connected to compatible streamlet ports.

```
channel-definition ::= “channel” channel_name description
channel_name ::= token
                           ;is ALWAYS case-insensitive
description ::= “{” ports attributes “}”
ports ::= “port” “{” “in” “:” type-declaration “;”
                               “out” “:” type-declaration “;”
                            “}”
attributes ::= “attribute” “{”
                                             channel_type
                                             category
                                             buffer_size
                                     “}”
channel_type ::= “type” “=” value1 “;”
category ::= “category” “=” value2 “;”
buffer_size ::= “buffer” “=” value3 “;”
value1 ::= “SYN” | “ASYN”
value2 ::= “S” | “BB” | “BK” | “KB” | “KK”
value3 ::= *(DIGIT) “Kbytes”
token ::= *(<any (US-ASCII) CHAR except SPACE, CTLs, or tspecials>)
tspecials ::= “(” | “)” | “<” | “>” | “@” | “,” | “;” |
                    “:” | “\” |<” > “/” | “[” | “]” | “?” | “=”
```

Figure 4-4: BNF notation of the channel definition

A channel represents a reliable, directed, and optionally buffered flow of information in time. Reliable means that all messages placed into a channel are guaranteed to flow through without loss, error, or duplication, with their order preserved. Directed means a channel always has two identifiable ends: an *in* and an *out*. Once a channel is established between two streamlets, it operates autonomously and transfers the message from its input to its output port. Figure 4-4 shows the formal definition of the channel. Like the streamlet, it is also defined by port declarations and certain important attributes:

- *Type*. Two channel types are distinguished: synchronous and asynchronous. Synchronous channels are zero-length buffers and can receive a value only if they can be delivered immediately, while asynchronous channels are unbounded FIFO buffers.

- *Category*. The possibility of pending units existing in a channel makes it meaningful for a channel to remain connected at one of its ends, after it is disconnected from the other. Based on this property, channels are distinguished as *S*, *BB*, *BK*, *KB*, and *KK*. The *S* channel guarantees that there are never any pending units in the channel. The *BB* (break-break) channel is automatically disconnected from the other of its streamlets, as soon as it is disconnected from one. The *BK* (break-keep) channel does not disconnect from its target streamlet when it is disconnected from its source streamlet. The *KB* (keep-break) channel simply reverses the semantics of the *BK*. The *KK* channel cannot be disconnected at either side of the connection.

- *Buffer*. The buffer size in the channel is specified in units of Kbytes. Ideally, an asynchronous channel should have an unbounded buffer, as introduced above. However, in reality, a large buffer size is generally chosen to simulate this property.

As with streamlets, there is a differentiation between channels and channel definitions in MCL. Channels (or channel instances) can be created from a definition, using the *new-channel* primitive or destroyed, using the *remove-channel* primitive.

4.2.3 Stream

A stream is purely a composition script, also known as a coordination script, running on the coordinator side. It is within a stream that different streamlet and channel instances are created, network topologies are constructed, and actions in response to different events, are specified. Streams can be viewed as streamlets connected by channels with the ability to perform adaptations. Simultaneously, a stream can also be viewed as a "streamlet" with input/output ports, which come from the stream's inner streamlet ports and are unsatisfied by any inner connections. Figure 4-5 is the formal definition of a stream object.

In addition to the primitive *new-streamlet, remove-streamlet, new-channel,* and *remove-channel* introduced above, there also are *connect*, *disconnect*, and *disconnectall* primitives to set up/break down connections in stream descriptions. For example, *connect* (*p.o*, *q.i*, *c*) is written to set up a connection between the port *o* of the streamlet *p* and the port *i* of the streamlet *q*, using the channel *c*. For simplicity *connect* (*p.o*, *q.i*) can be used instead, whereby the system automatically creates a channel instance of an asynchronous *BK* type with 100 Kbytes of buffer to connect between the ports.

```
stream-definition ::= “stream” stream_name declaration
stream_name ::= token
                            ;is ALWAYS case-insensitive
declaration ::=  “{”
                                      *(streamlet_instantiation)
                                      *(channel_instantiation)
                                      *(connection_setup)
                             “}”
streamlet_instantiation ::=  “streamlet”  str_instance “=”
                            “new-streamlet”  “(” streamlet_name “)”  “;”
channel_instantiation ::=  “channel”  chan_instance  “=”
                            “new-channel”  “(” channel_name “)”  “;”
connection_setup ::=  “connnect”  “(” port_ID “,”
                           <port_ID> “,” <chan_instance> “)”  “;”
port_ID ::= str_instance “.” port_name
```

Figure 4-5: BNF notation of the stream definition

Dynamic reconfiguration is another important task that needs to be addressed in the description of a stream. It’s also an important advantage of MCL over most existing coordination languages, as shown in Table 2-2. The interaction model in MCL is event-driven. That is, a coordinator process waits for an occurrence of a specific event to stimulate entry to a predefined state and perform some actions. These actions typically consist of setting up or breaking off connections of ports and channels. The coordinator then remains in that state until it observes the occurrence of some other related events.

System events are generated by the `Event Manager`, an important component in the MobiGATE environment that facilitates the adaptation of the streamlets. Several types of events in MCL have been predefined. Such events are introduced and described in Section 6.4. They represent external events that can be subscribed to initiate the adaptation through reconfiguration of the composition of the streamlets. The selection of the event types include *LOW_ENERGY* (Client devices running out of power), *LOW_BANDWIDTH* (Poor network bandwidth), *LOW_GRAYS* (Client devices supporting only shallow grayscale), and *END* (End of application). Note that, unlike other coordination languages featuring events, MCL events are not parameterized and cannot be used to carry data – they are used purely for triggering the evolution of the composition of coordinated streamlets in response to contextual events.

There is an important primitive in descriptions of a stream, *when (event) {…actions…}*, identifying reactions to different events. In principle, the coordinator picks up any broadcast event; in practice, however, only a subset of the potential receivers is usually relevant to an event as these receivers specify actions in the corresponding *when* sections.

4.3 Case Example of Using MCL

A pragmatic example of the composition of service entities based on MCL is presented in this section. To illustrate and highlight the robustness of the language in regulating complex adaptations in response to evolving wireless and mobile operating environments, a modified datatype-specific distillation application, which was deployed at U.C. Berkeley [Fox98a], is adopted. The service entities, in the form of streamlets, used in this example are listed below.

- *Switch*: Dividing incoming messages based on the semantic type of the data;
- *Image Down Sampling*: Lossy compression of an image by reducing the sample rate;
- *Map to 16 grays*: Reducing images to 16 grays to support shallow grayscale displays;
- *PostScript-to-Text*: Discarding some information on format and converting documents to rich-text supported by most devices;
- *Text Compressor*: A generic text compressor;
- *Merge*: Integrating different types of information into a whole body;
- *Power Saving*: A power-saving mechanism as discussed in [Anastai02].

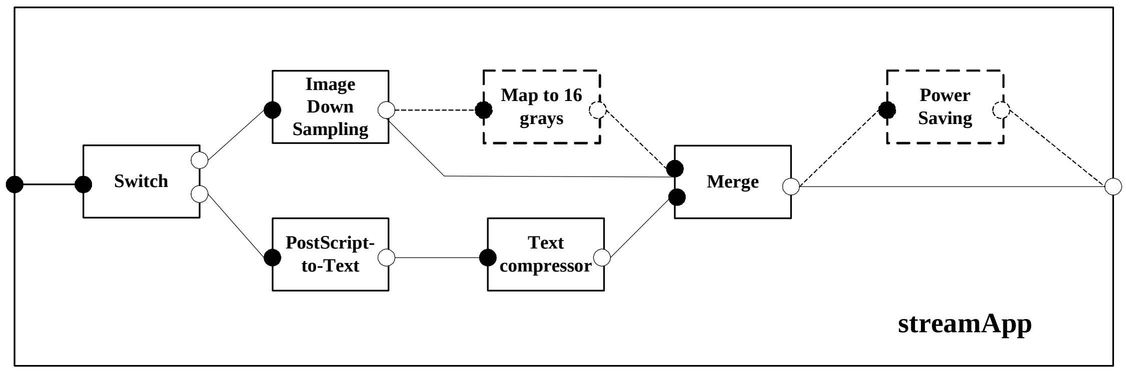


Figure 4-6: The composition model of a datatype-specific distillation application

Figure 4-6 shows the composition model of the application. The rectangle boxes represent the service entities modeled as streamlets associated with input ports (black points) and output ports (white points). Lines between different ports embody intermediate channel objects. Note that the dashed parts are optional, which means they will be included in the architecture only when certain specific events take place. For example, the power-saving entity is invoked when the system subscribes and correspondingly receives the *LOW_ENERGY* signal from the hardware abstraction driver. The abstraction of the stream application *streamApp*, which exercises recursive composition, contains the composition of the streamlets. The composite *streamApp* streamlet has its own input/output ports, derived from those internal ports, not satisfied by any internal connections. Therefore, from the outside the *streamApp* can also be regarded as a streamlet object and can be graphically represented in the form of an encapsulated box and ports to be reused in other stream applications. The concept of recursive composition is discussed in detail later in this thesis.

Below is a description of individual streamlets in MCL. Considering the large size of image data, a channel with a buffer of 1024 Kbytes is created specifically to connect image-related streamlets, while for others the default 100 Kbyte-sized channel is used.

```
streamlet switch{
   port{
      in pi     :   multipart/mixed;
      out po1 :   image;
      out po2 :   application/PostScript;
   }
   attribute{
      type = STATELESS;
      library = “/general/switch.class”;
      description =
          “Divide incoming message based on
         the semantic type of the data.”;
   }
}
streamlet img_down_sample{
   port{
      in pi     :   image;
      out po   :   image;
   }
   attribute{
      type = STATELESS;
      library = “/image/downSample.class”;
      description =
          “reduce sample rate of the image”;
   }
}
streamlet map_to_16_grays{
   port{
      in pi     : image;
      out po   : image;
   }
   attribute{
      type = STATELESS;
      library = “/image/mapGrays.class”;
      description =
          “To support clients with shallow
            grayscale displays”;
   }
}
streamlet powerSaving{
   port{
      in pi     : multipart/mixed;
      out po   : multipart/mixed;
   }
   attribute{
      type = STATEFUL;
      library = “/general/powerSaving.class”;
      description =
          “Power saving mechanism.”;
   }
}
```

```
streamlet postscript2text{
   port{
      in pi     :   application/PostScript;
      out po   :   text/richtext;
   }
   attribute{
      type = STATELESS;
      library = “/text/p2t.class”;
      description =
          “Convert PostScript material to
      richtext document.”;
   }
}
streamlet text_compress{
   port{
      in pi     :   text;
      out po   :   text;
   }
   attribute{
      type = STATELESS;
      library = “/text/Compressor.class”;
      description =
          “a generic text compressor.”;
   }
}
streamlet merge{
   port{
      in pi1   :   image;
      in pi2   :   text;
      out po   :   multipart/mixed;
   }
   attribute{
      type = STATELESS;
      library = “/general/merge.class”;
      description =
          “Merge messages together.”;
   }
}

channel   largeBufferChan{
   port{
      in     :   image;
      out   :   image;
   }
   attribute{
      type = ASYN;
      category = KB;
      buffer = 1024 Kbytes;
   }
}
```

Figure 4-7: Streamlet and channel descriptions

Based on these streamlet descriptions, the final composition script for the stream *streamApp* is written as follows:

```
stream streamApp{
    streamlet s1 = new-streamlet (switch);
    streamlet s2 = new-streamlet (img_down_sample);
    streamlet s3 = new-streamlet (map_to_16_grays);
    streamlet s4 = new-streamlet (powerSaving);
    streamlet s5 = new-streamlet (postscript2text);
    streamlet s6 = new-streamlet (text_compress);
    streamlet s7 = new-streamlet (merge);

    channel c1, c2, c3 = new channel (largeBufferChan);

    connect (s1.po1, s2,pi, c1);
    connect (s1.po2, s5,pi);
    connect (s2,po, s7.pi1,c2);
    connect (s5.po, s6.pi);
    connect (s6.po, s7.pi2);

    when(LOW_ENERGY){
        connect (s7.po, s4.pi);
    }
    when(LOW_GRAY){
        disconnect(s2.po, s7.pi1);
        connect(s2.po, s3.pi, c2);
        connect(s3.po, s7.pi1, c3);
    }
}
```

Figure 4-8: Stream description

As shown in Figure 4-8, the occurrence of *LOW_ENERGY* triggers the reconfiguration of the stream by introducing streamlet *powerSaving*. Similarly, the occurrence of *LOW_GRAY* triggers the insertion of a new streamlet *map_to_16_grays* to provide transcoding of colour images to grey scale images.

4.4 Design Issues of MCL

The MCL design is greatly influenced by a set of core design issues. These issues, in a way, differentiate MCL from existing and general coordination languages. It has specific focus as an underlying coordination language to facilitate robust composition and support for dynamic reconfiguration in a mobile and wireless environment.

4.4.1 Compatibility Check

In a manner analogous to the type checking in programming languages, it is desirable to be able to perform the limited static checking of compatibility when connecting or transforming the composition of service entities. Such controls facilitate the construction of correct and consistent architectures while helping designers focus their attention on more complex issues. MCL provides such a mechanism, based on the matching of streamlet port types.

MCL imposes several semantic restrictions and constraints on the ability of streamlets to connect to each other. The two most important restrictions are:

- Streamlet ports can only connect to channel ports (and vice versa).
- Sink ports can only connect to source ports that are equal to or are a specialization of the sink ports.

It is desirable to encode such restrictions and constraints so that a number of compatibility tests can be automatically performed by the language at the time of compilation. Since all MCL connections are between ports, it is desirable to be able to perform compatibility checks at the port level.

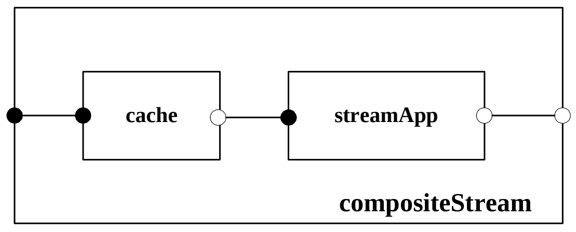
The first restriction is relatively easy to validate by language. Before establishing a connection, MCL checks the source of two ports. If both are from streamlets, or channels, the connection is considered illegal. For the second restriction, MCL bases its compatibility check on port types. As introduced above, multiple associated direct subtypes or supertypes can be assigned to a port type. These subtype/supertype relations are used to specify the second restriction on compatibility. To establish a connection, MCL performs a match of port types: if the type of source port is equal to or is a subtype of a type of sink port, the connection is considered legal. In the application shown in Figure 4-6, the connection between the *PostScript-to-Text* output port and the *Text*

Compressor input port is valid, since the source port type *text/richtext* is a subtype of the sink port type *text*.

4.4.2 Recursive Composition

As mentioned above, the stream and streamlet processes are indistinguishable, in terms of their abstraction, as boxes with associated input/output ports. Thus, a stream object can logically be regarded as a streamlet written in native MCL composition languages and reused in another stream application. This is known as *recursive composition.* In addition, the key word **main** is included to indicate the highest-level stream object in a coordination script. The system can thus start to execute an MCL application by locating a stream object that is labeled **main** in the coordination script.

To support this recursive composition, the composition of a separate description of streamlets associated with each stream object is needed. Based on these descriptions, the system instantiates instances of streamlets and sets up connections to each streamlet, just as it does for common streamlets. For example, the example stream discussed above can be reused as follows.



```
main stream compositeStream{
    streamlet s1 = new-streamlet (cache);
    streamlet s2 = new-streamlet (streamApp);

    connect (s1.po, s2.pi);
}
```

```
streamlet streamApp{
    port{
        in pi : multipart/mixed;
        out po   : multipart/mixed;
    }
    attribute{
        type = STATEFUL;
        library = “/general/streamApp” ;
        description =
            “match the stream object streamApp to
        a streamlet” ;
    }
}
```

Figure 4-9: Recursive composition

As shown in Figure 4-9, *compositeStream* is oblivious to the internal structure of the stream *streamApp*. From the view point of *compositeStream*, this stream object is just a common streamlet implemented in MCL. In a similar manner, *compositeStream* can also be reused in another higher-level stream object, as a common streamlet object.

The support of the recursive composition model corresponds to the spirit of the coordination theory in facilitating organized composition. As MobiGATE evolves and, coupled with the proliferation of streamlets, a need to provide a coordinated and structured organization of streamlets is envisaged to promote ease of use and management. This is reflected in MCL through the support of the hierarchical modeling of streamlet composition based on recursive coordination.

4.4.3 Streamlet Sharing

Another important issue of this study is the concept of *streamlet sharing*. Each streamlet is oblivious to the source or destination of the messages and is concerned only with imposing its computation on incoming messages and producing response messages. The complete decoupling of coordination from computation makes it possible to share instances of streamlets between different streams.

The question is, how can messages be distributed to their corresponding streams when the messages are generated on the output ports of the shared instances of streamlets? In other words, how can messages belonging to different stream instances be identified?

As introduced previously, streamlets, exchange messages based on MIME. In the MIME message format, a header exists called the *MIME-extension-field* for applications to define their own application-specific headers. A new field in the message header to identify messages from different streams is defined, using this feature.

```
session ::= “Content-Session” “:”session-id
```

Before executing a coordination stream, the system automatically generates a unique session ID for each instance of a stream. Subsequently, all messages belonging to this stream are labeled with the assigned session ID in their “Content-Session” field. By this means, the system can easily differentiate messages from different streams.

5 MCL Semantic Model

The definition of a formal language includes two parts: the specification of the proper construction of sentences, the syntax; the specification of the meaning of sentences, the semantics [Kolman96].

The primary focus in Chapter 4 is on formal definitions of the architecture descriptions in the syntax domain. Definitions of basic elements, such as streamlets, channels, and streams, are given. While these descriptions may provide useful documentation, the current level of informality limits their usefulness. In particular, the syntax does not capture the intrinsic semantic properties of the language, thus rendering analysis of the architecture for consistency impossible.

In this chapter, MCL is formalized by means of the development of a semantic model. The model specifies precisely all the language elements introduced previously, and is described by using the specification language *Z*. The *Z* schemas, which can be regarded as generalized type definitions, are used to represent the basic constructs. These schemas provide semantics that permit the formal verification of properties of the model. Additional details on *Z* can be found in [Spivey89].

5.1 Formalization of MCL Language Elements

It is assumed that sets [ENTITY, DATA, PORT] exist. The ENTITY identifiers represent global names. Name clashes between distinct streamlets and streams are disallowed. The set DATA includes different data types defined by MIME media type representation, as discussed in Section 4.1. The PORT members are the streamlet interfaces and are also introduced as a given set in the model.

5.1.1 Streamlet

In order to define the behavior of a streamlet, its input and output data ports and the data type that may be passed along each data port must be known. This latter information is represented by a (partial) function from data ports to their data types. In addition, a streamlet is identified with a unique id. This streamlet information is formalized in the schema *streamlet*.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Streamlet}} \\
\textit{id} : \textit{ENTITY} \\
\textit{inputs, outputs} : \mathbb{P}\ \textit{PORT} \\
\textit{port-type} : \textit{PORT} \mathrel{+\!\!\!\!\!\twoheadrightarrow} \mathbb{P}\ \textit{DATA} \\
\hline
\textit{inputs} \cap \textit{outputs} = \Phi \\
\textit{dom port-type} = \textit{inputs} \cup \textit{outputs} \\
\hline
\end{array}$$

Some enforced constraints on streamlets are

- Input and output data ports are distinct (first predicate);
- Each port is associated with a data type (second predicate).

5.1.2 Channel

The streamlet data ports are connected by channels, modeled as typed data streams. Each channel has a distinct source and sink for receiving and sending data. Recall that PORT represents an input or an output of a particular streamlet. Thus a channel represents a data transmission from one streamlet to another.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Channel}} \\
\textit{id} : \textit{ENTITY} \\
\textit{sink, source} : \textit{PORT} \\
\textit{type} : \mathbb{P}\ \textit{DATA} \\
\hline
\textit{sink} \neq \textit{source} \\
\hline
\end{array}$$

5.1.3 Stream

A stream can now be modeled as a set of streamlets connected by channels. More formally, a stream now agglomerates a set of streamlets together with a set of channels.

Stream

$$\begin{array}{l}
\textit{streamlets} : \mathrm{P}\ \textit{Streamlet} \\
\textit{channels} : \mathrm{P}\ \textit{Channel} \\
\hline
\forall\ s1, s2 : \textit{streamlets} \mid s1 \neq s2 \bullet s1.id \neq s2.id \\
\forall\ c1, c2 : \textit{channels} \mid c1 \neq c2 \bullet c1.id \neq c2.id \\
\forall\ c : \textit{channels} \bullet \exists\ s1, s2 : \textit{streamlets} \bullet \\
\qquad\quad c.source \in s1.outputs \\
\qquad \wedge\ c.sink \in s2.inputs \\
\qquad \wedge\ s1.port\text{-}type(c.source) \subseteq c.type
\end{array}$$

Because the ENTITY identifiers represent global names, name clashes between distinct streamlets and channels are disallowed. It also is important to point out that the port type of two connected streamlets must be compatible with that of the intermediate channel. In other words, the port type must be equal to, or a subtype of, that of the intermediate channel. This is specified by the last precondition in the above definition.

5.1.4 Composite Streamlet

As introduced previously, recursive composition allows streams to be considered as streamlets and reused in the composition of other high level streams. The main problem here concerns the resulting streamlet type, since the resulting streamlet should not be independent of the associated architecture. Consequently, the input and output types declared at the composite level are selected first. All the inner architecture types not concerned with any inner connection, are then added. The resulting composite streamlet is then formalized as follows.

Composite_Streamlet

$$
\begin{array}{|l}
id : ENTITY \\
inputs, outputs : \mathbb{P}\ PORT \\
port\text{-}type : PORT \pfun \mathbb{P}\ DATA \\
streamlets : \mathbb{P}\ Streamlet \\
channels : \mathbb{P}\ Channel \\
\hline
inputs = BasicIn \bigcup InnerIn \\
\forall in \in InnerIn \bullet \exists s \in streamlets \bullet in \in s.inputs \\
\qquad\qquad \wedge \forall c \in channels \bullet in \neq c.sink \\
outputs = BasicOut \bigcup InnerOut \\
\forall out \in InnerOut \bullet \exists s \in streamlets \bullet out \in s.outputs \\
\qquad\qquad \wedge \forall c \in channels \bullet out \neq c.source \\
inputs \bigcap outputs = \Phi \\
dom\ port\text{-}type = inputs \bigcup outputs \\
\hline
\end{array}
$$

The most difficult part of the formal definition concerns the definitions of the sets *InnerIn* and *InnerOut*. For simplicity, the process of selecting the unsatisfied input types are formalized as being those that are not concerned with any connection involving the inner components.

5.2 Analysis of Architectural Descriptions in MCL

Based on the semantic model defined in Section 5.1, different kinds of analysis and checking are now considered. In this section some representative examples of analysis supported by the formal framework displayed in this study are presented. To address the topological requirements, a stream configuration is considered as a directed graph in which two streamlets are connected if any of their ports are attached to a common channel, as shown below.

$$
\begin{array}{|l}
\textit{StreamGraph} \\
\hline
\textit{Stream} \\
\textit{connect} : \textit{streamlets} \leftrightarrow \textit{streamlets} \\
\hline
\textit{connect} = \\
\quad \{(s1,\ s2) : \textit{streamlets} \times \textit{streamlets} \mid \\
\qquad \exists\, c \in \textit{channels} \bullet \\
\qquad\quad c.\textit{source} \in s1.\textit{outputs} \ \wedge \\
\qquad\quad c.\textit{sink} \in s2.\textit{inputs}\ \} \\
\hline
\end{array}
$$

5.2.1 Feedback Loops Detection

An important restriction on the definition of stream configurations is that the architecture has no feedback loops, or the connection graph is *acyclic*. Informally, in terms of streamlets, this means data processed by a streamlet will never re-enter the streamlet. The *acyclic* can be defined as follows:

$$
\begin{array}{|l}
\textit{Acyclic} \\
\hline
\textit{StreamGrapah} \\
\hline
\mathrm{id}\ \textit{streamlets} \cap \textit{connect}^{+} = \Phi \\
\hline
\end{array}
$$

5.2.2 Open Circuit Detection

In addition, it is highly possible that some intermediate output ports might, by mistake, be left unconnected during the composition activities, possibly resulting in the loss of incoming messages entering the corresponding streamlet. This is called an *open circuit* problem and must be detected and avoided during stream configurations. Based on the definition of *StreamGraph*, a formal definition of the *open circuit* problem is provided as follows:

OpenCircuit

$StreamGraph$

$$\begin{array}{l} \exists s : streamlets \mid \\ \quad \forall s' \in streamlets \bullet s'.id \neq s.id \\ \quad\quad \wedge \ (s, s') \notin connect \end{array}$$

5.2.3 Mutual Exclusion Detection

It is worthy of note that in the MobiGATE system there are some streamlets that are mutually exclusive. Thus in the stream compositions, the incoming messages cannot be processed by these exclusive streamlets simultaneously. That is to say, the exclusive streamlets cannot be deployed in the same path from the START to the END of the stream configurations. This exclusion relationship can be represented as a partial function from the *streamlets* set to its power set in its formal definition, as shown below.

MutualExclusion

$StreamGrapah$

$repel : streamlets \rightarrowtail \mathbb{P}\ streamlets$

$$\begin{array}{l} \forall x \in dom(repel) \quad y \in repel(x) \bullet \\ \quad\quad (x,y)\ ,\ (y,x) \notin connect^{+} \end{array}$$

5.2.4 Dependency Verification

In contrast to mutual exclusion restrictions, in some situations when a streamlet is added to the stream configuration, a set of closely related streamlets should also be included. In other words, these streamlets are said to be mutually dependent. This assurance is a desirable enforced constraint in the stream composition. The formalization of this requirement is described as follows:

MutualDependency

StreamGrapah

$depend : streamlets \pfun \mathbb{P}\ streamlets$

$$\begin{array}{l} \forall x \in dom(depend)\ y \in depend(x) \bullet \\ \qquad (x,y) \in connect^{+} \\ \quad \lor\ (y,x) \in connect^{+} \end{array}$$

5.2.5 Preorder Verification

The deployment order of the streamlets in the composition is another composition restriction. Some streamlets are predefined to impose their services on the incoming messages in a specific order. For example, encryption and compression are two independent service entities, and generally the encryption must be deployed before the compression entity. If it is not so, it is necessary for the system to be able to detect this order error. This *PreOrder* restriction is defined below.

PreOrder

StreamGrapah

$preorder : streamlets \leftrightarrow streamlets$

$$preorder \cap (connect^{+})^{\sim} = \Phi$$

One virtue of the semantic model defined above is that it has proven to be an excellent way to obtain an in-depth understanding of MCL, and may even result in discovering MCL features that were not apparent from a textual description. With these formal definitions of system properties, many existing tools for *Z* notation can be utilized to automate the analysis process, such as *Z/EVES* [Saaltink97]. This is analogous to the use of type checking to guarantee that all uses of procedures are consistent with their programming language definitions.

5.3 Case Example of Analyzing MCL Descriptions

In this section, a simple example is used to demonstrate the usefulness of the MCL's semantic model in verifying the correctness of system compositions. Figure 5-1 shows a composition example comprising three streamlets: *s1*, *s2*, and *s3*. A feedback loop is part of this composition architecture. As discussed in Section 5.2.1, this loop must be detected and avoided in the definition of stream configurations. How the MCL semantic model is used to find and correct this composition mistake is shown below.

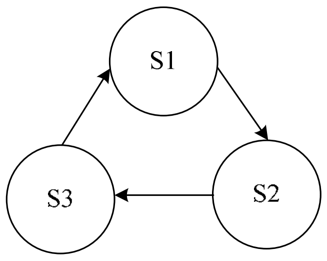


Figure 5-1: The composition example with a feedback loop

In MCL, the above composition model can be simply described as follows:

connect (s1.out, s2.in);
connect (s2.out, s3.in);
connect (s3.out, s1.in);

Based on the definitions of *StreamGraph*, the above description can be mapped into the semantic domain as follows:

$s1,s2,s3 \in streamlets \wedge (s1,s2) \in connect \wedge (s2,s3) \in connect \wedge (s3,s1) \in connect$

As $connect^{+}$ stands for the strongest or smallest transitive relation containing *connect* [Spivey89], it is not difficult to deduce the following statement from the above:

$(s1,s1), (s2,s2), (s3,s3) \in connect^{+}$

But : *(s1,s1), (s2,s2), (s3,s3)* $\in$ *id streamlets*

Thus: *id streamlets* $\bigcap$ *connect* $^{+} \neq \Phi$

Obviously this conflicts with the definition of the *acyclic* requirement for a stream description. The conflict is indicated in the validation process, a process, which for other system properties, is quite similar.

The above example shows that a semantic model can be used to analyze application architecture to ensure that it is consistent in its internal structure. This is not possible using syntax descriptions alone. It has been found that a large amount of effort is involved in validating a given application configuration. In contrast, the semantic model defined in this chapter, has made the correctness verification of MCL descriptions feasible and much easier.

The derived MobiGATE semantic model has proved to be effective in providing an in-depth understanding of MCL and given an insight into the complexity of configuration semantics. Of importance in this respect is that the MCL composer's intended meaning of streamlet and channel descriptions and composition semantics can be captured precisely. As a result, the overall MCL description can be validated to ensure that potential conflicts, such as open circuit and mutual exclusion introduced in the previous section, are resolved at compilation time and also during runtime. A more ambitious aim is to develop a complete theory of architecture description that allows reasoning about the behavior of the system as a whole.

6 Development of the MobiGATE System

This chapter describes the design and development of the MobiGATE system that supports the necessary framework for streamlets in order that they may be easily composed, inserted and removed. This system forms the underlying runtime execution environment where streamlets are deployed and executed on the proxies residing between the two ends of the wireless link. The MobiGATE runtime model is implemented on a Java platform, in which common runtime operating system elements are abstracted as either residing in the coordination or computing sub-layers. Significantly, the runtime system is designed to promote maximum reusability of system services while minimizing overheads that may be incurred due to streamlet operations. The aim is to provide a general and flexible system that supports rapid development and deployment of streamlet applications without dictating how the streamlet operation flows.

The low-level details of the implementation codes are not discussed here. Rather, the chapter highlights three major abstract classes that are pervasive in the MobiGATE model.

- *Streamlet* base class is the core abstraction of a streamlet that implements and manages the lifecycle operations associated with a streamlet object, such as pause, activate, and end.

- *MessageQueue* abstracts the communication among all streamlets residing in MobiGATE. Importantly, it provides a convenient way to separate the communication parts from the computation codes in a streamlet application.

- *Stream* base class is responsible for managing the insertion, removal, and replacement of streamlets that are composed within a stream.

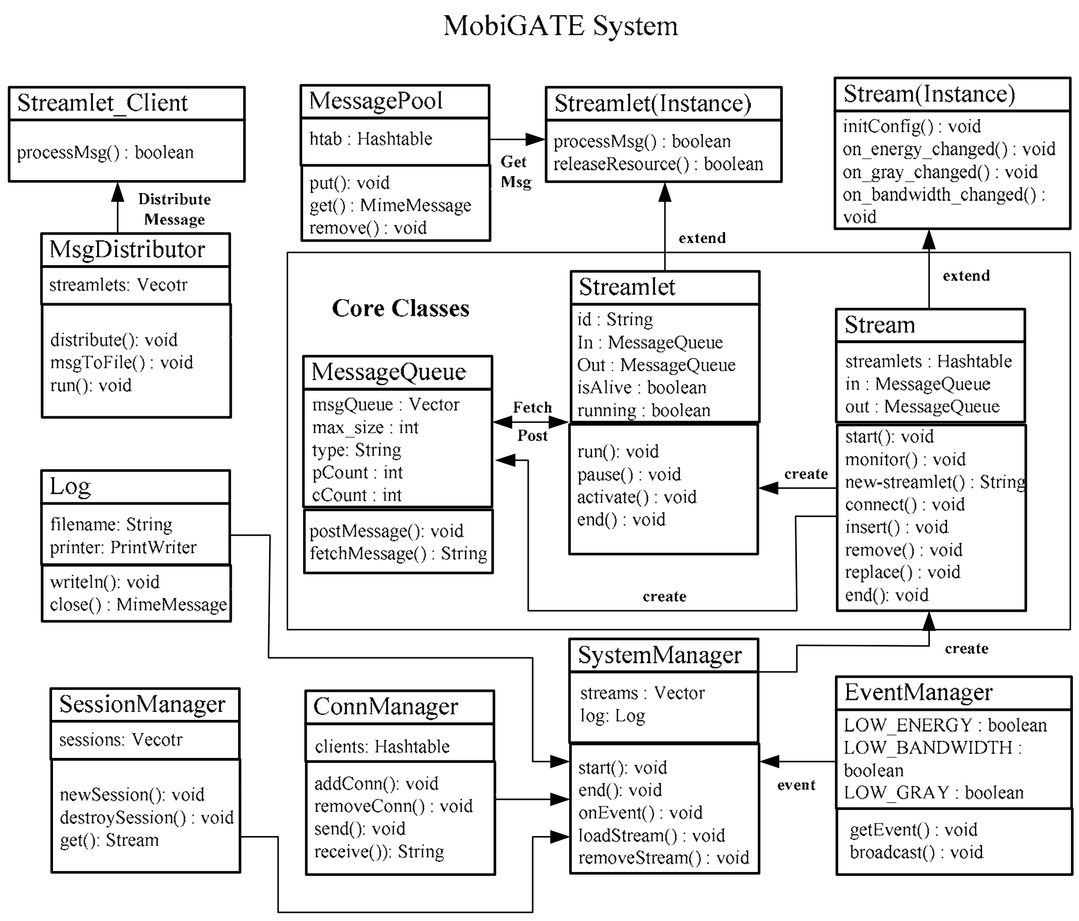


Figure 6-1: The MobiGATE system class diagram

Figure 6-1 shows the greatly simplified but representative class diagram of the complete implementation. The following sections briefly describe the main classes that make up the MobiGATE infrastructure.

6.1 The Base Class: *Streamlet*

An excerpt of the *Streamlet* base class is shown in Figure 6-2. Any streamlet that is to be deployed within the MobiGATE infrastructure needs to extend this base class. The *Streamlet* class extends the *Thread* class and thus is inherently runnable. The author of a specific streamlet is required to write the functional code within the `processMsg`() method, which will be invoked by the `run`() method in the *Streamlet* class. The

Streamlet class contains an *In* and an *Out* object, along with their corresponding standard references to manipulate the stream connections. A group of methods (e.g. `setIn`, `setOut`, `getIn`, `getOut`) is used to establish reference to the *In* and *Out* objects in the *Streamlet* code itself. In addition, several lifecycle methods are also defined in the *Streamlet* class, such as `pause`(), `activate`(), and `end`(), to manage streamlets lifecycle operations during runtime.

The computing model can be used to define general types of streamlets by providing the developer with the flexibility to include any application-specific processing by overriding the streamlet's `processMsg`() method. For example, streamlets can be rapidly developed to provide important services such as image down sampling, color to gray conversion, compression, and encryption. Connection between streamlets in the *Streamlet* instance is achieved through the use of the *In* and *Out* object abstractions.

```
public class Streamlet extends Thread implements Serializable, Cloneable {

    //The Streamlet identifier
    private String id;

    //The input port
    public MessageQueue In;

    //The output ports
    public MessageQueue Out;

    //The setIn and setOut methods allow on to set up their own
    // references names to the actual message queue
    public void setIn(MessageQueue input){
        In = input ;
        input.incr_cCount();
    }
    public void setOut(MessageQueue output){
        Out = output;
        output.incr_pCount();
    }

    // specific processing logic goes here
    // waiting to be override by developers
    public void processMsg(MimeMessage msg){}

    //Life cycle methods
    public void pause() { ... }
    public void activate() { ... }
    public void end() { … }
}
```

Figure 6-2: Excerpt from the class *Streamlet*

6.2 The Base Class: *MessageQueue*

MessageQueue is used to manage the communications among streamlets on a given stream. In the class, there is a message vector *msgQueue*, accessible to the producer and consumer streamlets and holding references to the passing MIME messages. The main concern with the vector is how to synchronize producer and consumer activities. The class implements methods `postMessage()` and `fetchMessage()` and obtains the synchronization in two ways. First, the two threads must not simultaneously access the *msgQueue*. A Java thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread blocks until the object is unlocked. Second, the producer must have some way to indicate to the consumer that the message is ready and the consumer must have some way to indicate that the value has been retrieved. The *Thread* class provides a collection of methods--*wait*, *notify*, and *notifyAll*--to help threads wait for a condition and notify other threads of when that condition changes.

In particular, two important integer-typed attributes have been included in the class *MessageQueue*, producer count *pCount* and consumer count *cCount*, which respectively represent the number of producers and consumers attached to a queue object. By increasing the corresponding *pCount* by 1, the system assumes that a producer streamlet has been connected to the channel. If the value of *pCount* is 0, the system assumes that the channel does not at the moment have a producer attached. For the variable *cCount,* the representation is similar. The code segment below is excerpted from the *MessaeQueue* class.

```
public class MessageQueue{

    //The Message Vector
    private Vector msgQueue;
    private int max_size = MAX_SIZE;
    private String type = "*/*";

    //Poducer/Consumer Count
    private int pCount = 0;
    private int cCount = 0;

    //The method to insert messages
    public synchronized void postMessage(String msgID){ … }

    //The method to read&remove messages
    public synchronized String fetchMessage(){ … }

}
```

Figure 6-3: Excerpt from the class *MessageQueue*

6.3 The Base Class: *Stream*

The *Stream* class is the base class that serves to manage stream applications in the MobiGATE infrastructure. Unlike the *Streamlet* class, *Stream* is responsible for managing the stream of composed streamlets. Its concern is not the operations of the streamlets, but how the streamlets are composed and their interactions with one another. The three primary tasks of the *Stream* class are initializing connection setup, reconfiguration of the system in response to different events, and definition of composition primitives. The initializing connection setup method provides an opportunity for developers to allocate and initialize stream specific parameters in preparation for the stream to be deployed. To support the reconfiguration setup, an important method `onEvent()` is abstracted to allow developers to override and react to external contextual events. The composition primitives are fundamental to the *Stream* class in that they provide method calls to support dynamic streamlet compositions. In particular, the class implements methods for inserting and removing streamlets from the stream, as well as methods for creating new streamlet instances in the stream. All these defined primitives are used in the composition of specific stream applications. Figure 6-4 is excerpted from the class *Stream*.

```
public class Stream implements Serializable, Cloneable{

    // member streamlets hash table
    protected Hashtable htab;

    // life cycle methods
    public void start(){ … }
    public void end() { … }

    // environment monitor
    public void monitor() { … }

    // initial configuration setup, to be override by stream developer
    public void initConfig(){
    }

    // reconfiguration activities, to be override by stream developer
    protected void onEvent(ContextEvent evt){}

    // composition primitives definition
    protected String new_streamlet(String name){ … }
    protected void connect(String p, String c, MessageQueue channel){ … }
    protected void insert(String p, String c, String i){ … }
    protected void remove(String t, String p){ … }
    protected void replace(String old, String alt){ … }

}
```

Figure 6-4: Excerpt from the class *Stream*

In expressing a stream for an application, the developer is required to capture in MCL the streamlets' composition, which essentially captures the initial connection topology and reconfiguration schemes. On deploying the stream application within the MobiGATE infrastructure, the system automatically creates the corresponding stream instances from these descriptions by extending the base class *Stream* and overriding the related methods (e.g. `initConfig`(), `onEvent(ContextEvent evt)`, where *evt* represents the contextual event). Importantly, the composition model greatly relieves programmers of complex and low-level streamlet programming and system activities, such as event listening or resource recollection. In short, the clear separation of concerns in terms of the computation and composition enhances the modularity and flexibility of the system, while facilitating ease of service reconfiguration through dynamic stream composition.

6.4 MobiGATE Event System

The generation and propagation of system events is another important issue that needs to be considered especially in the design of the MobiGATE system. Today's Internet clients vary widely with respect to both hardware and software properties: screen size, color depth, effective bandwidth, processing power, and the ability to handle different data formats. To build a dynamically adaptable system, the various client variations must be captured and modeled into a standard and recognizable form, before some further actions are taken to respond to them.

The MobiGATE event system, where each client variation is modeled as an object called *MobiGATE Event*, has been designed for this purpose. In the system, all the client variations have been classified into four different categories: *System Command*, *Network Variation*, *Hardware Variation*, and *Software Variation*, each of which represents one axis along which clients may vary. It is necessary to point out that each category may have more than one event defined. For example, there are three events *PAUSE, RESUME,* and *END* in the *System Command* category. The category and its corresponding event list are shown in Table 6-1.

Category	**EventID**	**Description**
System Command	PAUSE	Pause the stream application
	RESUME	Resume the paused application
	END	End the whole application
Network Variation	LOW_BANDWIDTH	The effective bandwidth < 100Kb/s
	NORMAL_BANDWIDTH	The effective bandwidth ≥ 100Kb/s
	HIGH_ERROR	High error rate
	NORMAL_ERROR	Normal error rate
	LONG_DEALY	The transmission delay ≥ 1s
	NORMAL_DEALY	The transmission delay < 1s
Hardware Variation	LOW_GRAY	The shallow grayscale display
	LOW_ENERGY	The client is under low energy mode
	NORMAL_ENERGY	The client is under normal energy mode
Software Variation	JPEG_ONLY	The client device only supports Jpeg image
	GREY_ONLY	The client does not support colored display
	PS_TO_TEXT	The client does not support PostScript

Table 6-1: MobiGATE event definition

Note that, unlike the case with other systems featuring events, MobiGATE events are not parameterized and cannot be used to carry data – they are used purely for triggering the evolution of the coordinated streamlets. As shown in Figure 6-5, each MobiGATE event object is associated with three primary attributes:

- *eventID*: The identity of the event object.
- *categoryID*: The category the event object belongs to.
- *evtSource*: The source of the event. In other words, which stream application does the event object belong to?

```
public class ContextEvent{

    private String eventID;
    private int categoryID;
    private String evtSource;

    ContextEvent(String evtID, int cgID, String  src){
        eventID = evtID;
        categoryID = cgID;
        evtSource = src;
    }

    public String getEventID(){
        return eventID;
    }

    public int getCategoryID(){
        return categoryID;
    }

    public String getSource(){
        return evtSource;
    }

}
```

Figure 6-5: MobiGATE event object

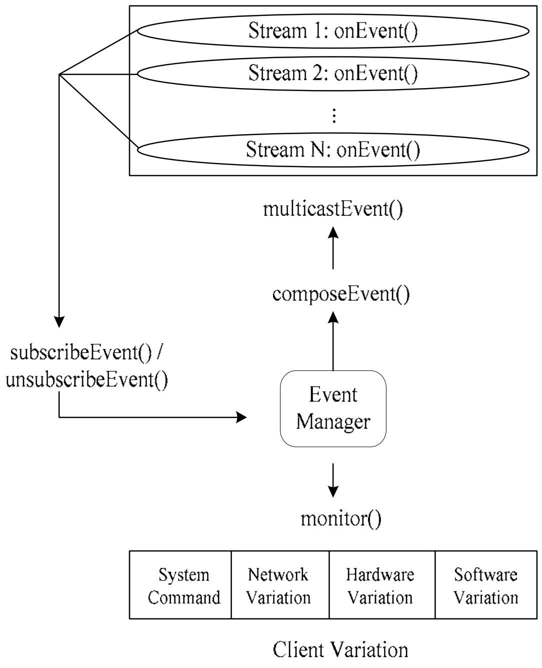


Figure 6-6: MobiGATE event system

Figure 6-6 shows the MobiGATE event system diagram. A kernel entity, called `Event Manager`, was designed to control the operation of the event system including event subscription, triggering, and monitoring. The `Event Manager` monitors the underlying client variations and composes corresponding events in response to various situations. Simultaneously, the `Event Manager` multicasts events among different stream applications, whose method `onEvent()` will then be invoked upon the reception of these events.

To avoid overheads incurred in processing the flood of events, individual stream applications may subscribe to events of interest and react to these events by performing appropriate reconfiguration, while ignoring those events that they consider superfluous. To support this function, the `Event Manager` maintains an array *subscriberList* to hold subscribers for different event categories. Each element of this array is vector-typed, which holds a collection of subscribers of the corresponding event category. The *EventManager* class is equipped with the method `subscribeEvent()` for stream applications to register events of their interests. Figure 6-7 is excerpted from the class *EventManager*.

```
public class EventManager extends Thread{

    //maintain subscribers of different categories
    private Vector[] subscriberList;

    void EventManager(){
        categoryList = new Vector[EventUtility.CategoryCount];
        for(int i=0;i<EventUtility.CategoryCount;i++){
            categoryList[i] = new Vector();
        }
    }

    public void subscribeEvt(int categoryID, Stream app){
        categoryList[categoryID].add(app);
    }

    public void unsubscribeEvt(int categoryID, Stream app){
        categoryList[categoryID].remove(app);
    }

    public void multicastEvent(ContextEvent evt){
        try{
            int id = evt.getCategoryID();
            for (Enumeration e = categoryList[id].elements() ; e.hasMoreElements() ;){
                ((Stream)e.nextElement()).onEvent(evt);
            }
        } catch (Exception e){}
    }

    public void run(){
        //Monitor the underlying resources
        //Compose new event objects in response to various situations
    }

}
```

Figure 6-7: Excerpt from the class *EventManager*

Thus when a new event object is generated, the `Event Manager` is required to check the attribute *evtSource* of the event and verify whether the corresponding stream application has subscribed to the event category. If they have, the event is forwarded to the stream for processing; otherwise it is ignored.

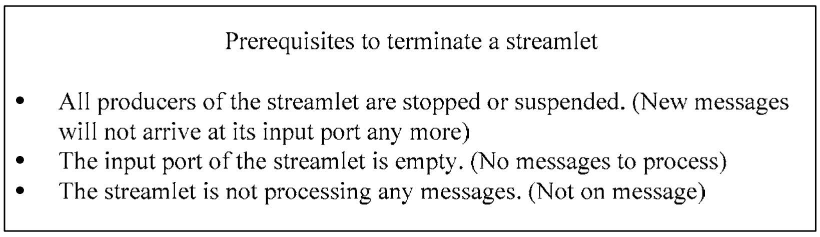
6.5 Sender and Receiver Streamlet Matching

Each streamlet, if necessary, has associated with it a unique *peerID*, which is used to identify a peer streamlet on the other side of the communication channel. Given a streamlet that performs some processing on an outgoing message, its peer streamlet performs the reverse processing on incoming messages. For example, a text *Compressor* streamlet on the sending end of a connection requires a *DeCompressor* streamlet on the receiving end. Each streamlet on the sending side of a connection adds a header field to the messages before writing them to its output port. The field identifies the peer streamlet needed at the receiver. When a message arrives at the receiving side, it is first distributed to a message distributor, where the *peerID* of the streamlet is checked. If the distributor can find a streamlet whose identification matches the *peerID* contained in the incoming message, then the distributor will deliver the message to the streamlet. Once a message has been processed by all necessary peer streamlets, it is delivered to the application.

6.6 Message Loss Avoidance

In the process of stream configuration, it is not unusual for messages to be queued in a streamlet buffer, while waiting to be processed. As a result, it is necessary that MobiGATE exercises message loss avoidance to prevent unprocessed messages being discarded owing to the removal and insertion of streamlets. It is important to note that MobiGATE does not attempt to facilitate peer-to-peer streamlet synchronization during the removal process. While it provides mechanism for peer-to-peer streamlets to pass control messages, it is the responsibility of the peer streamlets to ensure that state information and data are appropriately handled before MobiGATE removes the peer streamlets from the stream.

To avoid pre-mature termination of streamlets and avoidance of message loss during the reconfiguration process, the system checks if the pre-established conditions have been satisfied for the target streamlet. The conditions are depicted in Figure 6-8. If the conditions are satisfied, the streamlet can be removed safely. Otherwise the system has to wait some time or take special actions, until all conditions are satisfied.



Prerequisites to terminate a streamlet

- All producers of the streamlet are stopped or suspended. (New messages will not arrive at its input port any more)
- The input port of the streamlet is empty. (No messages to process)
- The streamlet is not processing any messages. (Not on message)

Figure 6-8: Prerequisites to terminate a streamlet

By adopting this mechanism, incoming messages can be guaranteed to ultimately appear at the output port in a stream under normal operations. Though it is still possible to lose messages in some very special conditions (such as streamlets processing speed mismatch, a problem that is discussed later), it is argued that some further actions can be taken to minimize the occurrence, which forms part of future work in the implementation of the MobiGATE system.

6.7 Further Improvement

As introduced in Chapter 3, MobiGATE has a number of desirable properties. First, it maintains the intuitive flow of processing. Second, it supports reusability by promoting strong modularity between streamlets and decoupling of coordination from computation. New functions are easily added to the system by inserting streamlets at the appropriate point in the processing sequence. Third, it supports ease of modification, since streamlets are logically independent of other streamlets. In implementing the MobiGATE system, there exist several challenges and issues that may significantly impact the system's performance and usability. One of the major issues pertains to the incurrence of potentially large latency overheads caused by message copying across streamlets.

Furthermore, different streamlets may run at radically different speeds: it is unacceptable to slow one streamlet down because another streamlet is still processing data.

To handle these problems, the MobiGATE infrastructure employs a centralized message storage management, while utilizing memory references to pass messages between streamlets. In particular, the system maintains all incoming messages by storing them in a message pool and passing them between different streamlets by their associated message identifier. In other words, the system employs the passing by reference instead of value. The benefit of significantly reducing the copying overheads is demonstrated and discussed in the performance evaluation chapter. In addition, the system permits messages to be ignored by slow streamlets if they are in the middle of processing other messages. This is obtained by modifying the method `postMessage()` in the class *MessageQueue*, as shown in Figure 6-9.

```
public synchronized void postMessage(String msgID){

    while(msgQueue.size() >= max_size){
        try{
            //if the message is full / the downstream streamlet is a slow one
            wait(T);
            // if still full after T , drop the message
            if (msgQueue.size() >= max_size){
                System.out.println("Queue full, message "+msgID+" was dropped!!!");
                return;
            }
        } catch (InterruptedException e) {
        }
    }

    //add the message id into the queue
    msgQueue.add(msgID);

}
```

Figure 6-9: Excerpt from the method `postMessage()`

7 Performance Evaluation

In order to study the operation and performance of the MobiGATE system, a set of experiments on an emulated and controlled wireless environment is conducted. Significantly, these experiments provide a unique opportunity to measure the potential computation overheads that may be incurred by the MobiGATE system in providing active transport services, while allowing the collection of empirical data on the performance of the system. By analyzing and comparing the results, further insights into the characteristics of MobiGATE are expected. It is also hoped to thoroughly exercise the interactions between the software components with the ultimate aim of validating the functionality of the system.

The experiments begin with testing the MobiGATE streamlet in isolation, measuring the overhead brought by each streamlet when serving incoming messages. A set of experiments on the reconfiguration time was then conducted. These experiments enabled validation of the effectiveness of MobiGATE in facilitating context-aware computing through streamlet reconfiguration, together with the collection of empirical results on overheads incurred during reconfiguration. Finally, a case example with a particular application reacting to a changing bandwidth was studied to demonstrate the use of MobiGATE while verifying the insignificant overheads incurred in runtime processing. A comparison was made with the performance gained in service deployment and reconfiguration.

7.1 Testing Environment

As shown in Figure 7-1 the setup includes the use of three PCs: one acts as the MobiGATE server residing on the wired departmental LAN, a second acts as the mobile node, and the third is configured to act as a wireless router for emulating a wireless operating environment. The MobiGATE server and the Linux router are located on the same fixed LAN (158.132.11) within the campus network. Any requests to hosts outside the campus have to go through the transparent campus proxy server. The mobile node is connected to the second network interface of the Linux router using different network identification (10.0.0).

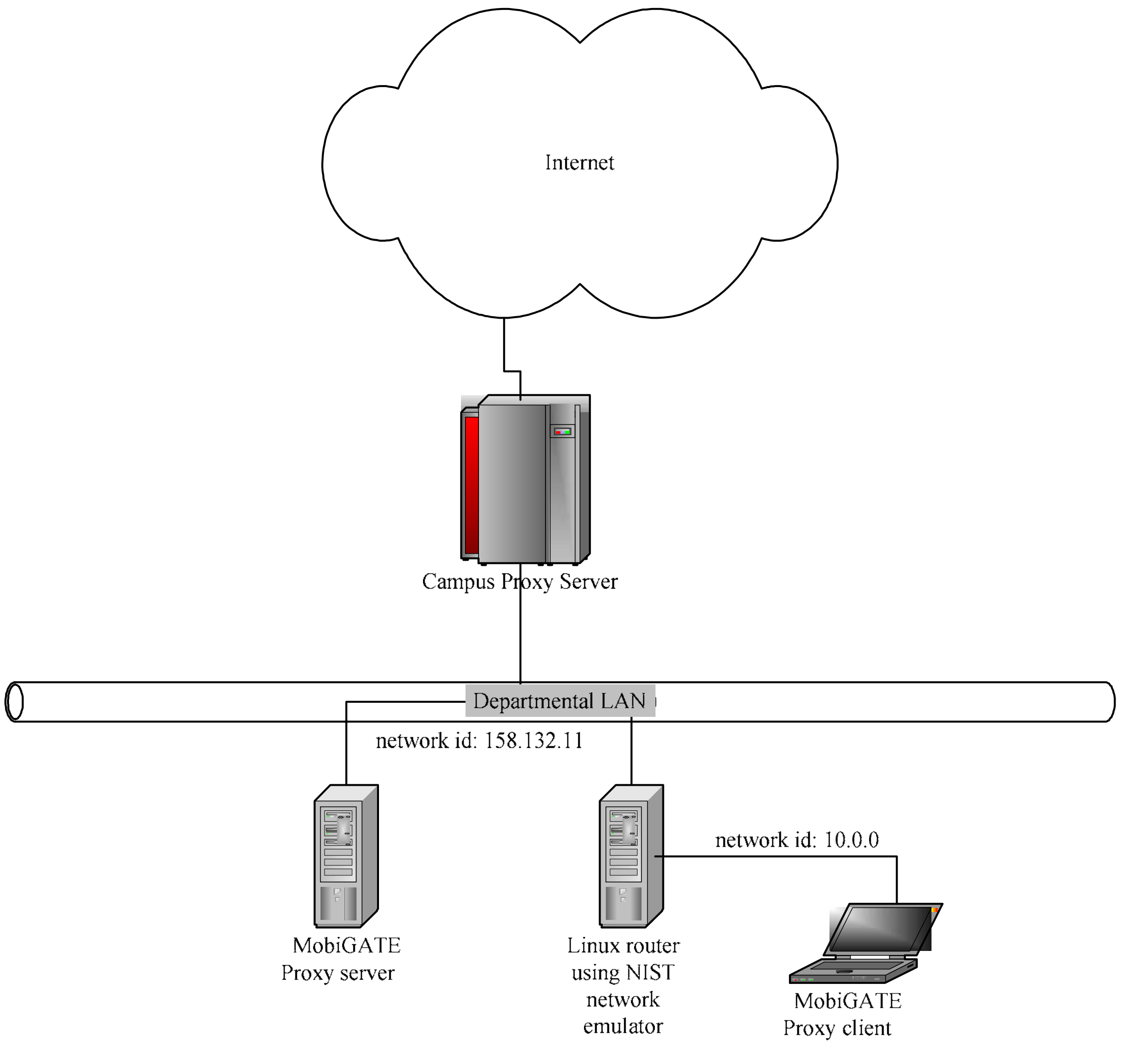


Figure 7-1: Testing environment

7.2 Streamlet Overhead Analysis

For a specific streamlet, ignoring the service processing time, the incurred overheads primarily come from two sources:

- The added work to parse and unparse incoming messages.
- The additional overhead in transmitting messages to and from other streamlets.

In this experiment, a special streamlet, named *redirector,* has been designed. Its primary logic is to read and parse incoming messages from its input port, encapsulating the necessary headers and sending the messages to its relevant output port. Significantly, the *redirector* streamlet contains core service codes that can be evaluated for overheads incurred in maintenance and execution over the MobiGATE runtime. Delay times can easily be captured by measuring the time needed for a size-specific message to pass through a configured number of streamlet *redirectors*. Considering the fact that the primary overheads incurred by the *redirector* streamlet are inherent in any streamlet for processing incoming messages, it is argued that the experiment setup is reasonable and realistic. The experimental results are shown in Figure 7-2.

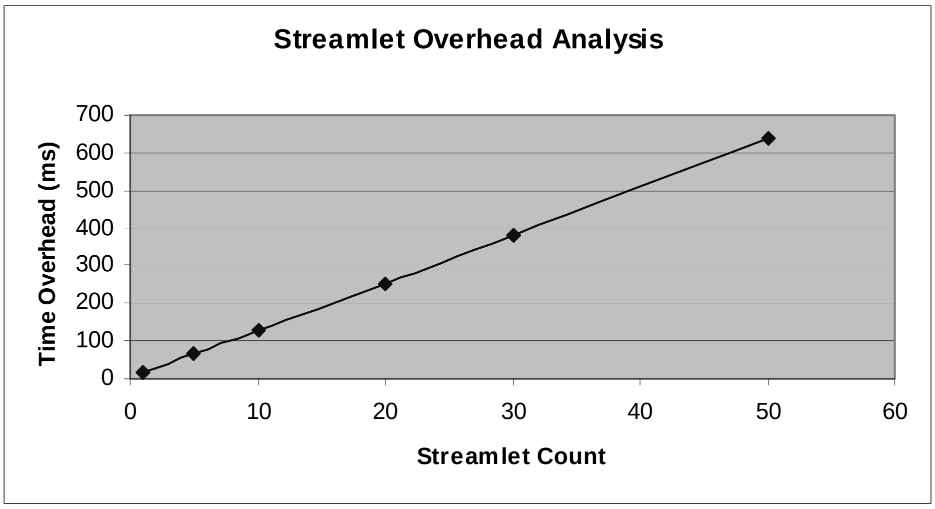


Figure 7-2: Streamlet overhead

The above figure shows that the delay overhead increases linearly with the increase in the number of streamlets the messages passed through. On average, the overhead is about 12 ms per streamlet. It is believed that the overhead can be further reduced with improved hardware configuration such as increasing the processor speed and increasing the available memory. Furthermore, in the realistic deployment of services, it is unlikely that more than ten streamlets will be used. That is to say, the overhead brought by these streamlets can safely be bound to about 100 ms, which is relatively acceptable compared with the potentially long transmission delay incurred in wireless transmissions.

7.3 Passing by Reference versus Passing by Value

The MobiGATE system maintains all incoming messages by storing them in a message pool and passing the messages between different streamlets by their associated message identifier. In other words, the system employs passing by reference instead of value. Figure 7-3 shows the experimental results when buffer management of MobiGATE is implemented based on reference passing versus value passing. In this experiment, several messages of different sizes were prepared and made to pass through a number of streamlet *redirectors* (thirty in the experiment) successively.

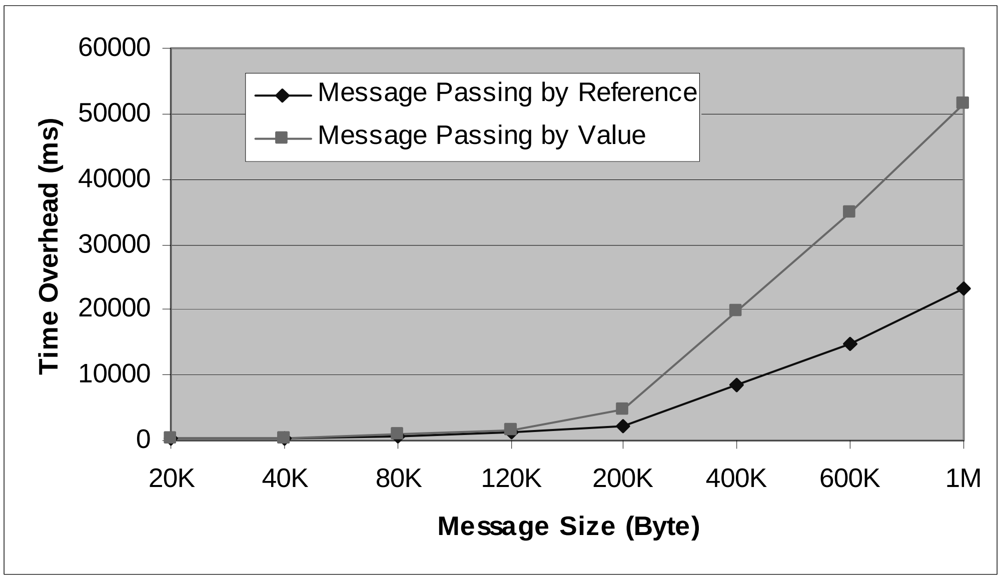


Figure 7-3: Passing by Reference versus Passing by Value

As expected, the experiment clearly indicates an increase in processing overheads with a progressive increase in the message size. The rate of increase is more prominent as the message size increases beyond 200K bytes. Across different message sizes, the processing latency is significantly lower for messages that are passed by reference compared to messages that are passed by value. In the former case, new incoming messages are copied into the message buffer pool once, while message headers and identifiers are treated as meta-data and references to be passed between streamlets. While the message header size may increase as more streamlets are chained in the stream, the size is still significantly lower than that of the actual message data. Avoiding copying of actual message data across streamlets also significantly reduces the amount of memory required by MobiGATE. This has the benefit of keeping messages stored and cached on fast memory, avoiding the need to swap between resident memory and secondary storage.

7.4 Reconfiguration Time

Dynamic reconfiguration in MobiGATE aims to maximize the performance of wireless access under a vigorously changing context environment. However, the reconfiguration process of service composition brings a certain number of performance penalties that are unavoidable. Reconfiguration time is the time taken for the MobiGATE system to adapt to changes in the wireless environment. In other words, reconfiguration time is the amount of time during which a user will find the MobiGATE system inactive due to reconfiguration.

Before going into the details of the experiment, the addition of a new streamlet is used as an example to illustrate a complete reconfiguration process. Figure 7-4 shows the steps of this process in detail:

1. *Three streamlets: A, B, and C. A and B are initially connected by a channel m. Assuming the need to insert C between A and B.*
2. *Suspend streamlet A.*
3. *Detach A from the channel m.*
4. *Attach C to the channel m.*
5. *Create a new channel n between A and C.*
6. *Activate streamlet A and the reconfiguration is finished!*

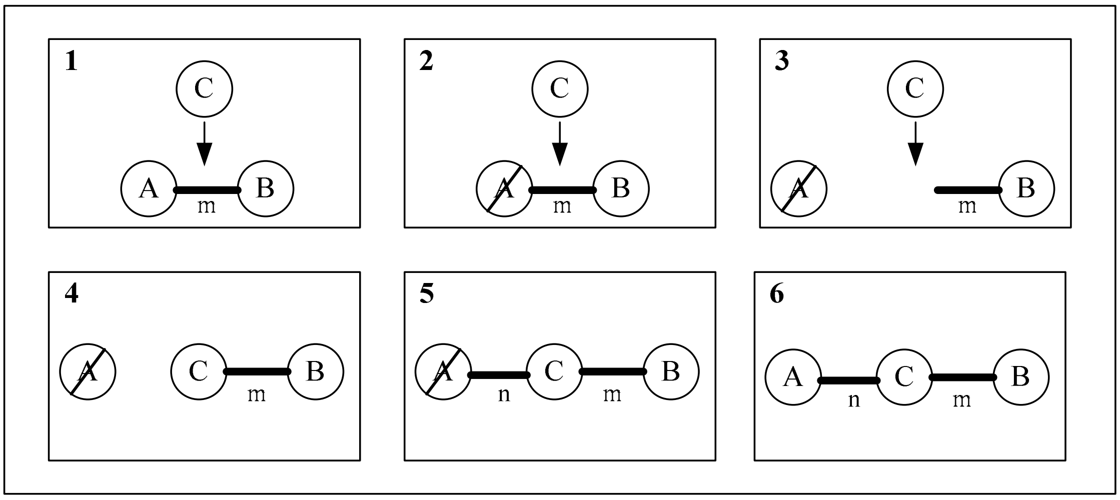


Figure 7-4: The addition of a new streamlet

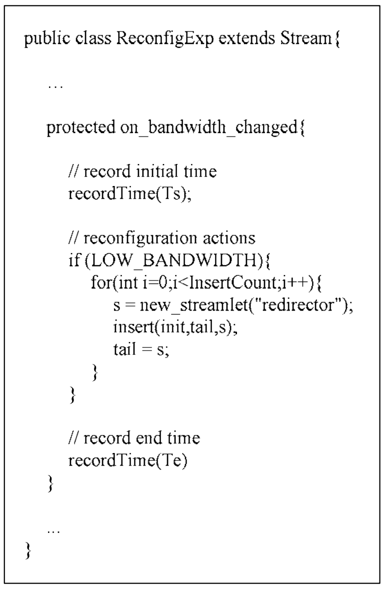
From the above illustration, it is not difficult to derive the reconfiguration time, which involves the following factors:

- $\sum_{i=1}^{k} s_i$ - Suspension of k streamlets
- *nc* - Creation (or Deletion for removal operation) of n channels
- $\sum_{i=1}^{k} a_i$ - Activation of suspended streamlets

Thus, the reconfiguration time (T) can be represented as:

$$T = \sum_{i=1}^{k} s_i + nc + \sum_{i=1}^{k} a_i \; . \qquad \text{--- Equation 7-1}$$

To evaluate the time required to reconfigure using the MobiGATE system, several reconfiguration actions were performed. Specifically, a stream application *ReconfigExp* was designed. It reacts to the *LOW_BANDWIDTH* event, which is defined in Table 6-1, by inserting a number of streamlets *redirectors*. As shown in Figure 7-5, the time T_s is recorded once at the beginning of the method and then, after a series of actions, the time T_e is recorded again as the ending time of the reconfiguration. By varying the number of streamlets inserted (the variable *InsertCount* in Figure 7-5), different numbers of reconfigurations can be measured and T_e - T_s will be the resultant time cost. Figure 7-6 shows the result of the experiment.



```
public class ReconfigExp extends Stream{
    ...
    protected on_bandwidth_changed{
        // record initial time
        recordTime(Ts);

        // reconfiguration actions
        if (LOW_BANDWIDTH){
            for(int i=0;i<InsertCount;i++){
                s = new_streamlet("redirector");
                insert(init,tail,s);
                tail = s;
            }
        }

        // record end time
        recordTime(Te)
    }
    ...
}
```

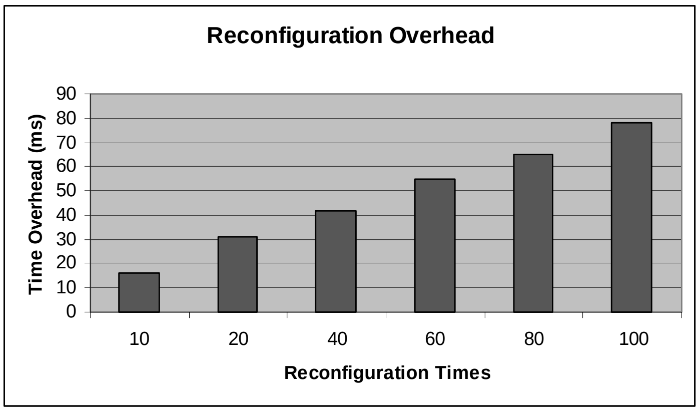


Figure 7-5: Excerpt from the class *ReconfigExp*

Figure 7-6: Reconfiguration overhead

Notice that when the number of added streamlets is less than 10, the reconfiguration time is less than 20 ms. Even when the number of streamlet additions reaches 100, the reconfiguration overhead is still less than 100 ms. This is a noteworthy and promising result considering that the reconfiguration rate is likely to be comparatively low (typically in terms of tens of seconds to minutes, depending on the contextual changes of the wireless environment) and the reconfiguration time is insignificant. The good reconfiguration performance is the result of an extensive use of multi-threading and object code sharing across streamlets, and of the separation of coordination from computation to accelerate and support ease of reconfiguration.

7.5 MobiGATE End-to-End Performance

After evaluating the overheads of key MobiGATE mechanisms, this section describes the overall system performance of MobiGATE from an end-to-end perspective. In particular, it is aimed to fully exercise the MobiGATE system components by setting up a realistic test bed in the form of a stream application operating over an emulated wireless network. The purpose is to verify the benefits of the MobiGATE system by asserting that the operations overhead is small compared with the improvement in performance that comes from using this system in a wireless environment.

Traditionally, if the MobiGATE system is not utilized, the time cost to transmit a certain amount of information can simply be represented like this:

$T_1 = \frac{Size}{Band}$, where *Size* represents the amount of information to transmit, and *Band* represents the bandwidth value.

By using the MobiGATE system, the information size for transmission can be greatly reduced[*], but this will also bring some overhead into the system, as introduced in previous sections.

$$T_2 = \frac{Size'}{Band} + T_{overhead} = \frac{Size - Size_{reduced}}{Band} + T_{overhead} = \frac{Size}{Band} + T_{overhead} - \frac{Size_{reduced}}{Band}$$

$$= T1 + (T_{overhead} - \frac{Size_{reduced}}{Band}) \qquad \text{--- Equation 7-2}$$

To justify the effectiveness of the MobiGATE system, the time costs T_1 and T_2 need to be evaluated for the same amount of information to be transmitted over wireless links. That is to say, the system throughput for these two different schemes must be compared to draw a conclusion.

For this purpose, a case study of an application that reacts to changes in bandwidth has been prepared. The application speeds up web surfing over slow links by including the following streamlets:

[*] Note that MobiGATE is not restricted to introducing services that optimize the amount of data to be sent across a wireless link. However, this is a direct and visible example to demonstrate the benefit of MobiGATE in terms of reducing transmission latency and improving link performance.

- *Switch*: Dividing incoming messages based on the semantic type of the data;
- *Gif2Jpeg*: Converting incoming image messages into Jpeg format;
- *Image Down Sampling*: Lossy compression of an image by reducing the sample rate;
- *Communicator*: Sending messages onto the network;
- *Text Compressor*: A generic text compressor. This streamlet has the potential to reduce the data size by up to 75%. Importantly, this streamlet is activated only if the bandwidth of the wireless link falls below 100 Kb/s. This setup provides the opportunity both to test the responsiveness of MobiGATE to context changes and to exercise the reconfiguration mechanisms.

In the application, an amount of real image and text messages are generated continuously. Image messages are processed by the streamlets *Switch*, *Gif2Jpeg*, *Image Down Sampling,* and *Communicator* successively from the start to the end, whereas the situation is different for text messages. Under normal conditions (bandwidth >100 Kb/s), the text messages only pass through the streamlets *Switch* and *Communicator*. But when the bandwidth falls below 100 Kb/s, the third streamlet, *Text Compressor,* is inserted between the above two streamlets to adapt to the poor bandwidth. After recording the sending and receiving time of each message, the time cost to transmit each message can be calculated and the overall system throughput is then obtained.

In the experiment, the system throughput under the bandwidth of 20Kb/s, 50Kb/s, 100Kb/s, 200Kb/s, 500Kb/s, 750Kb/s, 1Mb/s, and 2Mb/s was measured successively. For each bandwidth setup, three different transmission delays, <1ms, 50ms, and 100ms, were adjusted to evaluate the performance of the system. The final results are shown in Figure 7-7.

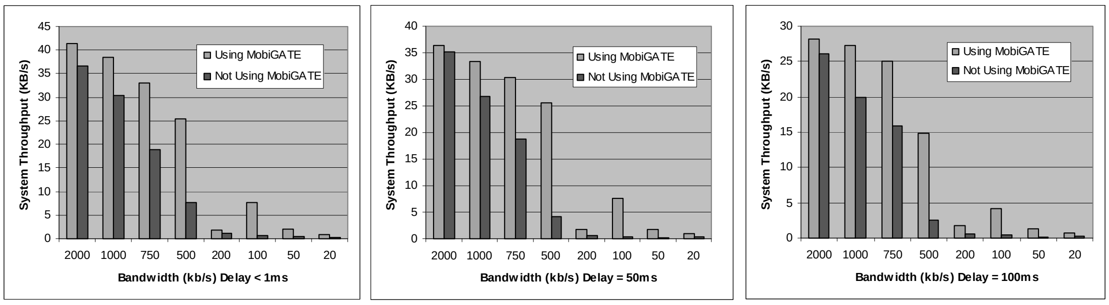


Figure 7-7: The effectiveness of the MobiGATE system

The above results can be analyzed from the following points:

1. A noticeable improvement in system throughput has been obtained with the MobiGATE system as compared with a setup for direct transfer of messages across the wireless link. Back to Equation 7-2, that is: $(T_{overhead} - \frac{Size_{reduced}}{Band}) < 0$ and $T_2 < T_1$.

2. When the bandwidth is about 2Mb/s, the system throughput difference is not very large with/without the MobiGATE system. This can be explained from Equation 7-2. When the bandwidth is relatively large (near 2Mb/s), for the same size of transmission, $(T_{overhead} - \frac{Size_{reduced}}{Band})$ will approach 0, and T_2 approach T_1. Thus the difference in the system throughput is minimal. But with a decreasing bandwidth, this difference becomes larger and larger. This is expected since the effect of applying streamlet services to reduce the amount of required bandwidth begins to take prominence.

3. When the bandwidth approaches 200Kb/s, which is a relatively low bandwidth for transmission, the difference in the system throughput becomes unnoticeable again. This is because the time costs T_1 and T_2 at this moment are very large,

causing both system throughputs to be relatively poor. Hence their differences can not be seen clearly from the result diagram. However, it does exist.

4. In an event when the bandwidth falls below 100 Kb/s, a special reconfiguration mechanism is invoked, in which the streamlet *Text Compressor* is inserted into the stream. The result indicates that the system throughput improves greatly. By comparison, the system throughput without the MobiGATE continuously drops with the decrease of the bandwidth.

The experiments clearly indicate the benefit of the MobiGATE system and its ability to offset processing overheads that may be incurred in deploying the streamlet application. This is particular true if MobiGATE is deployed in an environment where resource availability is dynamic and scarce.

8 Conclusions and Recommendations

In this study, a middleware system MobiGATE and its supporting coordination language MCL are proposed in order to adapt network data flows in the wireless domain. The focus of the work is to apply the coordination theory in the adaptation service composition and system reconfiguration at infrastructural proxies. This approach has a number of desirable properties, including reusability, ease of modification, and maintenance of the intuitive flow of processing.

8.1 Conclusions

A reconfigurable and adaptive system for mobile computing is expected to continue to take a prominent role in alleviating the poor traffic conditions of wireless links and resource limited mobile devices. The research study described in this thesis aims to develop a highly adaptable and reconfigurable middleware to adapt data flows across a wireless and mobile environment. To achieve this goal, an adaptive middleware, MobiGATE, has been designed, implemented, and verified to support robust and flexible composition of adaptable services, termed streamlets. Streamlets form the basic building blocks of a stream that adapts the flow of data across the wireless link. To achieve utmost flexibility and management of service adaptation, MobiGATE adopts the principle of *separation-of-concerns* to facilitate clear separation of streamlets' coordination from the service computation codes. This has resulted in the formulation of a two-layered MobiGATE execution platform that supports rapid deployment of service streamlets, while facilitating adaptive composition in reaction to changing environmental contexts. Additionally, MobiGATE is equipped with the necessary mechanisms and system services to support peer-to-peer streamlet collaborations with its thin-client model, which sets the MobiGATE system apart from other existing adaptive middleware. The design of MobiGATE is validated through the complete implementation of the system on a Java platform. Empirical experimental results conducted on the system demonstrated the effectiveness of the middleware in adapting data flows over an emulated wireless link, while incurring insignificant computational overheads in its execution environment.

MobiGATE Coordination Language (MCL) plays an important role in providing bridging between streamlets' computation and their interdependencies. The language provides rich constructs to support the definition of compositions, with constrained type validation and checking. In the description of the coordination, each service entity is regarded as a black box with well-defined interfaces. MCL enables the core functional pieces of an application to be clearly separated from its application-specific patterns of interdependencies. This is supported by two distinct language elements: *streamlets*, for representing core functional service entities; and *channels*, for representing relationships of interconnection among streamlets. The novel features of MCL include the modeling of service interfaces based on an MIME media type system, support for a check on the compatibility of the compositions, support for recursive compositions, and the concept of streamlet sharing. Significantly, the language is reinforced with a semantic model in *Z* language. Based on the derived semantic model, the applications running in the MobiGATE system can be analyzed to ensure that they are consistent in their internal structures.

8.2 Recommendations

As in most research work, the progress made in this study undoubtedly has not covered all new and interesting directions, but suggestions for future work to further enhance the performance of MobiGATE are given below. The future work can generally be organized into two parts: the MobiGATE architecture and its supporting language MCL.

8.2.1 MobiGATE Architecture

Throughout this thesis, the important function of the MobiGATE architecture in supporting MCL composition and providing runtime environment is especially emphasized. The separation of concerns is the underlying theme of this system. To make MobiGATE more complete and powerful, the following work is necessary:

- *Dynamic inclusion of new event objects*. In the current MobiGATE system, all the event objects are predefined and assumed to be recognized by all of the application developers. However, the future inclusion of the function of the dynamic addition of new event objects for the system is planned. By this means, application developers can propose their own event objects and define the corresponding event handlers.

- *The mechanism to support a wireless handoff*. When a mobile client equipped with multiple wireless interfaces switches between wireless networks, also known as a wireless handoff, a specific mechanism is needed to enable these mobile clients to use the MobiGATE system consistently. This mechanism may include the notification of the characteristics of the new network, the migration of adaptation services if necessary, and the synchronization of the application status. With the *separation-of-concerns* fulfilled in the design of MobiGATE, streamlet adaptation services can run independently of the environment and other streamlets. It is argued that this advantage can greatly facilitate the implementation of the handoff mechanisms in the future.

- *Communications between streamlets and the coordinator*. According to the current design, streamlets communicate with the external environment only through their data ports. In the future, it is expected to associate each streamlet with a control interface that allows the external coordinator to set operation parameters for the streamlets. For example, the text compression streamlet might have parameters that determine compression rate. These inputs serve as configuration parameters for the whole application. In this way, each streamlet will have two methods to communicate with the external world: data ports to communicate with other streamlets for message processing, and control interfaces to receive parameter setting information from the coordinator.

- *Security and transaction concerns*. As a middleware system, MobiGATE needs to consider many system issues, far more than separation of concerns discussed in depth in previous chapters. System security and transaction control are such two important topics necessitating future exploration before MobiGATE can be realistically deployed in an open and wide area environment.

- *Other problems open to the future*. There are still some problems left in the current system to be solved, such as the problem of the processing speed mismatch between streamlets, and the synchronization of peer-to-peer streamlets during the removal process. The final resolution of these problems depends on the success of implementation of the work discussed above.

8.2.2 MobiGATE Coordination Language

No programming language design is ever complete. As more experience is gained with a programming language, additional features are added and existing features are modified to enrich its expressive power. MCL is expected to be the same. Below some immediate areas of future enhancements of MCL are identified:

- *Experience with MCL*. The most pressing short term need for research on MCL is to gain usage experience. To this date, only a sample application architecture is characterized using MCL, as introduced in Section 4.3. The applicability of MCL is explored with this application and how MCL's facilities can be of benefit in the system reconfiguration is shown. However, the case study described here does not capture the architecture properties of interest completely. A limitation is that this work has been carried out in an academic setting. MCL remains largely untested in actual practice in the work place.

- *More automated tools*. As discussed in Chapter 5, some automated tools based on *Z* notation have already existed for the analysis process. However, they are still too general to be used directly on MCL descriptions. It is planned in the near future to develop tools that are specific to the MCL language and can provide automated checking of the properties for at least a subset of MCL.

- *More systematic expression of architectural assumptions*. MCL uses attribute definitions for expressing architectural assumptions. Although attribute definitions seem to be powerful enough to express a number of relationships and constraints, the current system does not provide systematic guidelines on how and when to use them.

More research is needed to classify architectural assumptions, and standardize the way these assumptions are expressed in MCL.

8.3 Publications

Yongjie Zheng and Alvin T.S. Chan, "Stream Composition for Highly Adaptive and Reconfigurable Mobile Middleware", *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), Page: 122-127, Hong Kong, 28-30 September, 2004, IEEE.

Yongjie Zheng and Alvin T.S. Chan, "MobiGATE: A Mobile Gateway Proxy for the Active Deployment of Transport Entities" *Proceedings of the 2004 International Conference on Parallel Processing* (*ICPP 2004*), Page: 566-573, 15-18 Aug 2004, Montreal, Quebec, Canada, IEEE.

Yongjie Zheng and Alvin T.S. Chan, "MobiGATE: A Mobile Computing Middleware for the Active Deployment of Transport Services", *revised submission to IEEE Transactions on Software Engineering.*

Yongjie Zheng and Alvin T.S. Chan, “MCL: A MobiGATE Coordination Language for Highly Adaptive and Reconfigurable Mobile Middleware”, *submitted to Special Issue of “Software, Practice and Experience” journal on “Experiences with Auto-adaptive and Reconfigurable Systems”*.

References:

[Abowd95] Gregory D. Abowd, Robert Allen, David Garlan “Formalizing Style to Understand Descriptions of Software Architecture” ACM Transactions on Software Engineering and Methodology, Vol 4, No 4, October 1995.

[Agha02] Gul A. Agha, “Adaptive Middleware”, Communications of the ACM. June 2002/Vol. 45, No.6.

[Allman97] M. Allman, C. Hayes, H. Kruse and S. Ostermann, “TCP performance over satellite links”, in 5th International Conference on Telecommunications Systems (1997).

[Anastasi02] Giuseppe Anastasi, Marco Conti, Willy Lapenna “A Power-Saving Network Architecture for Accessing the Internet from Mobile Computers: Design, Implementation and Measurements” The COMPUTER JOURNAL, Vol. 46, No. 1, 2003.

[Arbab96] F. Arbab, “The IWIM Model for Coordination of Concurrent Activities”, First International Conference on Coordination Models, Languages and Applications (Coordination ‘96), Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.

[Astley01] Mark Astley, Daniel C. Sturman, and Gul A. Agha, “Customizable Middleware for Modular Distributed Software”, Communications of the ACM. May 2001/Vol. 44, No.5

[Badrinath00] B. Badrinath, A. Fox, L. Kleinrock, G. Popek, P. Reiher, and M. Satyanarayanan, "A conceptual framework for network and client adaptation," ACM MONET Journal, Vol. 5, No. 4, (Dec. 2000), pp. 221-231.

[Bakre97] A. Bakre and B. Badrinath, "Implementation and performance evaluation of Indirect TCP," IEEE Transactions on Computers, Vol. 46, No. 3, (Mar 1997), pp. 260-278.

[Balakrishnan95] H. Balakrishnan, S. Seshan, E. Amir and R. Katz, Improving TCP/IP performance over wireless networks, in: Mobicom ’95 (November 1995).

[Balakrishnan97] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz, "A comparison of mechanisms for improving TCP performance over wireless links," IEEE/ACM Transactions on Networking, December 1997.

[Barbacci93] Mario R. Barbacci, Charles B. Weinstock, Dennis L. Doubleday, Michael J. Gardner and Randall W. Lichota “Durra: a structure description language for developing distributed applications”. Software Engineering Journal, March 1993

[Barbara99] Daniel Barbara, “Mobile Computing and Databases --- A survey”. IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, January/February 1999.

[Beringer98] Dorothea Beringer, Catherine Tornabene, Pankaj Jain, Gio Wiederhold “A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules” DEXA 98: Large-Scale Software Composition, Vienna, August 1998.

[Blair00] Gordon S. Blair, Lynne Blair, Valerie Issarny, Petr Tuma, Apostolos Zarras, “The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms”. IFIP/ACM International Conference on Distributed systems platforms, 2000.

[Caceres95] R. Caceres and L. Iftode, "Improving the performance of reliable transport protocols in mobile computing environments," IEEE JSAC, Vol. 13, No. 5, (Jun 1995), pp. 850-857

[Chan03] Alvin T.S. Chan and Siu Nam Chuang, "MobiPADS: A Reflective Middleware for Context-Aware Computing", IEEE Transactions on Software Engineering, vol. 29, no. 12, Dec 2003, pp. 1072-1085.

[Chan04] Eugene Wong, Alvin T.S. Chan, H.V. Leong. “Xstream: A Middleware for Streaming XML Contents over Wireless Environments”; IEEE Transactions on Software Engineering, Volume: 30, Issue: 12, Dec. 2004 Pages: 918 - 935

[Fox96] A. Fox, S.D. Gribble, E.A. Brewer and E. Amir, “Adapting to network and client variability via on-demand dynamic distillation”, in: Proc. 7th Internat. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), Cambridge, MA (October 1996).

[Fox98a] A. Fox, S. Gribble, Y. Chawathe and E. Brewer, "Adapting to network and client variation using infrastructural proxies: lessons and perspectives" IEEE Personal Communications, Vol. 5, No. 4, (Aug. 1998), pp. 10-19.

[Fox98b] A. Fox, I. Goldberg, S.D. Gribble, D.C. Lee, A. “Polito and E.A. Brewer, Experience with top gun wingman, a proxy-based graphical Web browser for the USR PalmPilot”, in: Proc. of IFIP Middleware ’98, Lake District, UK (September 1998).

[Freed96] N. Freed, “Multipurpose Internet Mail Extensions, (MIME) Part Two: Media Types” RFC 2046, November 1996.

[Garlan94] David Garlan and Mary Shaw “An Introduction to Software Architecture” Carnegie Mellon University Technical Report CMU-CS-94-166, January 1994.

[Gelernter92] David Gelernter and Nicholas Carriero “Coordination Languages and their Significance”. Communications of the ACM, 1992. 35(2): P.97-107.

[Katz94]	R. Katz, "Adaptation and Mobility in Wireless Information Systems," IEEE Personal Communications, Vol. 1, No. 1, (Q1 1994), pp. 6-17.
[Kolman96]	Bernard Kolman, Robert C. Busby, Sharon Ross "Discrete Mathematics Structures" 1996 by Prentice Hall, Inc. pp 369.
[LaMaire96]	R. LaMaire, A. Krishna, P. Bhagwat, J. Panian, "Wireless LANs and mobile networking: standards and future directions," IEEE Communications Magazine, Vol. 34, No. 8, (Aug. 1996), pp. 86 -94.
[Magee89]	Jeff Magee, Jeff Kramer, and Morris Sloman "Constructing Distributed Systems in Conic". IEEE Transactions on Software Engineering, vol. 15, no. 6, June 1989.
[Magee93]	Jeff Magee, Naranker Dulay, and Jeff Kramer "Structing parallel and distributed programs". Software Engineering Journal, March 1993.
[Malone94]	T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination", ACM Computing Surveys 26, 1994, pp. 87-119.
[Maes87]	Pattie Maes, "Concepts and experiments in computational reflection". Proceedings of OOPSLA'87, pages 147-155. ACM, October 1987.
[McKinley03]	Philip K. McKinley, Udiyan I. Padmanabhan, Nanadagopal Ancha, and Seyed Masoud Sadjadi, "Composable Proxy Services to Support Collaboration on the Mobile Internet". IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 6, JUNE 2003.
[McKinley04]	Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng "A Taxonomy of Compositional Adaptation" Technical Report MSU-CSE-04-17
[Noble97]	B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn and K. Walker, "Agile application-aware adaptation for mobility", in: Symposium on Operating System Principles (November 1997).
[Papadopoulos96]	George A. Papadopoulos, Farhad Arbab "Coordination Models and Languages" Advances in Computers, Marvin V. Zelkowitz (ed.), Academic Press. Vol. 46, August, 1998, 329-400.
[Perkins98]	C. Perkins, "Mobile Networking through Mobile IP," IEEE Internet Computing, Vol. 2, No. 1, (Jan.-Feb. 1998), pp. 58-69. (Also available as a PDF file)
[Peschanski01]	Frederic Peschanski, Christian Queinnec, Jean-Pierre Briot "A typeful Composition Model for Dynamic Software Architectures" July 2001.
[Rice94]	M.D. Rice and S.B. Seidman "A formal Model for Module Interconnection Languages" IEEE Transactions on Software Engineering, Vol. 20, NO. 1, January 1994

[Saaltink97]	M. Saaltink "The Z/EVES System." In ZUM'97 The Z Formal Specification Notation, pages 72-85. Lecture Notes in Computer Science. 1212 Springer, 1997.
[Satyanarayanan95]	M. Satyanarayanan, "Fundamental Challenges of Mobile Computing," ACM Symposium on Principles of Distributed Computing, 1995.
[Schmidt96]	Douglas C. Schmidt, "A Family of Design Patterns for Application-Level Gateways". Jounal Theory and Practice of Object Systems, special issues on Patterns and Pattern Languages, Wiley & Sons, Vol. 2, No. 1, December 1996.
[Silva98]	Sushil da Silva, Danilo Florissi, Yechiam Yemini, "Composing Active Services in NetScript" Position Paper, DARPA Active Networks Workshop, Tucson AZ, March 9-10, 1998.
[Sommerville96]	Ian Sommerville and Graham Dean, "PCL: a language for modeling evolving system architectures". Software Engineering Journal, March 1996.
[Spivey89]	J.M. Spivey, "The Z Notation, A Reference Manual." Englewood Cliffs, NJ: Prentice-Hall, 1989.
[Yang03]	Jian Yang, "Web Service Componentization". Communications of the ACM, October 2003/Vol. 46, No.10.