MobiGATE: A Mobile Computing Middleware for the Active Deployment of Transport Services

Yongjie Zheng and Alvin T.S. Chan, *Member*, *IEEE*

Abstract—The use of gateway proxies is one important approach to facilitating adaptation across wireless and mobile environments. Importantly, augmented service entities deployed within the gateway proxy residing on the wired network can be composed and deployed to shield mobile clients from the effects of poor network characteristics. The usual approach to the static composition of service entities on the gateway proxy is to have these service entities interact with each other by explicitly invoking procedures on the named interface, but such a tight coupling of interfaces inhibits the flexible composition and adaptation of the service entities to the dynamic operating characteristics of wireless networks. In this paper, we present a ***Mobi***le ***G***ATEway for the ***A***ctive deployment of ***T***ransport ***E***ntities or, for short, MobiGATE (pronounced Mobi-Gate). MobiGATE is a mobile middleware framework 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. To facilitate the dynamic reconfiguration of the streamlets, we advocate applying the concept of coordination as the unifying approach to composing these transport service entities. Importantly, MobiGATE delineates a clear separation of interdependent parts from the service-specific computational codes of those service entities. It does this by using a separate coordination language, called *MobiGATE Coordination Language* (*MCL*), to describe the coordination among streamlet service entities. The complete design, implementation, and evaluation of the MobiGATE system are presented in this paper. Initial experimental results validate the flexibility of the coordination approach in promoting separation-of-concern in the reconfiguration of services, while achieving low computation and delay overheads.

Index Terms—Mobile computing, coordination languages, adaptive middleware, dynamic reconfiguration, infrastructural proxies.

1 Introduction

Ubiquitous access to data is becoming a reality due to the large-scale deployment of wireless communication services and advances in mobile computing devices. However, mobile computing environments exhibit operating conditions that differ greatly from their wired counterparts. In particular, applications must be able to tolerate the highly dynamic channel conditions that arise as users move about an environment. Moreover, computing devices often vary in terms of display characteristics, processor speed, memory size, and battery lifetimes. For mobile applications to operate effectively and optimally in such environments, the communication-related software must be able to adapt to dynamic conditions at runtime [6].

One way to meet these challenges 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 [7]. With such architectures, applications are built from some interconnected building blocks and deployed at proxy stations. Each building block, or service entity, specializes in a specific task in processing the flow of data. 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 transiting of traffic 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.

A common approach to implementing the adaptation of services at the gateway proxy is to have the service entities interact statically with each other by explicitly invoking procedures on a named interface. The result is that the system integration code will become mixed up with the application-specific codes. Any replacement or modification of a service entity requires the updating not only of the code for the new service entity to be integrated to the system, but also of 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.

In this paper, we present the architecture of a ***Mobi***le ***G***ATEway for the ***A***ctive deployment of ***T***ransport ***E***ntities or, for short, MobiGATE (pronounced Mobi-Gate). MobiGATE is a mobile middleware framework 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

- *Y. Zheng is with the Department of CISE, University of Florida, 301 CSE, PO Box 116120, Gainesville, FL 32611. E-mail: yzheng@cise.ufl.edu.*
- *A.T.S. Chan is with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: cstschan@comp.polyu.edu.hk.*

Manuscript received 1 Nov. 2004; revised 28 Oct. 2005; accepted 29 Nov. 2005; published online 23 Jan. 2006.
Recommended for acceptance by B. Cheng.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0255-1104.

across the wireless network. To facilitate the dynamic reconfiguration of the streamlets, we advocate the application of the concept of coordination [13] as the unifying approach to composing these transport service entities. It is important to note that MobiGATE delineates a clear separation of interdependent parts from the service-specific computational codes of those service entities by using a separate coordination language to describe the coordination among streamlet service entities. To this end, we have defined a compositional language called *MobiGATE Coordination Language* (*MCL*) that provides rich constructs to support the definition of compositions, with constrained type validation and checking. In describing 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. MCL does this by supporting two distinct language elements: *streamlets*, for representing core functional service entities, and *channels*, for representing relationships of interconnection among streamlets. Each implementation-level entity is represented in MCL as optional attributes of streamlets and channels. This approach has a number of advantages, ranging from the ability to reuse common communication service entities, to offering flexible dynamic reconfiguration where transport entities can be inserted, removed, and reordered at runtime without recompilation or redefinition.

The MobiGATE middleware system directly supports composition using MCL and addresses the design and implementation of middleware services for dynamic, heterogeneous environments. A major goal of the MobiGATE architecture is to provide such an environment, where programmers can develop new mobile applications by combining some active service entities (streamlets), while the configuration structure of the application is completely separated from the computational activities of individual entities. The advantages of the architecture are that it supports ease of dynamic reconfiguration through the runtime reflective configuration of MCL and promotes the reusability of service entities across applications. The MobiGATE execution environment is comprised of a coordination plane for controlling MCL coordination activities and an execution plane for managing the execution of various computational service-specific processes, including streamlets.

The remainder of the paper is organized as follows: In Section 2, we describe background knowledge on adaptation using gateway proxies in mobile computing and provide an introduction to the area of coordination, software architecture, and mobile middleware. Section 3 details the architecture of MobiGATE, with an emphasis on some of the core modules driving the system. Section 4 focuses on the design of the coordination language, MCL, including its type system, language constructs, and some refinement works. Section 5 describes the design and development of the MobiGATE system. Section 6 presents the results of a series of experimental studies on an emulated wireless environment. Finally, Section 7 offers a discussion of future work and some relevant issues.

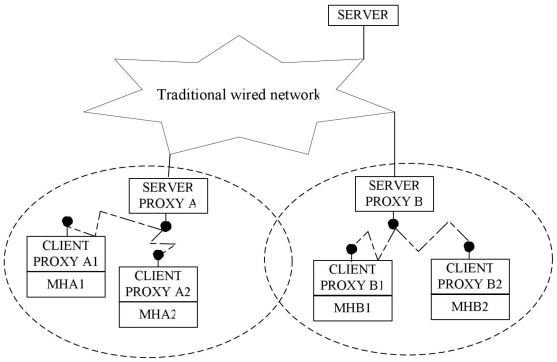


Fig. 1. Infrastructure proxy system.

2 Background and Related Work

This section begins with an introduction to gateway proxies, which represent an important architecture for introducing augmented services across a mobile and wireless environment. This is followed by a review of related work on coordination models and coordination languages to enable "plug-and-play" in the composition and evolution of services. Finally, we briefly introduce several architecture description languages and mobile middleware, which relate to the design of MobiGATE.

2.1 Gateway Proxy Services

Today's Internet clients vary widely with respect to both hardware and software properties. These variations are difficult to hide at the network level, making application-level techniques necessary. On-the-fly adaptation using transformational proxies is a widely applicable, cost-effective, and flexible technique for addressing all of these types of variations. Fig. 1 shows the infrastructure of the proxy adaptation system. It mainly consists of the following two network components residing between the wireless end-points: 1) a wired-side gateway called the server proxy that is commonly deployed at the edge of a wired network and 2) a peer client-side proxy called the client proxy that is deployed within the mobile host (MH).

The architecture supports augmented wireless network services by allowing adaptation-based service entities to be deployed at both the server and client proxies to shield clients from all kinds of variances. Importantly, 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 (filtering, format conversion, etc.), 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. In [7], a service entity based on image transcoding was applied to convert images on-the-fly so that the bandwidth requirement could be reduced and

the images displayed on a display-constrained device such as a PDA. Similarly, experiments have been conducted on text-compression, XML streaming, and caching service entities, which are deployed based on the architecture of the gateway proxy. The central motivation behind our work is to address more generally the complex challenges related to the design, implementation, and deployment of service entities that are highly composable and reconfigurable and that promote service adaptations that react to the dynamic operating conditions of wireless and mobile networking.

2.2 Coordination Models and Coordination Languages

Coordination models are a class of model that has recently been developed to describe concurrent and distributed computations. In the area of programming languages, coordination is defined as the process of building programs by gluing together active pieces. Thus, a coordination model can be regarded as the glue that binds separate activities into an ensemble [21]. 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 the coordination model is that we have 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. With recent advances in this field, a number of such coordination languages have become available, such as PCL [23], Conic [11], Durra [5], and Manifold [4]. These languages share many characteristics. In particular, the 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 output ports of producers and the input ports of consumers. While these languages support primitive constructs to enable a connection to be established between coordinating 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 as messages are exchanged between the respective input and output ports.

This paper presents the MCL language that supports the composition and reconfiguration of flexible streamlets in MobiGATE. In addition to all common properties shared by existing coordination languages, this newly designed language possesses its own type system, a compatibility check function, and the ability of recursively organizing streamlets. MCL employs *Multipurpose Internet Mail Extensions* (*MIME*) [8] specifications to model streamlet interfaces and message types. This is reasonable given the fact that MobiGATE is mainly used to facilitate mobile communications in wireless networks. MIME is an official Internet standard that specifies how messages are formatted and interpreted across e-mail systems. Since its inception, MIME has been extended and applied to diverse applications, including the adoption of the standard to the World Wide Web. In particular, MIME possesses a flexible and extensible format that easily accommodates well-known message types, such as text, images, video, sound, or other application-specific data. Based on the MIME type system, MCL allows type compatibility checks in the composition activities. Another effective property of MCL is the support of the notion of recursive composition in that streamlets and their compositions are absolutely indistinguishable from the point of view of other objects. 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 reusability.

TABLE 1
A Comparison of Coordination Languages

	PCL	Conic	Durra	Manifold	MCL
Coordination Unit	Family entities	Logical node	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
Application Domain	Model architectures of multiple versions of computer-based systems	Specify the configuration of software components in distributed systems	Support rapid prototyping of distributed heterogeneous applications	Manage interconnections among processes of a single application	Facilitate wireless proxy services composition based on streams

Table 1 offers a comparison between existing coordination languages and MCL along seven dimensions: *Coordination Unit* is the basic unit in terms of which the configuration is performed; *Computational Language* is the name of the language 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 in the above paragraph; the *Application Domain* refers to the application of the language in a domain for which it is designed.

It is important to note that Manifold, which is described as a typed language, does not have an actual type system to model the processed data information. By typed it is meant that Manifold only recognizes types of language elements, such as processes, ports, events, and streams [4]. Similarly, Conic cannot be regarded as a language with the full support of compatibility checking as it only employs some simple data types, such as integers, floats, and strings, to define data ports and cannot conduct the compatibility check that is supported by MCL based on the MIME type system. The reason that PCL and Conic only partially support dynamic reconfiguration is that the number of process and channel instances in these two languages is

fixed at the time the system is created [21], while the languages Durra, Manifold, and MCL do not have that restriction.

2.3 Software Architecture and Mobile Middleware

Our research on MobiGATE is also closely related to the fields of software architecture and mobile middleware. This section briefly introduces some relevant concepts applied to various systems and their impact on MobiGATE's design.

2.3.1 Software Architecture

The art of organizing software into various levels of abstractions has evolved since the inception of computing. From the perspective of programming languages, intuitive data structures and abstractions have been recognized as effective mechanisms to simplify programming efforts and organize program flows. On another plane, the increasing complexity of software programs has motivated software architects and researchers to investigate advanced techniques to organizing complex software. The advance from the functional to the object-oriented programming paradigm has most definitely driven the formulation of new languages such as C++, SmallTalk, and Java to promote the principles of encapsulation, information hiding, polymorphism, among other benefits of object-oriented programming. As software systems continue to grow in complexity, their design and specification in terms of coarse-grain building blocks become a necessity. The field of software architecture addresses this issue and provides high-level abstractions for representing the structure, behavior, and key properties of a software system. In this connection, software architecture represents an important framework for the MCL language that specifically focuses on dynamic service compositions in the MobiGATE system.

To date, many *architecture description languages* (*ADLs*) have been developed to aid architecture-based development. Examples of ADLs include Darwin [12], ACME [9], Rapide [10], Wright [2], and C2 [15]. Darwin is an architectural description language that describes a component type by an interface consisting of a collection of services that are either provided or required. In particular, Darwin provides a set of semantics for its structural aspects through the π-calculus. However, Darwin does not provide appropriate means of describing the core functions and connection capabilities of either a component or its services. ACME is intended to serve as a least-common-denominator interchange language for architectural descriptions. ACME is consistent with the ability to describe software architecture, but it does not, in itself, provide a sufficient basis. It is only through mappings to other languages that ACME descriptions can be interpreted. Rapide is an architectural description language based on modeling computations and interactions as partially ordered event sets. It is limited by its lack of an explicit means of introducing interaction types and not supporting symmetric interaction patterns. Wright provides not only a concise language to capture the architectural specification of software composition and interactions, it also facilitates automated checking of architectural properties to assert the architectural consistency and completeness of the system. However, Wright limits both the architectures that can be directly expressed (they must be static and asynchronous) and the properties that can be attributed to those architectures (only control and event ordering properties are naturally captured). Finally, C2 is a language designed specifically to support architecture-based evolution. Different from MCL, the component of C2 are specified as first-order expressions via invariants and operation pre and postconditions, which are generated as comments in an implementation and cannot be checked at system execution time.

2.3.2 Mobile Middleware

The infiltration of mobile computing has motivated the development of mobile middleware systems that support agile software architecture which reacts rapidly to varying operating environments. A software system dealing with agile components requires software architecture that is highly robust and composable to deal with potentially adverse operating conditions. Therefore, it is not surprising that the design of MobiGATE adopts and rides on the concepts of software architecture to promote a principled and systematic approach to organizing and managing adaptive software composition. Significantly, the underlying coordination language based on MCL provides syntactic expressions to enable component services to be composed and checked for consistency in a highly dynamic environment. In the remaining section, we describe several representative middleware technologies and present a comparison of these technologies with MobiGATE.

ArchJava [1] is an extension to Java that unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints. ArchJava currently has several limitations that would likely limit its applicability in the MobiGATE setting: Communication between ArchJava components is achieved solely via method calls, it is currently limited to Java, and its efficiency has not yet been assessed. Lime [18] is a Java-based middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility, or both. Lime is specifically targeted at the complexities of ad hoc mobile environments. XMIDDLE [14] is a data-sharing middleware for mobile computing. XMIDDLE allows applications to share data that are encoded as XML with other hosts, to have complete access to the shared data when disconnected from the network, and, when possible, to reconcile any changes made with all the hosts sharing the data. XMIDDLE also allows applications to influence the reconciliation process. Aura [24] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. Aura is thus only applicable to certain classes of applications in the MobiGATE setting. The other well-known systems also include TranSend [7], Odyssey [20], and RAPIDware [16].

Compared with the existing middleware systems introduced above, the advantage of the MobiGATE system is in its flexible and dynamic composition of services based on the principles of coordination and separation-of-concerns. MobiGATE supports various forms of service compositions, linear or branching, and can be configured at runtime without recompilation or redefinition. More importantly, as

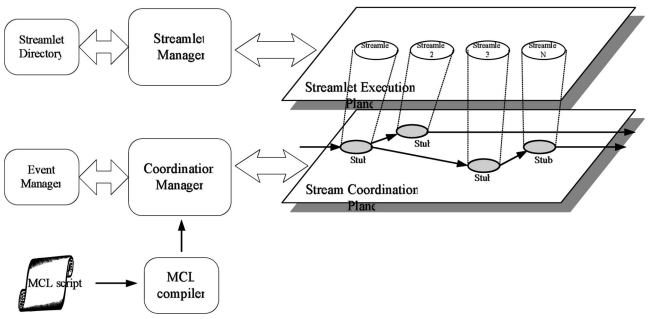


Fig. 2. Architecture of the MobiGATE server.

introduced above, the system supports the automatic compatibility check in the composition activities. All these desirable properties come from the application of the coordination principle advocated in this paper.

3 An Overview of the MobiGATE Architecture

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.

There exists in the MobiGATE server a clear distinction between the activities of coordination and computation. Fig. 2 shows the architecture of MobiGATE, 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 initial composition of streamlets and reconfiguration actions in response to different events. All this information comes from the compilation of the corresponding MCL script that enforces those high level policies. Based on the derived table, the `Coordination Manager` is responsible for initializing and reconfiguring related applications appropriately.

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 environmental and context conditions. Finally, there is also a `Streamlet Directory`, where the streamlet providers can advertise their services. This directory provides code-level implementations of streamlets at runtime. We discuss in detail various components of the MobiGATE architecture below.

Coordination Manager. The `Coordination Manager` controls the generation of stubs and channel objects and facilitates the exchange of messages 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 coming from the `Event Manager` and broadcast them among coordination streams, which may invoke dynamic reconfiguration actions if necessary.

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 the request to streamlet instances and receive the result. The exchange of data among the stubs is currently conducted through channels. The channels transport data using a frequently used method called *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 read only after they have been written.

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 will create an instance for it from the `Streamlet Directory`; otherwise, the manager will directly deliver the message to the `Streamlet Execution Plane`.

Streamlet Execution Plane. All of the computation activities take place in the `Streamlet Execution Plane`. In this plane, individual streamlets run independently of the others and focus on imposing services on the incoming messages. We distinguish two kinds of streamlets, *Stateless* and *Stateful*, depending on whether or not they keep state information 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 a certain threshold, the increase in the number of streamlets will have a direct impact on performance and may, as a result, lower 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, as well as 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 instances of streamlet 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 instance of a stream. Thus, the system can keep a much smaller number of streamlets, reusing each instance of streamlet to service different requests. By this means, it greatly reduces the resources actually needed to satisfy all of the requests for the service.

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

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. In case there are any incompatible connections in the script, the compiler should return with a detailed error message.

Streamlet Directory. The `Streamlet Directory` serves as the repository where streamlet providers, which may be application developers themselves or some external streamlet developers, 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, which defines a recursive composition of other native streamlets.

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. To support this function, each streamlet on the sending side of a connection adds a header field *peerST* to the messages before writing them to its output port. The field identifies the peer streamlet needed at the receiver. Given a streamlet that performs some processing on an outgoing message, its peer streamlet performs the reverse processing on incoming messages. When a message arrives at the receiving side, it is first distributed to a message distributor, where the *peerST* of the streamlet is checked. If the distributor can find a streamlet whose identification matches the *peerST* 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 overlaying application.

4 MobiGATE Coordination Language

The MobiGATE Coordination Language (MCL) provides a declarative specification of the composition and coordination of streamlets through various construct abstractions. The language is not concerned with the operational logic of the streamlet, but seeks to capture the streamlet's high-level abstract characteristics, such as interfaces, through the abstraction of input-output ports and the types of data associated with the messages. The coordination and relationship between the streamlets are captured through the abstraction of channels.

4.1 Message and Port Typing

Typing 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 accordingly treated as such during compilation and execution. In MobiGATE, we view the typing of messages exchanged between streamlets and the definition of port types as fundamental to enabling the flexible and robust composition of service entities. Importantly, it allows the developer to concisely capture the intended types of messages that are bound to the ports of streamlets. Runtime checking in the form of matching the types of messages to the streamlet ports can be exercised to ensure consistency during the operation. In MobiGATE, we propose the adoption of the *Multipurpose Internet Mail Extensions*, or MIME 1.0 Internet standard as the underlying type definition to represent messages and declarations of port type. As such, messages exchanged in the system are formatted based on Multipurpose Internet Mail Extensions, or MIME 1.0. 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 e-mail, news, and the World Wide Web.

Fig. 3 shows a graphical representation of the type system. A fundamental property of the type system is that, for a given type, there can be multiple associated direct subtypes or supertypes. This is useful in facilitating the process of checking for compatibility of type in activities to compose the architecture. Another interesting property of the type system we defined comes from the extensible nature of the MIME type media system, meaning that it is not difficult to introduce a new type of message into the system.

Based on the MIME type system, the BNF (Backus Normal Form) notation of a type declaration in MCL can be defined as shown in Fig. 4. Note that this definition is generated from a simplification of a standard MIME

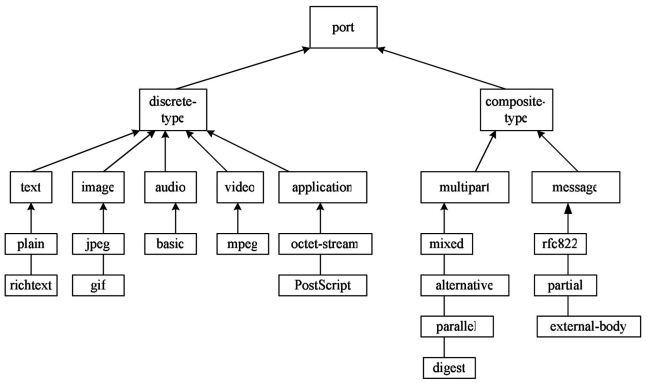


Fig. 3. Graphical representation of a type system.

Content-Type header field definition with some of our own modifications.

4.2 Language Elements

The MCL language 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 following section describes the important elements representing the core abstractions.

4.2.1 Streamlet

Streamlets represent the main functional pieces of an application. 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 placeholder, which 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, without the explicit need to know the real source/destination of messages. The separation and externalization of the interconnections of the streamlets promote the independence of streamlets from

```
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>
```

Fig. 4. BNF notation of the type declaration.

```
<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> ::= ' (' ' )' ' <' ' >' ' @' ' ,' ' ;'
                ' ' ' \' |<' >' /' ' [' ' ]' ' ?' ' ='
```

Fig. 5. BNF notation of the streamlet definition.

their environment and their reusability. In MCL, we use the notation $p.i$ to refer to port i of a streamlet instance p.

Streamlets are defined as sets of ports and attributes that describe their core functions and their capabilities to interconnect with the rest of the system, as shown in Fig. 5. As part of the declaration of ports, it is necessary to establish the type of input/output port. Notice that each streamlet may have more than one input/output port, each of which is identified with the name of a port. The attribute part specifies three important attributes: 1) **Type**. Type describes whether the streamlet needs to keep information on states for the corresponding application. Based on this attribute, streamlets are distinguished as STATELESS or STATEFUL. 2) **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. In particular, the library may refer to an MCL description file that defines another stream application in case of recursive compositions. This is illustrated by an example in Section 4.4.2. 3) **Description**. Description provides some general descriptive information about streamlets.

Furthermore, we distinguish between the descriptions of streamlets and their instances in MCL. In our terminology, a streamlet is 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, MCL requires that all interconnections among streamlets be explicitly represented using channels. Like streamlets, channels own ports. Channel 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>
                                  <buffer size>
                               ' }'
<channel type> ::= ' type" ' =' value1' ;'
<buffer size> ::= ' buffer" ' =' value2' ;'
value1 ::= ' SYN' ' ASYN'
value2 ::= *(DIGIT)' Kbytes'
token ::= *(<any (US-ASCII) CHAR except SPACE, CTLs, or tspecials>)
tspecials ::= ' (' ' )' ' <' ' >' ' @' ' ,' ' ;'
              ' :' ' \' |<' >' /' ' [' ' ]' ' ?' ' ='
```

Fig. 6. BNF notation of the channel definition.

```
<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>
```

Fig. 7. BNF notation of the stream definition.

A channel represents a reliable, directed, and perhaps buffered flow of information in time. Reliable means that the messages placed into a channel are guaranteed to flow through without loss, error, or duplication, with their order preserved. Directed means that there are always two identifiable ends in a channel: 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. Fig. 6 shows the formal definition of the channel. Like the streamlet, it is also defined by port declarations and some important attributes: 11) **Type**. We distinguish two types of channels: 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. 2) **Buffer**. Buffer specifies the size of the buffer in the channel in units of Kbytes. Ideally, an asynchronous channel should have an unbounded buffer, as introduced above. However, in the real world, we generally choose a large buffer size to simulate this property.

As with streamlets, we also differentiate 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. On the one hand, streams can be viewed as streamlets connected by channels with the ability to perform adaptations. On the other hand, a stream can also be viewed as a "streamlet" with input/output ports, which come from the stream's inner streamlet ports that are not concerned with any inner connections. Fig. 7 is the formal definition of a stream object.

In addition to the *new-streamlet*, *remove-streamlet*, *new-channel*, and *remove-channel* primitives introduced above, we also have *connect*, *disconnect*, and *disconnectall* primitives to set up/break down connections in descriptions of stream. For example, we write *connect* $(p.o, q.i, c)$ to set up a connection between port o of streamlet p and port i of streamlet q, using channel c. For simplicity, we can also just use *connect* $(p.o, q.i)$, whereby the system will automatically create a channel instance of an asynchronous type with 100 Kbytes of buffer to connect between the ports. The port types of the created channel are based on the types of streamlet ports being connected by this default channel.

Dynamic reconfiguration is another important task that needs to be addressed in the description of a stream. In MCL, the interaction model is event-driven. That is, a coordinator process waits on an occurrence of some specific events that trigger it to enter 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. In MCL, we have predefined several types of events that represent external events that can be subscribed to initiate the adaptation through the reconfiguration of the composition of the streamlets. The selection of the types of event 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 coordinated streamlets.

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

4.3 Case Example

In this section, we present a pragmatic example of the composition of service entities based on MCL to illustrate and highlight the robustness of the language in regulating complex adaptations in response to evolving wireless and mobile operating environments. We make use of the Datatype-Specific Distillation application that was deployed

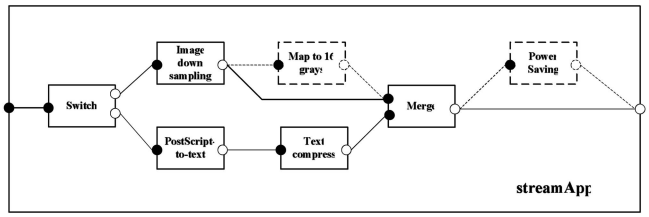


Fig. 8. The composition model of a Datatype-Specific Distillation application.

at the University of California, Berkeley [7], together with some of our own modifications, as an example for illustration. The service entities used in this example are listed below.

1. *Switch*: Dividing the incoming message based on the semantic type of the data;
2. *Image down sampling*: Lossy compression of an image by reducing the sample rate;
3. *Map to 16 grays*: Reducing images to 16 grays to support shallow grayscale displays;
4. *PostScript-to-text*: Discarding some information on format and converting documents to rich-text supported by most devices;
5. *Text compress*: A generic text compressor;
6. *Merge*: Integrating different types of information into a whole body;
7. *Power Saving*: A power-saving mechanism as discussed in [3].

Fig. 8 is the composition model of this application. In the figure, 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 on the condition that the system subscribes to and correspondingly receives the *LOW_ENERGY* signal from the hardware abstraction driver. Containing the composition of the streamlets is the abstraction of the stream application *streamApp*, which exercises recursive composition. The composite *streamApp* streamlet has its own input/output ports that are derived from those internal ports that are 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.

Fig. 9 is a description of individual streamlets in MCL. Considering the large size of image data, we specifically created a channel with a buffer of 1,024 Kbytes to connect image-related streamlets, while others use the default 100 Kbyte-sized channel.

Based on these streamlet descriptions, the final composition script for the stream *streamApp* is written as shown in Fig. 10.

As shown in Fig. 10, the occurrence of *LOW_ENERGY* triggers the reconfiguration of the stream by introducing the streamlet *powerSaving*. Similarly, the occurrence of

```
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;
    buffer = 1024 Kbytes;
  }
}
```

Fig. 9. Streamlet and channel descriptions.

LOW_ GRAYS triggers the insertion of a new streamlet *map_ to_16_grays* to provide transcoding of color images to gray scale images.

4.4 Design Issues

The design of MCL is greatly influenced by a set of core design issues. These issues, in a way, differentiate MCL

```
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_GRAYS){
    disconnect(s2.po, s7.pi1);
    connect(s2.po, s3.pi, c2);
    connect(s3.po, s7.pi1, c3);
  }
}
```

Fig. 10. Stream description.

from existing and general coordination languages, with the specific focus on facilitating robust composition and support for dynamic reconfiguration in a mobile and wireless environment.

4.4.1 Checking Compatibility

In a manner analogous to the checking of type in programming languages, it is desirable to be able to perform 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: 1) Streamlet ports can only be connected to channel ports (and vice versa) and 2) 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 of them come from streamlets, or channels, the connection is considered illegal. For the second restriction, MCL bases its checking of compatibility on types of port. 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 a subtype of a type of sink port, the connection is considered legal. In the application shown in Fig. 8, the connection between the *PostScript-to-text* output port and the *Text compress* 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, we include a keyword *main* to indicate the highest level stream object in a coordination script. As such, the system can start to execute an MCL application by locating a stream object that is labeled *main* in the coordination script.

To support this recursive composition, we need to compose a separate description of the streamlets associated with each stream object. 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, we can reuse the example stream discussed above, as shown in Fig. 11.

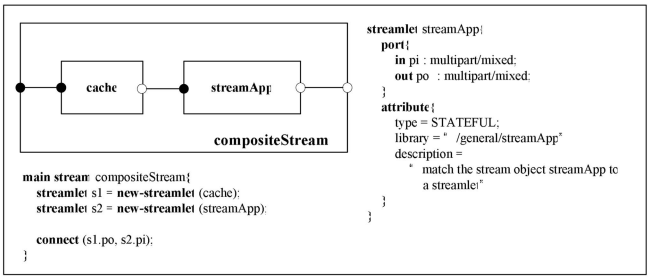


Fig. 11. Recursive composition.

As shown in Fig. 11, the composite stream is oblivious in the internal structure of the stream *streamApp*, which is defined in Fig. 10. From the view point of the composite stream, this stream object is just a common streamlet that is implemented in MCL. In a similar manner, this composite stream 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 coordination theory in facilitating organized composition. As MobiGATE evolves, coupled with the proliferation of streamlets, we envisage a need to provide a coordinated and structured organization of streamlets 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 contribution of our work is the concept of streamlet sharing. As each streamlet is only concerned with imposing its computation on incoming messages and producing response messages, it is oblivious to the source or destination of the 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 we differentiate between messages belonging to different instances of a stream?

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

$$session ::= \text{"}Content\text{-}Session\text{"}\ \text{":"}\ session\text{-}id.$$

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

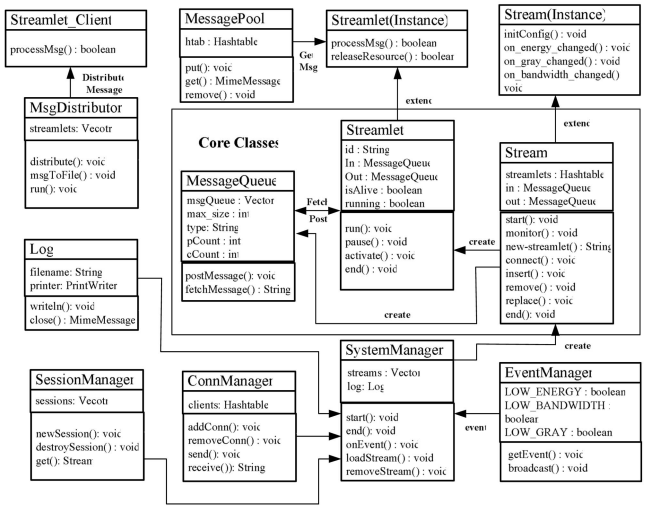


Fig. 12. The MobiGATE system class diagram.

```
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() { ... }
}
```

Fig. 13. Excerpt from the class *Streamlet*.

5 Development of the MobiGATE System

This section describes the design and development of the MobiGATE system that supports the necessary framework for streamlets to be easily composed, inserted, and removed. This system forms the underlying runtime layer 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 sublayers. Importantly, the runtime system is designed to promote the maximum reusability of system services while minimizing overheads incurred due to streamlets operations. The aim is to provide a general and flexible system that supports the rapid development and deployment of streamlet applications without dictating how the streamlet operation flows.

This section will not discuss the low-level details of the implementation code. Rather, it will highlight three major abstract classes that are pervasive in the MobiGATE model.

The *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.

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

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

Fig. 12 shows the simplified class diagram of the complete implementation. The following sections briefly describe the main classes that make up the MobiGATE infrastructure.

5.1 Streamlet

An excerpt of the *Streamlet* base class is shown in Fig. 13. 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 a reference to the *In* and *Out* objects in the *Streamlet* code itself. Several lifecycle methods are also defined in the Streamlet class, such as *pause()*, *activate()*, and *end()*, to manage lifecycle operations of the streamlets 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 downsampling, color to gray conversion, compression, and encryption. Connecting between streamlets in the *Streamlet* object is achieved through the use of the *In* and *Out* object abstractions.

5.2 MessageQueue

This class is used to manage the communications among streamlets on a given stream. In the class, there is a message vector, *msgQueue*, accessible to 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 will block it until the object is

```
public class MessageQueue{

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

    //Producer/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(){ … }

}
```

Fig. 14. Excerpt from the class *MessageQueue*.

unlocked. Second, the producer must have some way of indicating to the consumer that the message is ready and the consumer must have some way of indicating that the value has been retrieved. The *Thread* class provides a collection of methods—*wait*, *notify*, and *notifyAll*—to help the threads wait for a condition and notify other threads of when that condition changes.

In particular, we have included in the class two important integer-typed attributes, 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 the *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 shown in Fig. 14 is excerpted from the *MessageQueue* class.

5.3 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 the connection setup, reconfiguring the system in response to different events, and defining composition primitives. The initialize 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, several methods are 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 composition. 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 of these defined primitives are used in the composition of specific stream applications. Fig. 15 is excerpted from the class *Stream*.

In expressing a stream for an application, the developer is required to capture the streamlets' composition in MCL, which essentially captures the initial connection topology and reconfiguration schemes. Upon deploying the 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){ … }

}
```

Fig. 15. Excerpt from the class *Stream*.

application within the MobiGATE infrastructure, the system will automatically create the corresponding stream instances from these descriptions by extending the base class *Stream* and overriding the related methods (e.g., *initConfig()*, *on_xxx_changed()*, where *xxx* 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 resources 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 Performance Evaluation

In order to study the operation and performance of the MobiGATE system, we conducted a set of experiments on an emulated and controlled wireless environment. Importantly, these experiments provide us with a unique opportunity to measure the potential computation overheads that may be incurred by the MobiGATE system in providing active transport services and allow us to collect empirical data on the performance of the system. By analyzing the results, we hope to gain further insights into the characteristics of MobiGATE. We also hope to thoroughly exercise the interactions between the software components with the ultimate aim of validating the functionality of the system in terms of its core operations in providing stream compositions and to verify its performance gain against the potential overheads incurred from using MobiGATE.

6.1 Testing Environment

As shown in Fig. 16, the setup includes the use of three PCs: one (Pentium 4 CPU 2.60GHz 512M DDR RAM PC with WinXP) acts as the MobiGATE server residing on the wired departmental LAN (100M Ethernet), a second (Pentium 3 CPU 650MHz 256M SDRAM PC with WinXP) acts as the mobile node, and the third (Celeron PC with Linux) is configured to act as a wireless router for emulating a

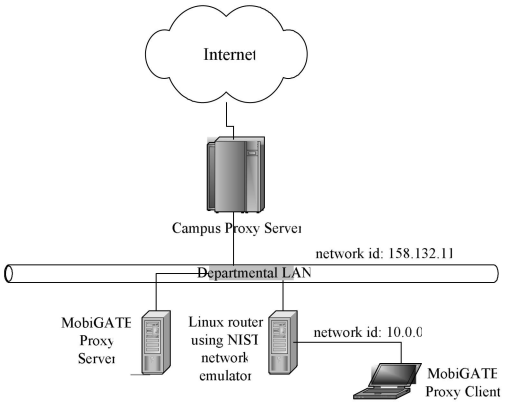


Fig. 16. Testing environment.

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 a different network identification (10.0.0).

This experimental system is necessary to enable us to validate the operations of MobiGATE in a real network environment. The Linux router is installed with the NIST NET network emulator [19] that acts as a general purpose tool for emulating performance dynamics in wireless networks. Importantly, NIST supports options for the user to select and monitor specific traffic streams passing through the router and to apply selected effects on the IP packets of those streams. The setup of a single server and single client is to simplify the experimental scenario, while still retaining the end-to-end semantics of the experiments.

6.2 Experimental Results

The experiments in this section focus on exercising some common operations in MobiGATE, such as message passing between streamlets and the reconfiguration of streamlet compositions, and evaluating the overheads that may subsequently be incurred. These operations form the core mechanisms that drive the streamlets and are inherent in all applications running in the MobiGATE system. Based on these basic measurements, a representative application consisting of all the evaluated operations can then be deployed and evaluated in the experiments. This gives us a realistic platform to evaluate the effectiveness of the complete MobiGATE system and its impact on the application's performance over a wireless environment.

Specifically, we started by testing the MobiGATE streamlet in isolation, measuring the overhead brought by each streamlet when serving incoming messages. After that, we conducted a set of experiments on the reconfiguration time, which indicates the level of responsiveness of MobiGATE to the changing transmission bandwidth. These experiments allowed us to validate the effectiveness of MobiGATE in facilitating context-aware computing through the reconfiguration of streamlets and to collect empirical results on overheads incurred during reconfiguration. Finally, a case example with a particular application reacting to changing bandwidth was studied 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.

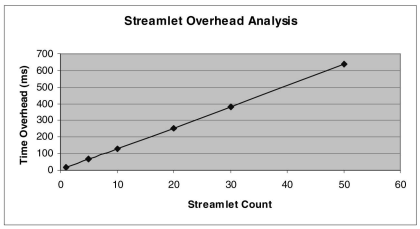


Fig. 17. Streamlet overhead.

6.2.1 Streamlet Overhead Analysis

In this experiment, we have designed a special streamlet, named the *redirector*, whose 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. Importantly, the *redirector* streamlet contains core service codes that can be evaluated for its 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, we argue that the setup of the experiment is reasonable and realistic. The experimental results are shown in Fig. 17.

From the above, we observed 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. This is acceptable compared with the potentially long transmission delay incurred in wireless transmissions. For a specific streamlet, ignoring the service processing time, the overheads incurred primarily come from the added work to parse and unparse incoming messages and the additional overhead in transmitting messages to and from other streamlets. We aim to investigate techniques, such as improving the configuration of the hardware and reducing the number of necessary streamlets, to improve the system performance.

6.2.2 Passing by Reference versus Passing by Value

The MobiGATE system maintains all incoming messages by storing them in a message pool and passes the messages between different streamlets by their associated message identifier. In other words, the system employs passing by reference instead of value. Fig. 18 shows the experimental results when the buffer management of MobiGATE is implemented based on reference passing versus value passing. In this experiment, we prepared several messages of different sizes and had them successively pass through a number of streamlet *redirectors* (30 in our experiment).

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

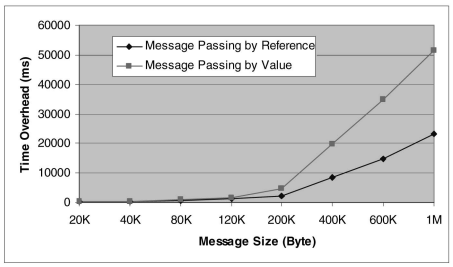


Fig. 18. Passing by reference versus passing by value.

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 the copying of actual message data across streamlets also significantly reduced 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.

6.2.3 Reconfiguration Time

Dynamic reconfiguration in MobiGATE aims to maximize the performance of wireless access under a vigorously changing environment. However, the reconfiguration process of service composition brings a certain number of performance penalties that are unavoidable. The 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, we would like to use the addition of a new streamlet as an example to illustrate a complete reconfiguration process. Fig. 19 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. Assume the need to insert C between A and B.

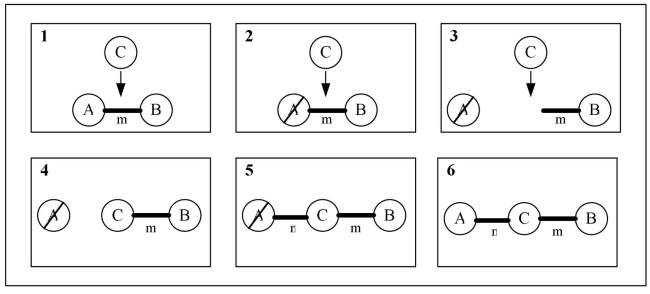


Fig. 19. The addition of a new streamlet.

```
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);
    }

    ...
}
```

Fig. 20. Excerpt from the class *ReconfigExp*.

2. Suspend streamlet A.
3. Detach A from channel m.
4. Attach C to channel m.
5. Create a new channel n between A and C.
6. Activate streamlet A and the reconfiguration is finished!

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:

$$T = \sum_{i=1}^{k} S_i + nc + \sum_{i=1}^{k} a_i.$$

To evaluate the time required to reconfigure using the MobiGATE system, we experimented with several reconfiguration actions. Specifically, we designed a stream application *ReconfigExp* that reacts to the *LOW_BANDWIDTH* event by inserting a number of streamlets *redirectors*. As shown in Fig. 20, we record the time Ts at the beginning of the method once and then, after a series of actions, record the time Te again as the ending time of the reconfiguration. By varying the number of streamlets inserted (the variable *InsertCount* in Fig. 20), we can measure different numbers of reconfigurations and $Te - Ts$ will be the resultant time cost. Fig. 21 shows the result of the experiment.

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 additional streamlets reaches 100, the

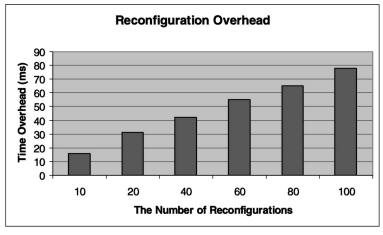


Fig. 21. Reconfiguration overhead.

reconfiguration overhead is still less than 100 ms. This is a noteworthy and promising result considering the fact 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 extensive use of multithreading and object code sharing across streamlets and of the separation of coordination from computation to accelerate and support ease of reconfiguration.

6.2.4 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, we aim to fully exercise the system components of MobiGATE by setting up a realistic test bed in the form of a streamlet application operating over an emulated wireless network. In this section, we aim to verify the benefits of the MobiGATE system by asserting that the operations overhead is small compared to the improvement in performance that comes from using this system in a wireless environment.

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

1. *Switch*: Dividing the incoming message based on the semantic type of the data.
2. *Gif2Jpeg*: Converting incoming image messages into Jpeg format.
3. *Image Down Sampling*: Lossy compression of an image by reducing the sample rate.
4. *Communicator*: Sending messages onto the network.
5. *Text Compressor*: A generic text compressor. This streamlet has the potential to reduce the data size by up to 75 percent. Importantly, this streamlet is activated only if the bandwidth of the wireless link falls below 100 kbps. This setup will provide us with the opportunity both to test the responsiveness of MobiGATE to context changes and to exercise the reconfiguration mechanisms.

In the application, a certain amount of image and text messages are continuously generated and transferred across the wireless link between the MobiGATE server and client via TCP socket connections. As for image messages, they are processed by the streamlet *Switch*, *Gif2Jpeg*, *Image Down Sampling*, and *Communicator* successively from the start to the end, while the situation is different for text messages. Under normal conditions (bandwidth >100 kbps), the text messages only pass through the streamlet *Switch* and *Communicator*. But, when the bandwidth falls below 100 kbps, the third streamlet, *Text Compressor*, will be inserted between the above two streamlets to adapt to the poor bandwidth. After recording the sending and receiving time of each message, we can get the time cost to transmit each message and finally calculate the overall system throughput.

In the experiment, we measured the system throughput under bandwidths of 20kbps, 50kbps, 100kbps, 200kbps, 500kbps, 750kbps, 1Mbps, and 2Mbps successively. For each bandwidth setup, we adjusted three different transmission delays, <1ms, 50ms, and 100ms, to evaluate the performance of the system. The final results are shown in Fig. 22.

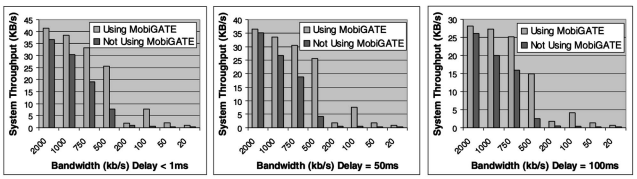


Fig. 22. The effectiveness of the MobiGATE system.

From the above results, we can see a noticeable improvement in system throughput using the MobiGATE system as compared to a setup using the direct transfer of messages across the wireless link. The throughput gain extends as bandwidth is decreased. This is expected since the effect of applying streamlet services to reduce the amount of bandwidth required begins to take prominence. A fall in the bandwidth below 100 kbps invoked a special reconfiguration mechanism in which the compress text streamlet was inserted into the stream. The results indicate that the system throughput improved greatly (from 1KB/s to 4-7KB/s). The experiments clearly suggest the advantages of the MobiGATE system and its ability to offset processing overheads that may be incurred in deploying the streamlet application. This is particularly true if MobiGATE is deployed in an environment where resources are dynamic and scarce.

7 Conclusion

This paper presents a novel coordination language called MCL that captures the description of the composition of proxy services in a wireless environment. The services offered in MobiGATE are composed of streamlets that are chained together in the form of a stream that adapts to the flow of data traffic to alleviate the poor characteristics of a wireless environment. The dynamic changing characteristics of a mobile and wireless environment mean that MobiGATE needs to support the dynamic reconfiguration of services through the evolutionary composition of a mix of streamlets. This is achieved by separating the interactions among streamlets from their computation through the abstraction of a coordination plane. In particular, the complexity of directly coding the flow of interactions among streamlets is captured via the abstraction of a new coordination flow language called MCL. The novel features of MCL include the modeling of service interfaces based on a MIME media type system, support for a check on the compatibility of the compositions, support for recursive compositions, and the concept of streamlet sharing. The complete implementation of the MobiGATE framework has, along with the experimental results, provided us with deep insights into the system. The results have verified the advantages of applying composable streamlet services to mitigate the effects of dynamically changing wireless conditions.

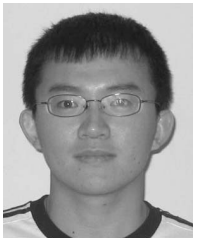
To date, we have completed the definition of the MCL coordination language and the supporting MobiGATE architecture. A prototype infrastructure of this architecture has been implemented on a Java execution platform, which supports a highly portable system for operating across heterogeneous environments. We plan to port the Java implementation of the MobiGATE client proxy so that it can operate on resource-constrained devices such as PDAs and mobile phones executing on a J2ME platform. The complete setup of the experimental platform will provide us with the unique opportunity to truly exercise and validate the operations of MobiGATE and to collect empirical results on the performance of the system. Initial experiments conducted with the system have produced promising results and will be the subject of our future publications.

ACKNOWLEDGMENTS

This work was supported by the Hong Kong Polytechnic University Central Research Grant G-U154 and Internal Competitive Research Grant AP-F82.

REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," *Proc. Int'l Conf. Software Eng. 2002,* May 2002.
[2] R.J. Allen, "A Formal Approach to Software Architecture," PhD thesis, CMU Technical Report CMU-CS-97-144, Carnegie Mellon Univ., May 1997.
[3] G. Anastasi, M. Conti, and W. Lapenna, "A Power-Saving Network Architecture for Accessing the Internet from Mobile Computers: Design, Implementation and Measurements," *The Computer J.,* vol. 46, no. 1, 2003.
[4] F. Arbab, "The IWIM Model for Coordination of Concurrent Activities," *Proc. First Int'l Conf. Coordination Models, Languages, and Applications (Coordination '96),* pp. 34-56, Apr. 1996.
[5] M.R. Barbacci, C.B. Weinstock, D.L. Doubleday, M.J. Gardner, and R.W. Lichota, "Durra: A Structure Description Language for Developing Distributed Applications," *Software Eng. J.,* Mar. 1993.
[6] A.T.S. Chan and S.N. Chuang, "MobiPADS: A Reflective Middleware for Context-Aware Computing," *IEEE Trans. Software Eng.,* vol. 29, no. 12, pp. 1072-1085, Dec. 2003.
[7] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer, "Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives," *IEEE Personal Comm.,* vol. 5, no. 4, pp. 10-19, Aug. 1998.
[8] N. Freed, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," RFC 2046, Nov. 1996.
[9] D. Garlan, R.T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," *Foundations of Component-Based Systems,* G.T. Leavens and M. Sitaraman, eds., pp. 47-68, Cambridge Univ. Press, 2000.
[10] D.C Luckham, L.M. Augustin, J.J. Kenney, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Trans. Software Eng.,* vol. 21, no. 4, pp. 336-355, Apr. 1995.
[11] J. Magee, J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans. Software Eng.,* vol. 15, no. 6, June 1989.
[12] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," *Proc. Fourth ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 3-14, Oct. 1996.
[13] T.W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination," *ACM Computing Surveys,* vol. 26, pp. 87-119, 1994.
[14] C. Mascolo et al., "XMIDDLE: A Data-Sharing Middleware for Mobile Computing," *Personal and Wireless Comm. J.,* vol. 21, no. 1, Apr. 2002.
[15] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution," *Proc. 21st Int'l Conf. Software Eng. (ICSE '99),* pp. 44-53, May 1999.
[16] P.K. McKinley, U.I. Padmanabhan, N. Ancha, and S.M. Sadjadi, "Composable Proxy Services to Support Collaboration on the Mobile Internet," *IEEE Trans. Computers,* vol. 52, no. 6, June 2003.
[17] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng, "Composing Adaptive Software," *Computer,* vol. 37, no. 7, pp. 56-64, July 2004.
[18] A.L. Murphy, G.P. Picco, and G.-C. Roman, "LIME: A Middleware for Physical and Logical Mobility," *Proc. 21st Int'l Conf. Distributed Computing Systems,* pp. 524-533, Apr. 2001.
[19] Nat'l Inst. Standards and Technology, "NIST Net," http://snad.ncsl.nist.gov/nistnet/, 2006.
[20] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile Application-Aware Adaptation for Mobility," *Proc. Symp. Operating System Principles,* Nov. 1997.
[21] G.A. Papadopoulos and F. Arbab, "Coordination Models and Languages," *Advances in Computers,* M.V. Zelkowitz, ed., vol. 46, pp. 329-400, Aug. 1998.
[22] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
[23] I. Sommerville and G. Dean, "PCL: A Language for Modeling Evolving System Architectures," *Software Eng. J.,* Mar. 1996.
[24] J.P. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," *Proc. Working IEEE/IFIP Conf. Software Architecture,* Aug. 2002.



Yongjie Zheng received the BE degree in computer science and technology from Tsinghua University in 2000 and the MPhil degree in computer science from the Hong Kong Polytechnic University in 2005. He is currently a PhD student in the Department of Computer and Information Science and Engineering at the University of Florida. He worked as a software engineer for IBM in Shanghai from 2001 to 2003. His current research includes adaptive middleware, software architecture, and mobile computing.

Alvin T.S. Chan graduated from the University of New South Wales with the PhD degree in 1995 and was subsequently employed as a Research Scientist by the CSIRO, Australia. He is currently an associate professor at the Hong Kong Polytechnic University. From 1997 to 1998, he was employed by the Centre for Wireless Communications, National University of Singapore as a program manager. Dr. Chan is one of the founding members of a university spin-off company, Information Access Technology Limited. He is an active consultant and has been providing consultancy services to both local and overseas companies. His research interests include mobile computing, context-aware computing, and middleware for adaptive computing. He is a member of the IEEE and ACM.