# Distributing a SIP Servlet Engine for Standalone Mobile Ad Hoc Networks

Basel Ahmad

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montréal, Québec, Canada

February, 2007

*Your file Votre référence*
*ISBN: 978-0-494-28909-9*
*Our file Notre référence*
*ISBN: 978-0-494-28909-9*

# Abstract

**Distributing a SIP Servlet Engine for Standalone Mobile Ad Hoc Networks**

**Basel Ahmad**

Mobile Ad Hoc Networks (MANET) are expected to become major components of Next Generation Networks (NGN). The success of MANET will be largely determined by the availability of services to users. The protocol of choice to establish service-enabling multimedia sessions in NGN is the Session Initiation Protocol (SIP). This is due to the ability of SIP to support both traditional telephony services and emerging services for convergent networks. Several technologies to enable easy programmability of SIP services have emerged; most notably the SIP Servlet Technology. In this technology, services are programmed using SIP servlets. A SIP servlet is a Java application that runs in an environment provided and managed by a SIP Servlet Engine (SE). Applicability of SIP Servlet Engines to MANET is an interesting topic because of the success that SIP Servlet Technology has enjoyed in provisioning services for infrastructure networks.

A SIP SE is a centralized server node that is not suited for infrastructureless environments that consist of nodes that are both mobile and transiently present. Currently, SIP Servlet technology has provisions for creating "distributable" servlets which allow multiple instances of the same servlet to run on multiple nodes. On the one hand this kind of servlet distribution can achieve load sharing and high availability in infrastructure networks. On the other hand, this scheme is not sufficient for MANET because the SIP SE remains a fixed and centralized entity requiring powerful processing and storage capabilities.

We propose a solution to de-centralize the SIP SE by distributing its functionalities across multiple nodes. A collection of nodes may be present in a MANET such that SIP SE functionality can be assembled from the combined capabilities of the available nodes. When this condition is present, these nodes can come together to collectively pose as a SIP SE to the entities that wish to either provide or consume SIP services.

In this thesis a distribution scheme that would allow a SIP SE to be used in MANET has been devised and implemented. The resulting scheme has been evaluated with two conferencing services.

# Acknowledgements

I would like to express my sincere gratitude to my mentors and supervisors Dr. Ferhat Khendek and Dr. Roch Glitho for guiding me at every stage of my research, their patience, encouragement, constructive criticism and support.

My thanks also go to my employer Ericsson Canada Inc. for the financial support of my studies through their employee tuition assistance program.

My gratitude goes to my wife Anne-Marie and my little daughter Athena for the sacrifices they made to help me complete my studies. I also thank my Mom and my family in Jordan, Ukraine and elsewhere for their encouragement and support.

I dedicate this work to the memory of my father Dr. Wahbeh Ahmad.

**Basel Ahmad, February 2007**

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms and Abbreviations

| | |
|---|---|
| AP | Access Point |
| API | Application Program Interface |
| ASP | Active Server Pages |
| BSS | Basic Service Set |
| CGI | Common Gateway Interface. |
| DARPA | Defense Advanced Research Projects Agency |
| ESS | Extended Service Set |
| EU | End-User |
| HTTP | Hyper Text Transfer Protocol |
| IBSS | Independent Basic Service Set |
| IEEE | Institute of Electrical and Electronics Engineers |
| IETF | Internet Engineering Task Force |
| J2EE | Java 2 Platform, Enterprise Edition |
| JVM | Java Virtual Machine |
| LAN | Local Area Network |
| LIME | Linda in a Mobile Environment |
| MANET | Mobile Ad Hoc Network |
| MMUSIC | Multiparty Multimedia Session Control |
| PAN | Personal Area Network |
| PRNet | Packet Radio Network |
| RFC | Request For Comments |
| RMI | Remote Method Invocation |

| | |
|---|---|
| RTP | Real-time Transport Protocol |
| SE | Servlet Engine |
| SEP | Servlet Execution Environment Provider |
| SMTP | Simple Mail Transfer Protocol |
| SP | Service Provider |
| SSP | Servlet Storage Provider |
| TCP | Transport Control Protocol |
| UA | User Agent |
| UAC | User Agent Client |
| UAS | User Agent Server |
| UAV | Unmanned Aerial Vehicle |
| UDP | User Datagram Protocol |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| WLAN | Wireless Local Area Network |
| WPAN | Wireless Personal Area Network |
| XML | Extensible Markup Language |

# Chapter 1: Introduction

This introduction provides definitions of key concepts of the research undertaken in this thesis, the issues, and outlines their importance. Both the problem statement and the proposed solution are presented and discussed. This chapter concludes with an outline for the rest of this thesis.

## 1.1 A Brief Introduction to the Domain

The advent of mobile, cellular telephony has enabled a vast number of consumers to communicate remotely. Moreover, wireless LAN technologies [1] also provided users with a wireless access to the Internet. However, both mobile telephony and wireless Internet access are only available where “the coverage exists”. In other words, a wireless terminal must be located within transmission range of an entity that provides access to voice or data communication. The traffic received by the entity that provides coverage, traverses multiple nodes towards destinations over a pre-deployed, centrally managed infrastructure. This places a limitation on those wireless services that must be made available in places where no such infrastructure exists. This has led to the inception of Mobile Ad Hoc Networks (MANETs) [2, 3, 4]; wireless networks formed without the aid of an existing infrastructure. Initially, MANETs were deployed by the military on the battlefield and by rescue and recovery crews in disaster areas. Recent advances in technology as well as changes in consumer habits (e.g., a large number of people accustomed to consuming mobile, wireless services) paved the way for the introduction of MANET technology to the consumer domain. This, in turn, requires service delivery platforms capable of delivering MANET services to the consumers.

Availability of services and user experience in accessing them is a determining factor that decides whether a given technology will succeed. In wireless communication, users are not satisfied with being limited to the traditional two-party voice call service. Instead, they demand additional services like: multi-party conferencing, instant messaging, and Internet access. In order to provide such services, signaling protocols capable of establishing sessions between mobile terminals are required. Moreover, service provisioning requires platforms that enable easy creation, distribution, deployment and delivery of services over the signaling protocols used.

Session Initiation Protocol (SIP) [5] is a signaling protocol to control multimedia sessions. SIP allows for session establishment, termination, participants' addition or removal, and modifying session characteristics. The description of the sessions is carried in a SIP message's body. SIP is text-based; its message structure resembles that of HTTP [6]. SIP is not specific to any particular type of multimedia sessions. It supports voice over IP sessions, traditional telephony services (e.g., caller-id, call-transfer, and multi-party voice conferencing) [7], and Internet services (e.g., instant messaging, presence, multimedia conferencing) [8].

SIP servlet technology [9] enables performing creation of SIP services. A SIP servlet is a Java [10] application specialized in performing SIP signaling logic. SIP servlets run in an environment provided by a SIP servlet engine (SE). SIP servlet development allows the programmer to focus on SIP service creation by letting SIP SE handle cumbersome SIP protocol details (like message retransmissions). SIP servlets are based on HTTP servlets [11] that have proven their success in server side web programming. A wide community of programmers exists with an expertise in Java, HTTP servlets, and XML [12]. This expertise would allow those developers to become productive in SIP servlet development with minimal learning curve.

## 1.2 Problem Statement and Contributions of this Thesis

SIP has become the protocol of choice for converged next generation networks. SIP servlets are the de facto tools for service provisioning with SIP in networks with fixed infrastructures. With the introduction of MANET to the consumer domain it becomes worthwhile to extend the use of SIP servlet technology to provide telecom services in MANET environments. However, SIP SE service provisioning model is not well suited for MANETs. In this model, a SIP SE is envisioned as a centralized, pre-deployed entity whose location is known to the clients. The SIP SE possesses powerful processing and storage capabilities and may implement a clustering mechanism to achieve high-availability. However, in MANET environments no such centralized entity with powerful capabilities exists.

The problem addressed in this thesis is to introduce SIP SE architecture suitable for MANET environments. We propose to do so by distributing a SIP SE across multiple nodes. The distribution scheme introduces a collection of nodes that combine their capabilities to collectively pose as a SIP SE to other actors. This distribution has multiple benefits. First, it will enable SIP service provisioning in environments where participant nodes have limited capabilities, by letting a group of nodes combine capabilities to collectively pose as a SIP SE. Furthermore, as a result of decentralizing the SIP SE, the distribution will allow for schemes that enable recovery should a participant node become unavailable by providing a substitute or by scaling down the service.

In this thesis we propose three contributions. First, we derive a set of requirements for distributing a SIP SE for MANET and show that the state-of-the-art fails to meet these requirements. As a second contribution, we propose and discuss a novel SIP SE distribution scheme that meets the requirements we derived. The third and final contribution is a proof-of-concept; a distributed SIP SE prototype. The performance of the prototype is evaluated by deploying and running two conferencing services. The results reported in this thesis have been submitted for publication [31].

## 1.3 Organization of this Thesis

Chapter 2, 'Mobile Ad Hoc Networks', provides background information on MANETs. It shows that the arrival of MANET applications to the consumer domain is a natural "next step" in the evolution of wireless, mobile networking. The chapter also discusses characteristics of MANETs and the impact of these characteristics on service provisioning. Understanding MANET characteristics is an essential step for deriving requirements for service provisioning in MANET.

Chapter 3, 'SIP Servlet Technology', is another background chapter. It helps build understanding of SIP - the signaling protocol of choice for MANETs, and SIP servlets - the de facto standard for creating, deploying and executing SIP services.

Chapter 4, ‘Distributing SIP SE for MANET’, presents the first two contributions of this thesis: a set of requirements for distributing a SIP SE for MANETs, and a distribution scheme that meets those requirements. We enumerate and discuss the nodes of the distributed SIP SE, and provide explanations on how those nodes interact with each other and with external actors. The chapter also provides rationale for the choices made when deciding on how to distribute SIP SE functionality across multiple nodes. The architecture is evaluated against all requirements that can be validated without a prototype implementation. The rest of the requirements are validated in Chapter 5.

Chapter 5, 'Prototype and Proof of Concept', presents a proof-of-concept for the distribution scheme introduced in Chapter 4. It discusses a distributed SIP SE prototype and presents two conferencing services deployed on the prototype. The prototype nodes' footprints are evaluated against the requirement that stipulates that a distributed SIP SE must be deployable on mobile, handheld devices.

Chapter 6, 'Conclusions and Future Work', provides discussions of the results and draws conclusions. This chapter also presents future work opportunities that were identified during the course of conducting this research.

# Chapter 2: Mobile Ad Hoc Networks

## 2.1 Introduction

A Mobile Ad Hoc Network (MANET) is a network formed by mobile, wireless peers without the aid of an existing infrastructure [3]. The infrastructure may be unavailable, unreliable, or non-secure. Therefore, it is up to the participating nodes to bootstrap in order to discover the nodes to directly communicate with.

The concept of ad hoc, wireless networking was introduced in the 1970 with DARPA [13] Packet Radio Project [14] aimed at providing data communication between mobile nodes on the battlefield [1]. However, it was not until recently that ad hoc became poised to become widely used in consumer applications.

The arrival of consumer ad hoc networking is a natural next step in the evolution of wireless communication [4]. With the advent of mobile phones wireless voice communication became widely available to the public. Usage of mobile phones for data communications (e.g., text messaging) followed. The next step in this evolution was to use mobile terminals to connect to the Internet. At first, mobile Internet was used to access services that had been normally accessed over fixed Internet connections. For example, professionals would access their email while on the move. However, soon after, purely "mobile" Internet applications began emerging. Advertising location-specific services for automobile drivers (e.g., locations of nearest gas stations) is a good example of such "mobile" services [4]. The next step in this evolution is to provide "anywhere and anytime" access to mobile services including in those places with no infrastructure in place.

The other factor that has made the adoption of commercial ad hoc networking possible is the availability of mobile devices that possess the characteristics necessary for ad hoc networking in terms of: usability, processing power, storage capacity and connectivity.

## 2.2 MANET Characteristics

MANETs can be considered to be a special case of wireless networks. Therefore, they share many characteristics with the wireless network in addition to some characteristics that are distinct to MANET. The rest of this section starts by discussing two radio-frequency technologies that are widely used to interconnect MANET devices: IEEE 802.11 (a WLAN technology), and the Bluetooth (a PAN technology). These technologies will serve as examples to in our discussion of MANET characteristics in the rest of this section. MANET characteristics are discussed starting from subsection 2.2.3.

### 2.2.1 Wireless LAN – IEEE 802.11e example

WLAN technologies provide connectivity within local areas such as campuses. Usually, the coverage area does not exceed 100m. An example of a WLAN technology is IEEE 802.11 [1] (also known as "Wi-Fi"). IEEE 802.11 is specified in a set of standards defined by the IEEE Wireless Local Area Networks Working Group. The concept of a "basic service set" (BSS) is central to the IEEE 802.11 architectures. A BSS can be viewed as a coverage area within which a set of wireless nodes is located. Each two nodes within a BSS should be able to establish communication between each other. IEEE 802.11 supports two modes of communication: infrastructure mode and infrastructureless (ad hoc) mode.

Infrastructure mode of communication occurs when an Access Point (AP) node is used either to form an extended service set (ESS) by interconnecting multiple BSSs, or to provide centralized access to a service (e.g., a Hot-Spot Internet access), as shown in Figure 2.1.

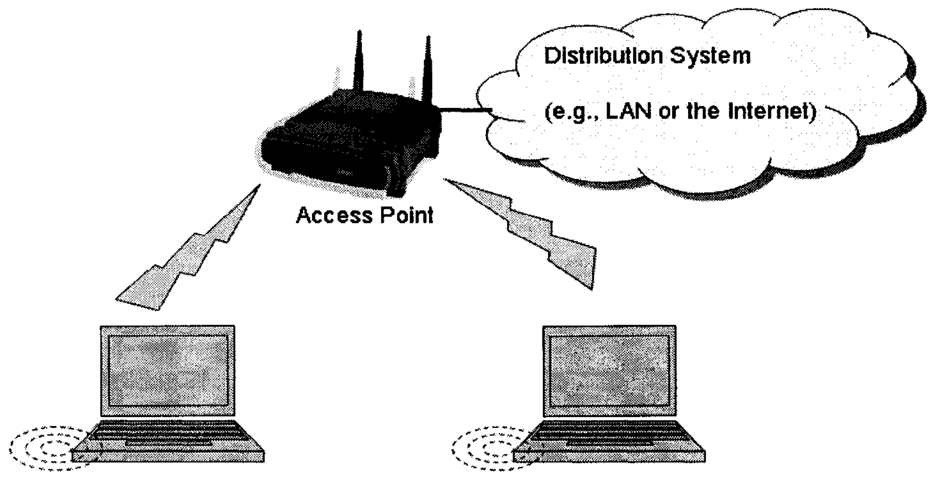


**Figure 2.1: An example of infrastructure mode of communication with IEEE 802.11**

Ad hoc mode is established when communication between all nodes in a BSS can occur directly without having to pass all the traffic through a centralized AP, as shown in Figure 2.2. In IEEE 802.11 terminology ad hoc mode is also referred to as *independent BSS* (IBSS) mode.

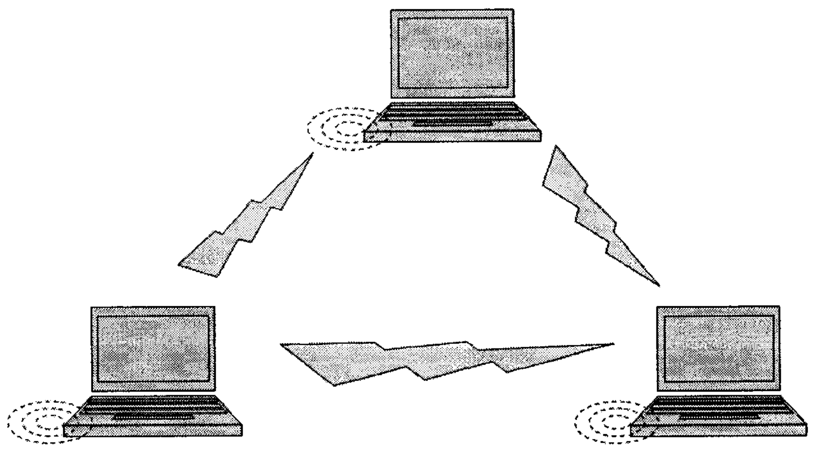


**Figure 2.2: An example of ad hoc (IBSS) mode of communication with IEEE 802.11**

## 2.2.2 Wireless PAN – Bluetooth example

Bluetooth is a PAN technology that provides short-range connectivity between devices [15]. Even though some Bluetooth device classes support communication of up to 100m, the most common use for this technology is to replace cable between devices located within proximity of up to 10m. One of the main advantages of Bluetooth is its low power consumption. This feature makes Bluetooth ideal for handheld devices operating on batteries.

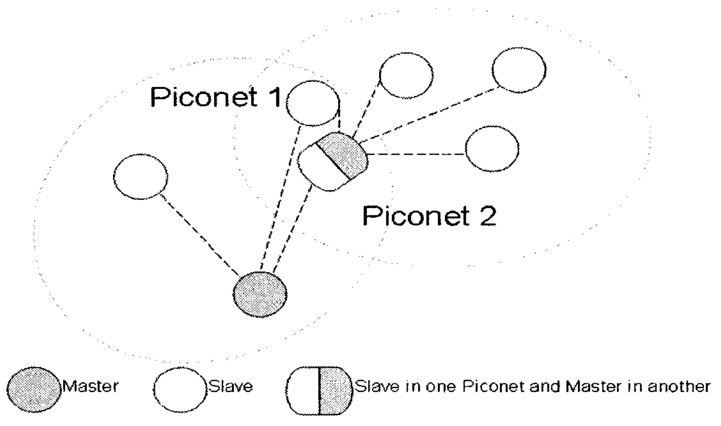


**Figure 2.3: An example of a Bluetooth scaternet with two piconets**

In order to communicate, two or more devices form a piconet (as shown in Figure 2.3). Piconet participants share the same radio channel (i.e., have the same frequency hopping scheme) and are synchronized to the same clock. Every piconet has exactly one master and up to seven active slaves. The master provides a synchronization point for the piconet. Two or more piconets may become connected to form a scaternet. A node can be a master in only one piconet. Transmission in a piconet can occur only between a master and one slave at a time, based on slotted time-division duplex. A slave is not allowed to transmit unless polled by a master. It should be noted that participants may change roles; a master can become slave and the other way around.

### 2.2.3 Ad Hoc formation and temporary service life

MANETs are formed on the fly as participant nodes come into contact with each other. They remain operational for limited periods of time as long as they cater for the, usually highly specialized, needs of the applications they were created to serve.

This automatic and temporary formation contrasts with the pre-planned, permanent nature of infrastructure wireless networks. Deployment of cellular networks involves careful planning of the network topology and base-station locations. This also applies to AP-based WLAN networks where a location for the AP has to be chosen before the network is deployed.

### 2.2.4 Infrastructure-less with Multi-Hop routing

Most wireless networks currently deployed are either cellular networks or WLAN networks. In the case of the cellular networks, the radio coverage is provided by pre-defined base-stations. In the case of WLAN, centralized access is provided by access-points (APs) that are stationary [1] (e.g., mounted on ceilings). As mentioned in section 2.2.1, WLAN can be utilized in both infrastructure and infrastructure-less networks. On the other hand, MANETs are infrastructure-less. MANET participants bootstrap to discover nodes to directly communicate with. In addition to communicating directly with adjacent nodes, a MANET participant may act as an intermediate node that relays traffic on its way from a source to a destination on behalf of other nodes. This enables multi-hop communication with the intermediate node referred to as a *router*. On the other hand, a node is said to be an ad hoc *host* if it is either a source or a destination for traffic. It is possible for a node to be both a host and a router [16].

### 2.2.5 Limited coverage range between adjacent hops

In terms of connectivity and coverage range MANETs fall under the category of LAN and PAN. The distance between two adjacent nodes is short and does not, usually, exceed the line of sight. For example, in Bluetooth the typical coverage range between two adjacent nodes is 10m. The maximum Bluetooth coverage range does not exceed the 100m. Extending coverage lengths requires more powerful transducers. This leads to higher power consumption and places limitations on the battery life of small mobile devices. Therefore, it is only possible for two MANET devices located out of transmission range of each other to communicate, if intermediary nodes that can act as MANET routers exist.

### 2.2.6 Decentralized Management

In cellular world subscriber and service management is highly-centralized (controlled by the operator). In WLAN infrastructure mode, a common way to achieve centralized management is by using wireless Hot-Spot gateways. Those gateways allow the provider to impose rules that control the way the access is provided (e.g., track the time a user spends being connected). On the other hand, MANET participants do not have any predefined roles. The roles are defined when the communication is established. Those roles are flexible and may change during the course of communication in order to adjust to changes in network conditions (such as disappearance of a node playing a certain role). As an example of flexible roles, consider a Bluetooth piconet; a slave in the piconet may change its role to become a master. Furthermore, a node may simultaneously play a role of a master in one piconet and that of a slave in all other piconets the node belongs to.

### 2.2.7 Standalone vs. Integrated

One of the motivations for commercial ad hoc networking is to extend network coverage to places where no coverage exists. In this scenario one or multiple nodes are connected to both a MANET and an infrastructure network. Those nodes act as gateways between the two networks enabling access to the fixed network by those participants that are only connected to MANET. In this case, the MANET is considered to be "integrated". On the other hand, a MANET that is isolated from infrastructure networks is considered to be "standalone". This thesis is mostly concerned with standalone MANETs.

### 2.2.8 Dynamic topologies

The topology of a MANET is constantly changing as new participant nodes join the MANET while existing participants disappear. The absence of fixed links results in sporadic connectivity between the peers. This, in turn, results in frequent disconnects and reconnects taking place.

Connectivity in MANET is also affected by the fact that MANET participants are wireless mobile devices. Node mobility and the fact that wireless interfaces are inherently lossy result in unpredictable delays, packet losses and disconnects.

### 2.2.9 Resource Constraints

MANET devices are typically equipped with limited capabilities. Small, handheld devices have limited processing power and storage capacity. They can remain operational for a limited amount of time determined by battery capacity. Another limitation is their transmission capabilities in terms of both transmission speed and range. Device capabilities place limitations on the services that a device can support and the roles a device can play. For example, a device may not be suitable for being an ad hoc router, because this results in higher energy consumption and therefore reduces its operational time. In addition to device limitations, the wireless nature of links in MANET adds limitations of its own. Wireless links have smaller bandwidths, slower transmission speeds, and limited coverage areas. Furthermore, wireless links pose considerable challenges on communication security because wireless interfaces are accessible to attackers.

### 2.2.10 Heterogeneity

Wide differences exist between MANET participant devices. First, those devices have varying processing and storage capabilities. They may also be manufactured by different vendors, connected using different interfaces and have different display sizes.

Service delivery platforms should be able to detect the characteristics of participant nodes, and adjust service delivery accordingly. For example, transmission speed may have to be slowed down to prevent overflowing a slower recipient.

### 2.2.11 Scalability

The nature of some ad hoc applications may require support for a large number of participant nodes. An example of this is a tactical network with a large number of combatants. Also, in a sensor network, hundreds of thousands sensor nodes may be present at any given time [17]. A MANET used for these types of applications should be able to scale up to such a large number of participants.

## 2.3 MANET Applications

Historically, MANET applications have been in the domain of the military and emergency services. This section starts by examining military application of MANET. Then, a recently emerging, specialized type of MANET – the sensor networks is examined. MANET applications have not yet established a wide presence in the consumer domain. However, the last part of this section provides examples of the possibilities MANET applications have to offer in the consumer domain.

### 2.3.1 Tactical Networks

Tactical networks where the first applications of MANET. They were introduced by the DARPA Packet Radio Project (PRNet) [4].

Communication on the battlefield cannot rely on a fixed infrastructure because it may not exist or can be compromised by the enemy. Radio communication is prone to interference and jamming. Increasing the communication frequency beyond radio frequencies may limit the range of adjacent communicating devices to the line-of-sight. The PRNet introduced a decentralized architecture that consisted of networks of broadcast radios. Network nodes were able to act as ad hoc routers enabling multi-hop communication and, therefore, vastly enlarging the network coverage geographically. Later projects have improved on such aspects as: scalability, energy consumption, security and adaptability.

Early DARPA scenarios involved devices that were either carried by the soldiers or mounted on tanks or ambulances [3]. The goal was to improve coordination on the battlefield as well as to assist rescue and recovery effort. In the future, MANET devices will be carried by unmanned aerial vehicles (UAV) to facilitate intelligence gathering and coordinate operations among large teams.

### 2.3.2 Sensor Networks

Sensor Networks [17] consist of a number of sensor nodes spread over a geographical area that collectively monitor environment conditions (such as air temperature) or objects (such as an object's position, orientation, velocity or functional state).

Each sensor node consists of three units.

1. Sensing unit: to measure the sensed condition or monitor an object.
2. Processing unit: process raw data by performing simple computations on the data in order to determine the portion of the data that should be transmitted.
3. Wireless Transducer unit: sends and receives data.

Sensor networks have a wide range of applications ranging from the military (e.g., measuring radiation levels) to the supermarket (e.g., measuring product inventories).

### 2.3.3 Emerging Commercial Applications

Commercial MANET applications are yet to realize. However, several usage scenarios can be contemplated [4]. For example, offering location based services like advertising the location of a nearest café. Another example is sending a message to a user's mobile terminal offering a discount on merchandise from a store the user is passing by. Multimedia conferencing sessions (e.g., gaming, interest matching) in open air settings or in places like exhibition centers or airports can be offered as well.

## 2.4 Conclusions

This chapter presented an introduction to the domain of MANET. It explained what a MANET is and outlined the evolution of wireless communication that led to the introduction of MANET technology into consumer domain. Two examples of wireless technologies were presented to assist in discussion of MANET characteristics. Then, MANET characteristics were enumerated and explained. Finally three types of MANET applications were discussed.

# Chapter 3: SIP Servlet Technology

This section provides an overview of SIP servlet technology. It starts by looking at the Session Initiation Protocol (SIP) and the features that make SIP suitable as a signaling protocol for multimedia sessions. Then, an overview of Java servlet technology is presented. Then, HTTP and SIP servlet technologies are explained and contrasted. A review of the state-of-the-art in distributable servlets follows. Finally, concluding remarks on this section are presented.

## 3.1 Session Initiation Protocol (SIP)

Session Initiation Protocol (SIP) is a signaling protocol for multimedia sessions with one or multiple participants. It is used to establish new multimedia sessions, and terminate or modify existing sessions (e.g., by inviting a new participant to join an established session). The development of SIP was started within the IETF MMUSIC (Multiparty Multimedia Session Control) working group. This work resulted in the introduction of RFC 2543 [18]. Since September 1999 the IETF SIP working group was established to become responsible for SIP standardization. The current version of SIP is specified by RFC 3261 [5] which obsoletes RFC 2543. In the rest of this section we, first, discuss SIP characteristics. Then, we discuss the applicability of SIP as a service delivery platform.

### 3.1.1 SIP characteristics and entities

SIP is a client/server, application layer, text-based protocol. The structure of SIP requests and responses is based on those of HTTP. SIP also reuses many of HTTP headers. The first line of a SIP request contains the request's method, followed by the destination's URI and concluded with SIP version. The first line of a SIP response consists of: SIP version, a response code, and a descriptive phase explaining a reason for the response. In both SIP requests and responses the first line is followed by lines containing SIP headers. Then, a presence of an empty line indicates that the request or a response has no more headers. After the empty line the request's or response's body follows.

SIP does not rely on a particular transport protocol. However, support for TCP and UDP is mandatory while other transport protocols may also be supported. It is possible that during the course of communication a SIP entity may switch from one transport protocol to another. For example, a request sent over TCP may be redirected to a UDP server connection.

SIP entities can be classified as User-Agents, Servers and Registrars. The User-Agent that initiates requests is referred to as a User Agent Client (UAC). User Agent Server (UAS) is the entity that serves incoming requests. SIP servers come in three types: redirect, proxy, and registrar. A redirect server receives a request and responds with a location at which the sender of the request should try to reach the destination (see Figure 3.1 below).

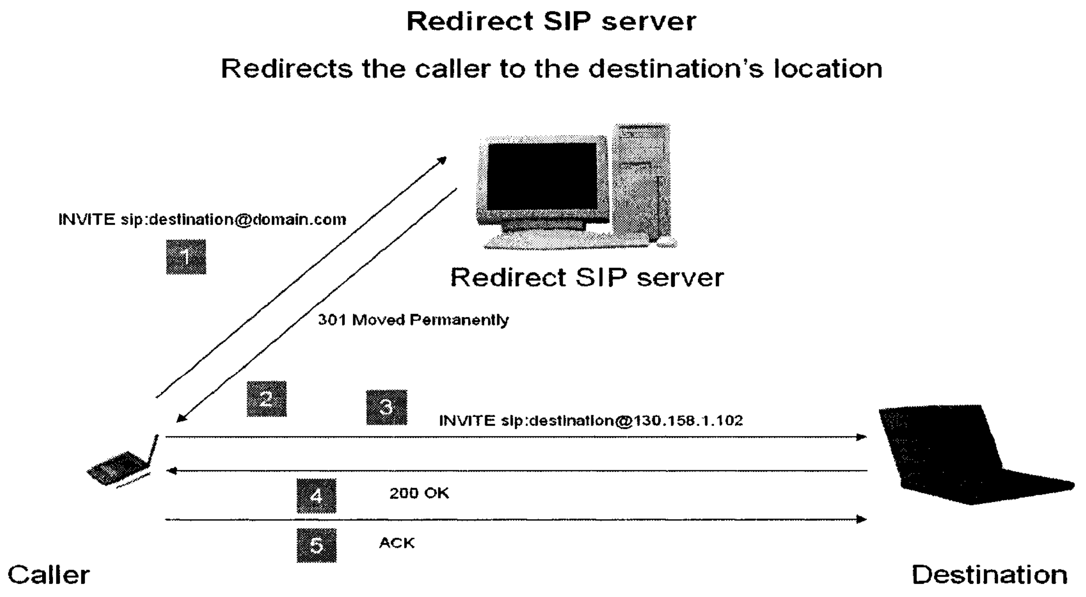

Redirect SIP server
Redirects the caller to the destination's location
INVITE sip:destination@domain.com
1
Redirect SIP server
301 Moved Permanently
2
3
INVITE sip:destination@130.158.1.102
4
200 OK
5
ACK
Caller
Destination

**Figure 3.1: Redirect SIP Server: redirects the caller to a destination's location**

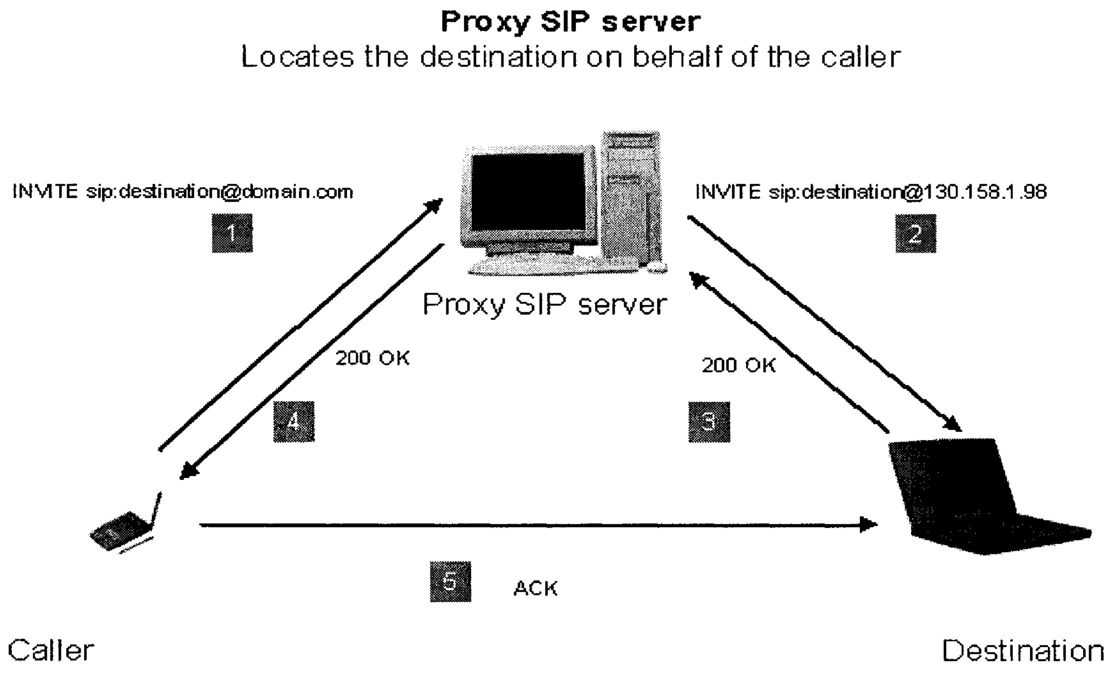

Proxy SIP server
Locates the destination on behalf of the caller
INVITE sip:destination@domain.com
1
INVITE sip:destination@130.158.1.98
2
Proxy SIP server
200 OK
200 OK
4
3
5
ACK
Caller
Destination

**Figure 3.2: Proxy SIP server: locates a destination on behalf of the caller**

A proxy server attempts to locate the destination and deliver the request to it. Figure 3.2 above shows an interaction where a proxy delivers a caller's invitation to the destination. Finally, Registrars, accept user registrations. It should be noted that the aforementioned entities may be collocated on the same device. For example, a mobile terminal would contain both UAC and UAS while a proxy server would also perform the role of a registrar.

SIP transactions go beyond the simple request/response mechanism used by HTTP. A SIP transaction includes a SIP request, zero or more provisional responses, and one final response. In the case of INVITE transactions the UAS must confirm reception of a final response with an ACK message; performing a three-way handshake.

In SIP, a peer-to-peer relationship between two user agents is referred to as a *dialog*. Dialogs allow for proper sequencing and routing of requests exchanged between the user-agents. INVITE is the only method defined in RFC 3261 that is capable of establishing dialogs. A dialog is established when the first provisional response is received by the UAC. The dialog becomes *confirmed* upon reception of a final SIP success response.

### 3.1.2 SIP as a service provisioning platform

SIP features exhibit similarities to those of the most widely used protocols of the Internet; HTTP and SMTP. SIP message structure was modeled after HTTP message structure while SIP message routing is done in a way similar to the routing of SMTP messages [8]. These similarities have multiple implications on the role SIP can play as a service provisioning platform. First, SIP can be used to combine existing web and email services with new multimedia services. Second, a vast community of developers familiar with service development for the web and email can use their skills to program SIP services with a minimal learning curve. Furthermore, existing platforms for developing services for the web can easily be adapted to become applicable to SIP service development. This leads to a reuse of existing models, techniques, even source code in providing SIP services.

Support that SIP provides for different media and session types is another feature that makes SIP highly suitable for service development. For example, if a SIP-based chat service is deployed, services based on other types of sessions (e.g., a gaming service) can be deployed without adding new SIP infrastructure.

Finally, in addition to providing services, SIP delivers services to their destinations using the concept of *indirection* [19]. Indirection means that a session initiating party does not need to know where the other participants are located. Since, SIP users have to register their locations the session initiator only needs to know the permanent SIP URIs of the other parties. Matching the permanent SIP URI to the actual location of the parties is done by SIP servers. Upon finding a match, a SIP server returns the actual location to the session initiator.

## 3.2 Java Servlet Technology

A servlet is a Java class that enables execution of protocol specific code on server side. For example, a servlet would contain logic responsible for generating a response to an incoming protocol request. Since servlets are written in Java, they are portable across operating systems.

No direct interaction exists between a servlet and a client. Instead, servlets interact with the client-side indirectly, through a Servlet Engine (SE). The SE is the entity that creates servlet class instances, provides the servlets with an execution environment, and manages servlets through their lifecycles. Furthermore, the SE performs protocol message parsing, dispatching incoming messages to servlet applications, providing storage for state information of servlet applications, and enforcing security constraints specified during application deployment.

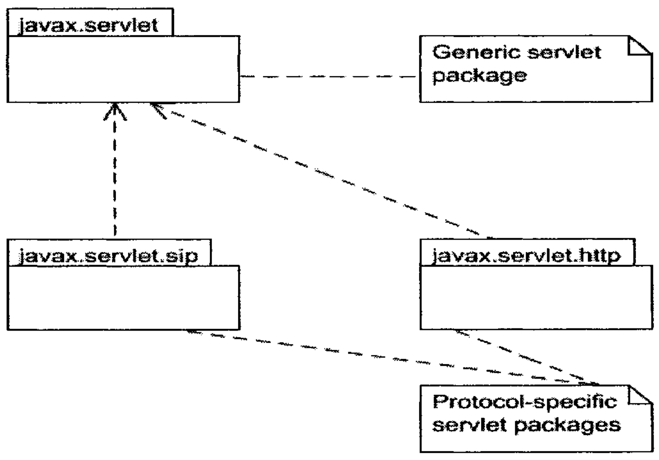


**Figure 3.3: Servlets package diagram**

Rules for interactions between a servlet and an SE are specified by a Servlet API. A generic part of the Servlet API specifies interactions that are not specific to a particular protocol.

This generic part is extended to provide protocol-specific Servlet APIs (see Figure 3.3). Generic Servlet API classes are located under `javax.servlet` Java package. Protocol-specific Servlet API classes for a protocol *xyz* are located under `javax.servlet.xyz` Java package.

The servlet lifecycle phases are shown in Figure 3.4 below (taken from [9]).

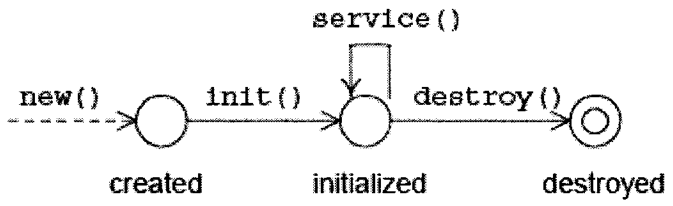


**Figure 3.4: Servlet Lifecycle**

First, the SE loads the servlet class from a repository [20]. Then, the SE calls the servlet's `init()` method. The `init()` method lets servlets perform one-time initializing activities (such as obtaining database connections) and loading persistent configurations. Initialization must complete successfully before any messages are dispatched to the servlet for processing.

After a servlet is initialized the SE will make repeated calls to servlet's `service()` method every time an incoming protocol message should be served by that servlet. The `service()` method contains the logic used for processing the protocol message.

When the servlet should be phased-out the SE calls `destroy()` method of the servlet. This, usually, occurs when the SE itself is being shutdown or when it is low on resources.

Inside the `destroy()` method clean-up is performed and resources are released in a proper manner.

In the rest of this section we present and contrast the two servlet technologies widely deployed today; HTTP servlet and SIP servlets.

## 3.3 HTTP Servlets

An HTTP servlet generates HTTP responses to incoming HTTP requests. This type of servlets has been the focus of servlet development so far in order to generate dynamic web content. Therefore, HTTP servlets can be viewed as a Java alternative to CGI [32] scripts and Active Server Pages (ASP) technologies [33].

The means of communication between HTTP Servlets and an HTTP SE are defined in the Java Servlet API as specified by the "Java Servlet Specification" [11]. The latest version of the API is version 2.4.

One or multiple HTTP servlets are bundled together to form a *web application*. Web applications reside in a hierarchical directory structure on the HTTP SE. The location of each web application relative to the "root" is referred to as the *context path* of the application.

The interaction between the HTTP SE and the client commences when the SE receives an HTTP request from the client. Then, the SE decides which servlet application gets to serve the incoming request. This is achieved by matching a portion of the request URI against the *context path* of all servlet applications running on the SE. The SIP SE dispatches the request to the servlet with the longest matching context path. The servlet extracts useful information from the received request's headers and body, processes this information and generates an HTTP response. After the response is generated, it is possible for the servlet to add new headers to the response, or modify values of existing response headers. The servlet may also attach content to the response body. Usually, the content of the response body is an HTML page. However, other contents such as XML may be carried in the HTTP response body. Finally, the servlet instructs the SE to deliver the response to the client. HTTP servlets never initiate requests; but only respond to incoming HTTP requests with exactly one HTTP response.

HTTP Servlet API specifies the interactions between an HTTP SE and an HTTP servlet. HTTP Servlet API classes are located under the `javax.servlet.http` Java package. HTTP Servlet API provides an indirect implementation of the `javax.servlet.Servlet` interface using an `HttpServlet` abstract class, as shown in Figure 3.5. `HttpServlet` overrides the `service()` method to dispatch an incoming HTTP request to a method specialized in serving that request type. The specialized methods' names have the form `doXXX()` where `XXX` is replaced by the HTTP request's type. For example, an HTTP GET request is dispatched to `doGet()` method. Default implementations of the `doXXX()` methods do not perform any action. They are overridden by the actual HTTP servlet implementations to provide the logic used for request processing.

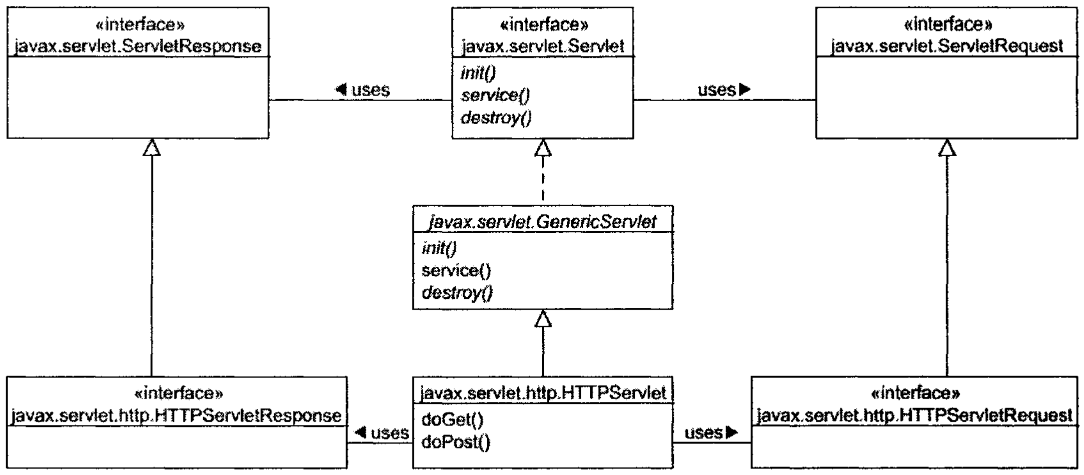

«interface»
javax.servlet.ServletResponse
◀ uses
«interface»
javax.servlet.Servlet
init()
service()
destroy()
uses ▶
«interface»
javax.servlet.ServletRequest
javax.servlet.GenericServlet
init()
service()
destroy()
«interface»
javax.servlet.http.HTTPServletResponse
◀ uses
javax.servlet.http.HTTPServlet
doGet()
doPost()
uses ▶
«interface»
javax.servlet.http.HTTPServletRequest

**Figure 3.5: HTTP servlet class diagram**

HTTP Servlet API also provides HTTP-specific interfaces to represent HTTP requests and responses: `HttpServletRequest` and `HttpServletResponse`. The methods of the `HttpServletRequest` provide methods to obtain HTTP specific information from an HTTP request. For example, they allow a caller to obtain the URI used by the client to make this request by calling `getRequestUri()` or to get all the cookies contained in the request by calling `getCookies()`. The methods in the `HttpServletResponse` allow the caller to modify the response (e.g., by setting response status or adding a header), or perform an HTTP action such as returning an HTTP error or a redirect to the client. It should be noted that the `HttpRequest` does not provide methods that perform actions beyond getting information from the request. On the other hand most methods of the `HttpResponse` are used either to modify the response or to perform an HTTP protocol action (such as sending a redirect to the client). This is due to the fact that HTTP servlets never generate requests; but only reply to an incoming request with exactly one response.

HttpSession interface provides servlets with means to maintain state information across multiple requests. The state information is stored in HTTP sessions as a collection of name/value pairs. Each HTTP session is identified by a unique id.

## 3.4 SIP Servlets

A SIP servlet is a Java class that performs SIP signaling logic and runs in an environment provided by a SIP SE. The interactions between SIP servlets and SIP SE are defined in "Sip Servlet API specification, Version 1.0" document [9].

The introduction of SIP servlet technology was motivated by the need for easy creation, customization and deployment of SIP services [21]. Extending Servlet API with SIP capabilities was feasible due to the success of HTTP servlets in delivering dynamic web content in addition to the fact that HTTP and SIP share many similarities.

One or multiple SIP servlet classes are bundled together to form a *SIP application*. A SIP application may contain multiple servlets in addition to other resources, for example, non-servlet java classes and text files. The information on how an application is structured is conveyed in the application's *deployment descriptor*. The deployment descriptor also provides mapping rules that a SIP SE uses to determine the servlet that gets to execute an incoming initial request. This mapping is based on a *rule language* defined in [9].

SIP Servlet API enables SIP servlets to perform a variety of SIP signaling tasks while, at the same time, hiding the cumbersome details of the SIP protocol. For example, servlet developers have neither to worry SIP message retransmissions nor maintaining sequencing information in CSeq headers. The SIP Servlet API simply does not provide servlets with access to these functionalities. The SIP Servlet API also guarantees that servlets cannot perform actions that would violate SIP protocol specifications.

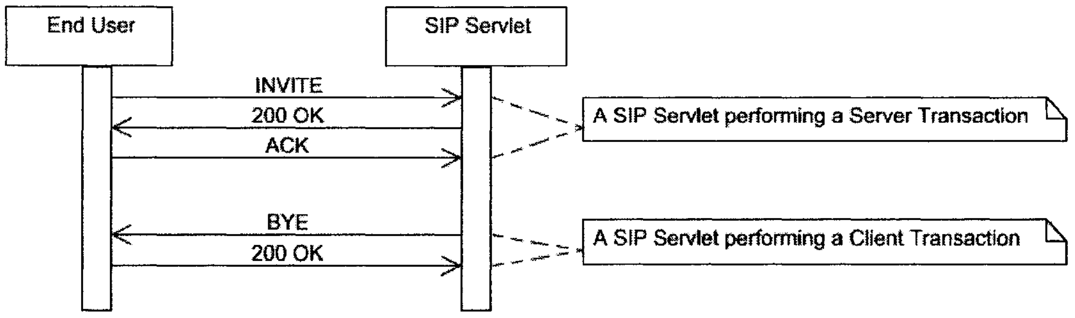

End User
SIP Servlet
INVITE
200 OK
ACK
BYE
200 OK
A SIP Servlet performing a Server Transaction
A SIP Servlet performing a Client Transaction

**Figure 3.6: SIP Servlet acting as both a transaction server and client**

An important feature of SIP servlets is that within a SIP transaction a servlet may assume either the role of a server or that of a client (see Figure 3.6). A SIP transaction consists of: a request, a number of provisional responses, and one final response. For any given transaction the party that sends the request becomes the client for that transaction. SIP servlets have the capability to act both as transaction clients and servers. A SIP servlet is able to respond to an incoming SIP request with zero, one or several responses. A SIP servlet may also proxy an incoming request to one or several destinations. Furthermore, a SIP servlet may generate new SIP requests on its own.

As an example of a service created using a SIP servlets, consider the case of an Outgoing Call Screening service. In this scenario, the operator forbids calls outgoing to destinations in "forbidden.com" domain. The listing for the servlet that implements the Outgoing Call Screening service is shown in Figure 3.7.

```
import javax.servlet.*;
import javax.servlet.sip.*;

public class OutgoingCallScreeningServlet extends SipServlet {

  public void doInvite(SipServletRequest req)
  throws ServletException, java.io.IOException {

    // Obtain the destination's address as a string.
    String to = req.getTo().getURI().toString();

    if (to.contains("forbidden.com")) {
      // Destination forbidden.
      // Send a 403-Forbidden response back to the sender.
      SipServletResponse res = req.createResponse(403);
      res.send();
    } else {
      // Destination allowed.
      // Send the message to the destination.
      req.send();
    }
  }
}
```

**Figure 3.7: Listing for OutgoingCallScreeningServlet**

The `doInvite()` method checks if the destination address contains a string "forbidden.com" to find whether the destination belongs to "forbidden.com" domain. If the destination belongs to "forbidden.com" domain a 403-Forbidden response is sent back to the sender. Otherwise, the message is sent to its destination.

### 3.4.1 API

SIP Servlet API specifies the interactions between a SIP SE and a SIP servlet. SIP Servlet API classes are located under the `javax.servlet.sip` Java package.

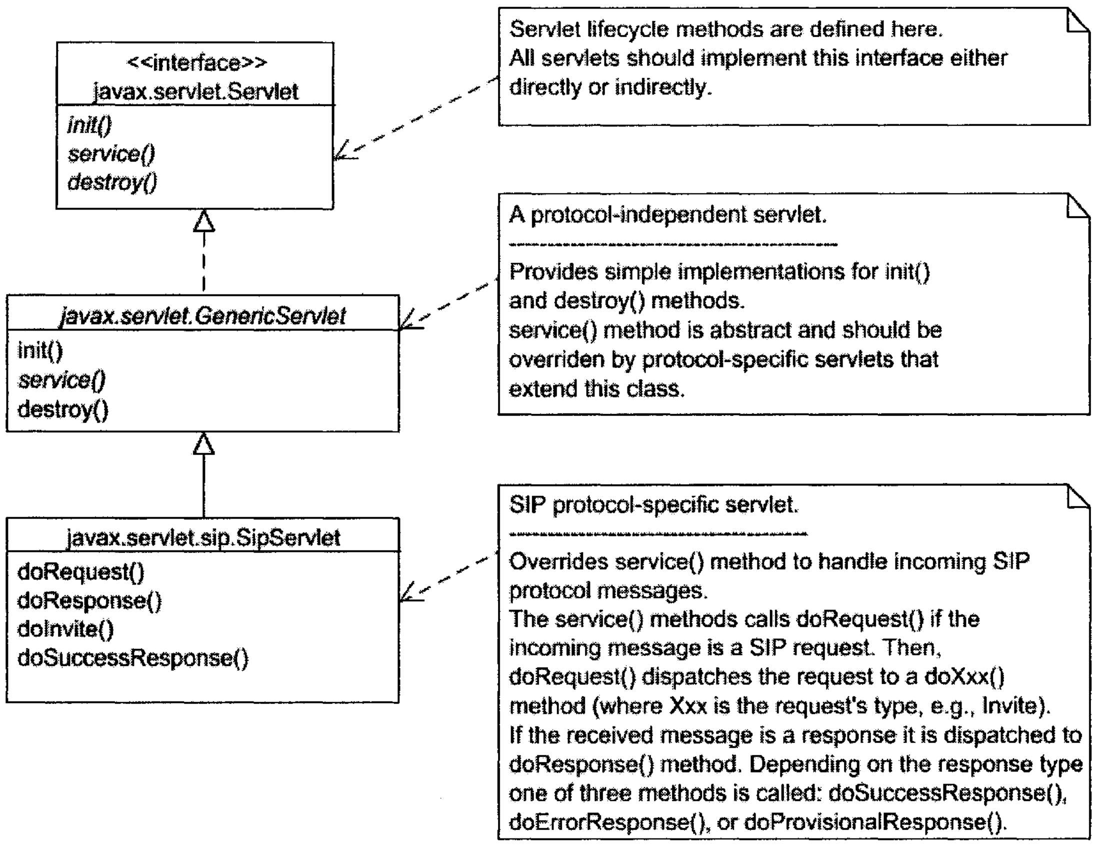

<<interface>>
javax.servlet.Servlet
init()
service()
destroy()
Servlet lifecycle methods are defined here.
All servlets should implement this interface either directly or indirectly.
javax.servlet.GenericServlet
init()
service()
destroy()
A protocol-independent servlet.
Provides simple implementations for init() and destroy() methods.
service() method is abstract and should be overriden by protocol-specific servlets that extend this class.
javax.servlet.sip.SipServlet
doRequest()
doResponse()
doInvite()
doSuccessResponse()
SIP protocol-specific servlet.
Overrides service() method to handle incoming SIP protocol messages.
The service() methods calls doRequest() if the incoming message is a SIP request. Then, doRequest() dispatches the request to a doXxx() method (where Xxx is the request's type, e.g., Invite).
If the received message is a response it is dispatched to doResponse() method. Depending on the response type one of three methods is called: doSuccessResponse(), doErrorResponse(), or doProvisionalResponse().

**Figure 3.8: SIP Servlet Class Hierarchy**

SIP Servlet API provides an indirect implementation of the `javax.servlet.Servlet` interface using a `SipServlet` abstract class (as shown in Figure 3.8). `SipServlet` implements the `service()` method of the `Servlet` interface to dispatch an incoming SIP message (whether request or response) to a method specialized in serving that message type. The specialized methods' names have the form `doXXX()` where `XXX` is replaced by the SIP message type. SIP Servlet API also provides SIP-specific interfaces to represent SIP messages; `SipServletMessage` and its two sub-interfaces: `SipServletRequest` and `SipServletResponse` (as shown in Figure 3.9).

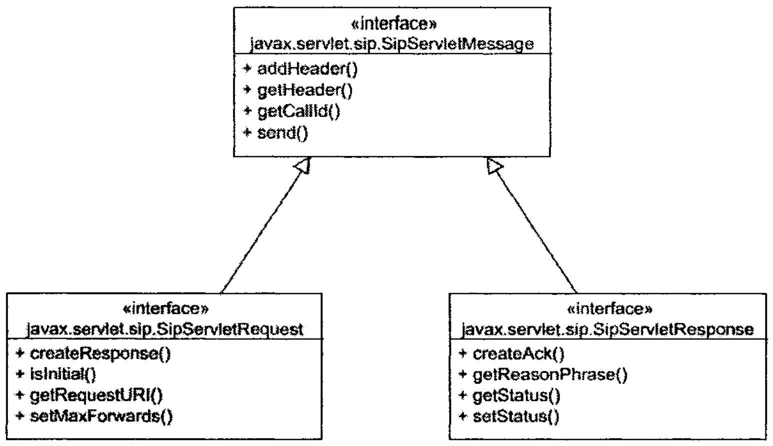


**Figure 3.9: SIP Servlet Message Hierarchy**

The reason for providing a `SipServletMessage` interface as a parent for the other two interfaces is the fact that SIP servlets have the ability to send requests in addition to receiving them. In HTTP Servlets information is read from HTTP requests and written into HTTP responses. In SIP servlets both requests and responses are readable as well as writable. This means that `SipServletRequest` and `SipServletResponse` have a lot of common methods. Those common methods are placed in `SipServletMessage` interface.

SIP Servlet API provides two types of sessions to maintain state information for a SIP application. First, `SipSession` provides means for storing state information for each point-to-point SIP relationship. Therefore, SIP sessions sometimes correspond to SIP dialogs. However, a SIP session is created before a SIP dialog is created. Furthermore, only certain SIP messages are capable of creating a dialog, while all SIP messages processed by a SIP SE belong to a SIP session. Second, a `SipApplicationSession` stores state information for a SIP application as a whole (including all SIP session that are created by the application).

## 3.4.2 SIP SE

A SIP SE is the entity that manages SIP servlets through their lifecycle and provides the servlets with other services such as taking care of cumbersome protocol details and security. The SE takes care of system programming tasks that are needed by the servlet applications but are not part of the application's logic. This enables servlet developers to focus on their area of expertise, namely, service application logic programming.

SIP SEs come as standalone entities or as extensions to existing servers. The discussion in this report applies only to standalone SIP SEs. Engines that come as part of J2EE, or any other type of servers are outside the scope of this thesis due to having additional requirements (e.g., allowing servlets to obtain references to enterprise java beans).

The architecture of a SIP SE is depicted on Figure 3.10 below.

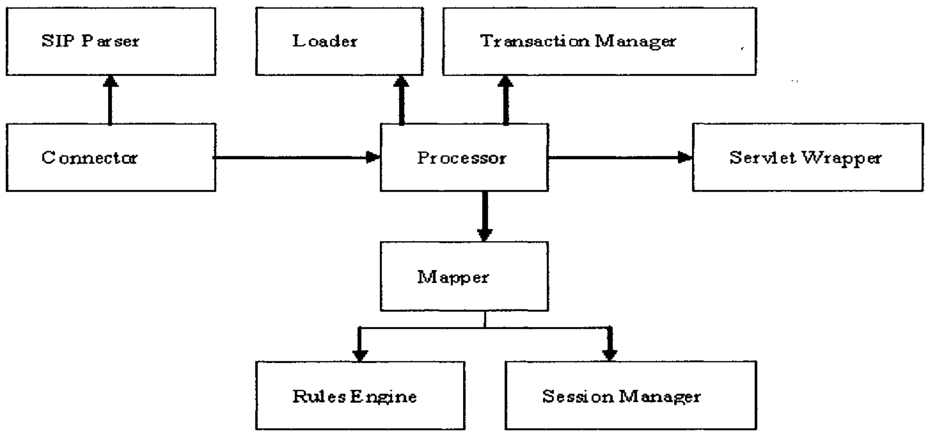
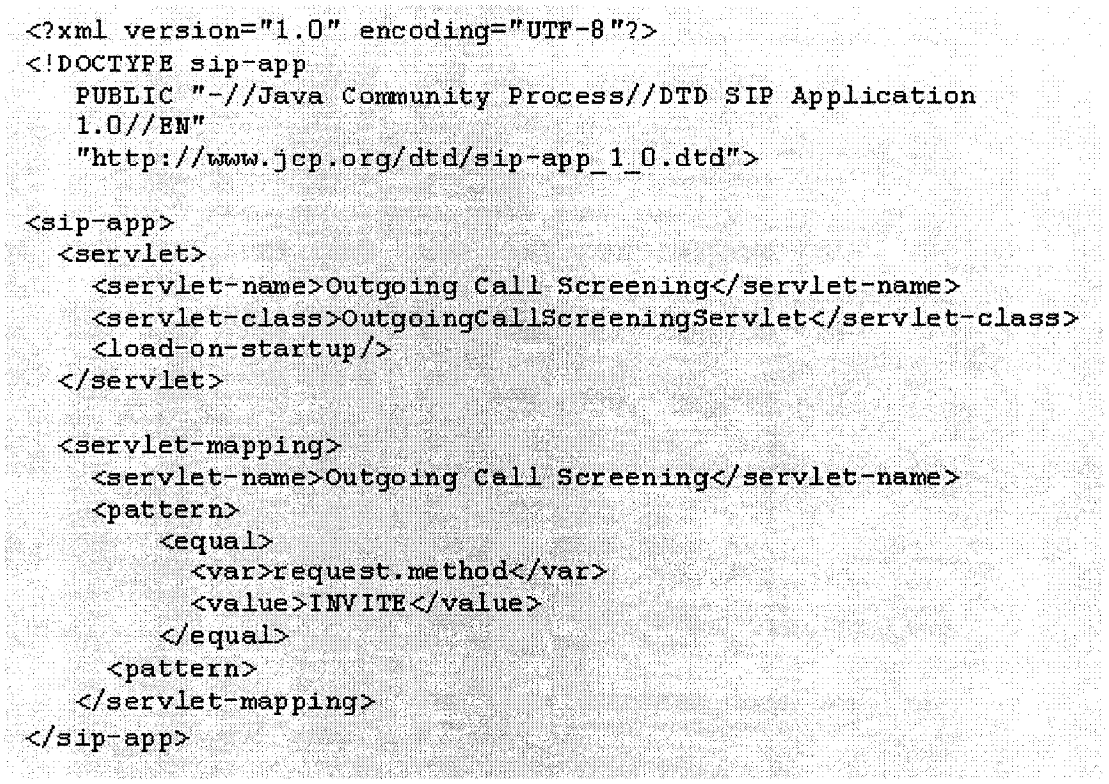


**Figure 3.10: SIP SE Architecture**

The Connector is the part responsible for sending and receiving SIP messages over the network. The Connector uses SIP Parser to decode an incoming SIP message from bytes into a `SipServletMessage` object. Inversely, outgoing SIP messages are encoded from a `SipServletMessage` object into a byte array by the SIP Parser.

The Processor is the central unit that coordinates the other SIP SE components. The Processor instructs the Loader to load the servlet classes from the repository where they are stored. After the servlet is loaded the Processor creates a Servlet Wrapper for the servlet. One wrapper is created for each servlet running on the SIP SE. During the lifecycle of the servlet the Processor will instruct the Servlet Wrapper to call init(), service(), destroy() methods of the servlet. Also, upon receiving a SIP message the Processor queries the Mapper on what servlet should service the incoming message.
Finally, the Processor handles deployment descriptors and enforces security policies.
The Mapper is responsible for the routing of SIP messages. Here, the term routing means: deciding what servlet should service the incoming message. Routing is done differently depending on whether the request is an initial or a subsequent request. A request is said to be an initial request if it is not part of an existing SIP dialog. The routing of initial requests is done by the Rules Engine. This kind of routing is based on the rules defined in the deployment descriptors. Figure 3.11 below shows a complete listing of a SIP SE deployment descriptor. The descriptor defines an application that consists of the servlet listed in Figure 3.7.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app
   PUBLIC "-//Java Community Process//DTD SIP Application
   1.0//EN"
   "http://www.jcp.org/dtd/sip-app_1_0.dtd">

<sip-app>
  <servlet>
    <servlet-name>Outgoing Call Screening</servlet-name>
    <servlet-class>OutgoingCallScreeningServlet</servlet-class>
    <load-on-startup/>
  </servlet>

  <servlet-mapping>
    <servlet-name>Outgoing Call Screening</servlet-name>
    <pattern>
        <equal>
          <var>request.method</var>
          <value>INVITE</value>
        </equal>
     <pattern>
   </servlet-mapping>
</sip-app>
```

**Figure 3.11: Deployment descriptor example**

When a SIP SE receives a SIP request it needs to decide which servlet should handle the incoming request. In order to make this decision the SIP SE refers to the *servlet-mapping* rules listed in the deployment descriptor of each SIP application.

The deployment descriptor in Figure 3.11 defines a SIP servlet application that consists of one servlet. The servlet is loaded at SIP SE startup. The servlet-mapping rule states that the criterion to invoke the OutgoingCallScreeningServlet is the request being of type INVITE. Therefore, all arriving INVITE requests will be handled by the OutgoingCallScreeningServlet. The routing of subsequent messages is done by the Session Manager and is based on the SIP dialog-id.

SIP Transaction Manager handles timeouts and SIP message retransmissions. It correlates a SIP request with all its responses.

Servlet Wrapper creates an instance of the servlet it “wraps” and manages the servlet through its lifecycle.

## 3.5 Comparison between HTTP and SIP servlets

The basic model of HTTP servlet execution proceeds according to the following sequence. First, an HTTP request arrives. Then, the HTTP SE decides on which servlet should serve the incoming request. Then, the HTTP SE calls `service()` method of that servlet. Finally, when the `service()` method returns the HTTP SE sends an HTTP response back.

The aforementioned sequence generally applies to SIP servlets except for the fact that this sequence is triggered by incoming responses in addition to incoming requests. Furthermore, SIP SE does not send a response back to the caller when `service()` method returns. Instead, responses are sent back during the course of `service()` execution; whenever the servlet instruct the SE to generate and send a response. In addition to generating responses a SIP servlet may also proxy a received request or generate and send a new request on its own.

While HTTP servlets act as extensions to web servers SIP servlet may play a larger variety of roles as: User Agents, Proxy and Redirect servers, Registrars and Back-to-back-user-agents.

| | **HTTP** | **SIP** |
|---|---|---|
| **Purpose** | Generating dynamic web content. | Performing SIP signaling logic. |
| **Specification document** | "Java Servlet Specification, Version 2.4" [11] | "Sip Servlet API specification, Version 1.0" [9] |
| **Servlet roles** | Extending HTTP server capabilities | Multiple possible roles:<br>- User Agent<br>- SIP server (Redirect, Proxy, or Registrar)<br>- Back-to-Back-User-Agent |
| **Events that trigger servlet execution** | Incoming HTTP request only | Incoming SIP request or response |
| **Action in response to a servlet triggering event** | Generate and send exactly one HTTP response. | - Generate zero, one, or multiple SIP responses.<br>- Generate zero, one, or multiple SIP requests.<br>- Proxy an incoming request to one or multiple destinations. |
| **Request routing** | Based on portions of request URI. | Based on rules defined in the deployment descriptor. |
| **State information storing** | State stored using: HTTP Session objects, URL rewriting, and Cookies. | State stored using: SipSession objects, and SipApplicationSession objects |

**Table 3.1: Summary of differences between HTTP and SIP servlets**

An HTTP SE extracts information on what servlet should execute an incoming request by using a portion of the request's URI. On the other hand, a SIP SE routes incoming

requests by matching certain attributes of the request (e.g., its method, source, destination, etc.) against a set of rules defined in the application's deployment descriptor.
Session management in HTTP servlets is performed in order to overcome the stateless nature of HTTP protocol by maintaining state across multiple HTTP transactions performed by the same user. State management in SIP servlets is more complex. First, a SIP session exists per each point-to-point relationship between two SIP user agents. Second, SIP servlets introduce the notion of application sessions that store state information for the application as a whole. A SIP application session may include multiple SIP sessions and, in the case of an integrated SIP/HTTP SE, a multiple number of HTTP sessions as well. Difference between HTTP and SIP servlets are shown in Table 3.1.

## 3.6 Distributable Servlets

Java Servlet Specification [11] allows the SEs to be distributed across multiple Java Virtual Machines. Some of those JVM may be running on the same host while others may be running on remote hosts.
This section summarizes the additional requirements imposed when an application is marked as being distributable. Some of those requirements are placed on the SE while others concern the servlet applications.
A distributable servlet may run on multiple JVM. Each JVM may host one instance of a servlet except for servlets implementing `SingleThreadModel` interface (the case when the SE must ensure that only one thread at a time has access to a servlet instance). In this case the SE may initiate a pool of servlet instances to enhance performance.

In distributed applications one servlet instance may process an initial request while other instances of the servlet may process subsequent requests and responses. Therefore, developers can neither depend on static nor on instance variables of the servlet class to store the state of the application. For this purpose, sessions or databases should be used instead.

Servlets running in the context of the same application have access to shared resources. Those resources are accessed through `ServletContext` interface, which provides a servlet with a view of the application that encloses it. In distributed environments servlet contexts are confined to the scope of a JVM. Therefore, in distributed environments, `ServletContext` should not be used to store state information that needs to be shared between servlets that belong to the same context [23]. Instead, session objects or databases should be used for this purpose.

In distributable applications all requests and responses that are part of the same session must be handled by one JVM at a time. This relieves the programmer from considerations about concurrency issues involved when several requests or responses belonging to the same session are being processed.

Session objects might be stored on one node or on multiple nodes (for failover or load balancing purposes). In the latter case a mechanism ensuring that instances representing the same session and residing on multiple nodes are kept in sync with each other. Sessions may also be migrated from one node to another node on the distributed SE. They can also be stored in a database. Distributed SEs must ensure consistent access to session information by the servlets.

It follows from the discussion in this section that successfully creating servlets that are distributable is a shared responsibility. On the one hand, servlet developers should ensure that the servlet classes they produce are stateless. They should also refrain from using `ServletContext` interface for storing state information. On the other hand, servlet SEs should handle concurrency, session storage, migration, and distribution issues. As a result servlet developers are relieved from system programming tasks and can focus on service application programming instead.

### 3.6.1. Distributed Session Management

This section describes session management for scenarios where multiple SEs are running instances of the same servlet. A central entity, called a "Load Balancer" or "Load Director" [24], directs the incoming traffic to either SE. As, mentioned above, these schemes are used for failover or load balancing purposes.

Both SIP and HTTP are stateless protocols. However, real-life applications require tracking states. In WEB applications client transactions involve multiple requests. State information is stored in sessions and is modified during the course of the transaction.

In SIP a session is used to represent a SIP dialog, which is a SIP relationship between two User-Agents. Dialog ID (a combination of Call-ID, From tag, and To tag) is used to uniquely identify a SIP dialog and the corresponding SIP session.

The SE uses the information stored in SIP sessions to perform subsequent request routing. The dialog id of an incoming SIP message is matched against the dialog ids of the SIP sessions managed by the SE. The matching session can be used to provide information about the servlet that should process the request.

Next we consider two approaches discussed in [24] to session management for distributable servlets.

### 3.6.2 Fixing a dialog to the SE that handles the initial request.

In this approach the Load Director remembers which SE handled the initial request of a dialog. All subsequent messages belonging to that dialog will be routed to the same SE.

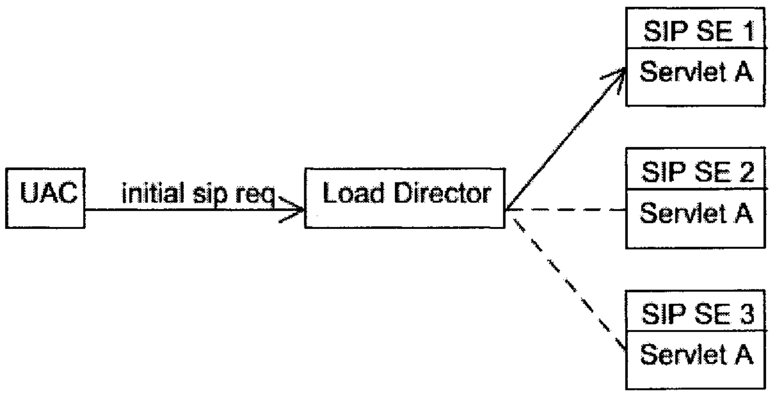


**Figure 3.12: "Sticky Sessions" technique**

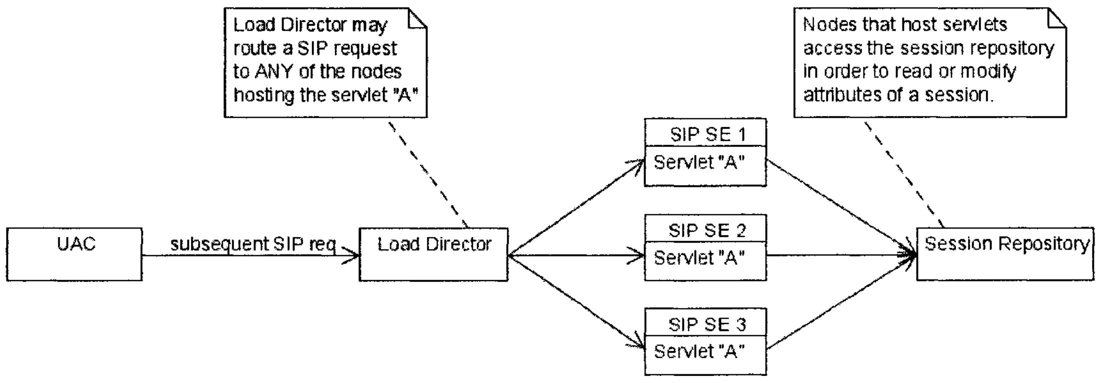
Figure 3.12 shows three SEs, each hosting an instance of servlet "A". When the initial request arrives the SE may choose any of the three nodes to handle it. In the diagram the chosen node is "SIP SE 1". Therefore, all subsequent requests belonging to the same dialog will be routed to "SIP SE 1".

The advantage of this approach is its simplicity. The Load Director routes dialog's subsequent requests to the same SE that handled the initial request. The SE running the servlet also acts as a session repository.

On the other hand, if the SE handling the dialog becomes unavailable, the session data is lost. Also, some SEs may become overloaded while, at the same time, other SEs running instances of the same servlet will be idle.

## 3.6.3 Separating session repositories from servlet hosting

In this approach, session repository is separated from servlet hosting and moved into a "central point of reference" [24] located on a different host. We will use the term "session repository" to refer the aforementioned "central point of reference". Therefore, nodes that host and execute servlets are not responsible for storing sessions. Instead, multiple hosts running instances of the same servlet may access the session repository in order to retrieve or modify session information. The location of session repository is provided to the servlet-hosting nodes by the Load Director. This approach is illustrated in Figure 3.13.



Load Director may route a SIP request to ANY of the nodes hosting the servlet "A"
Nodes that host servlets access the session repository in order to read or modify attributes of a session.
UAC
subsequent SIP req
Load Director
SIP SE 1
Servlet "A"
SIP SE 2
Servlet "A"
SIP SE 3
Servlet "A"
Session Repository

**Figure 3.13: Separating session repositories from servlet hosting**

At any given point, only one servlet instance (the one currently processing the message) can access or modify the session. After the servlet finishes processing the message, further messages of the same dialog can be served.

Since SEs do not store sessions state information the next subsequent message may be processed on a different SE. Failure in any SE hosting a servlet does not result in dialog session information loss. The drawback of this approach is that the session repository becomes the central point of failure.

## 3.7 Conclusions

In this chapter we discussed Session Initiation Protocol and the features that make it suitable as a signaling protocol for multimedia sessions. However, in order for SIP services to succeed it needs a technology that would allow for easy programmability and deployment of SIP services. SIP shares multiple similarities with HTTP. Therefore, it is feasible to apply to SIP those technologies that have proven their success in providing services in HTTP. The success of servlet technology in providing dynamic content for the web, paved the way for the introduction of SIP servlet technology. This chapter discussed and contrasted the HTTP and SIP servlet technologies. Finally, a state of the art in distributable servlets was presented. The in the next chapter we present a scheme for applying SIP servlets to MANET by distributing a SIP SE.

# Chapter 4: Distributing SIP SE for MANET

## 4.1 Introduction

In this chapter we propose an architecture for distributing a SIP SE across multiple nodes. First, we derive requirements for such an architecture. Then, we show that state-of-the-art fails to meet the derived requirements. Then, we discuss a business model [25] for service provisioning with SIP servlets in MANET. This is followed by proposing a distributed SIP SE architecture and its components. Then, we discuss mechanisms used by the nodes of the distributed SIP SE to communicate with each other and with external actors. Finally, we establish that the proposed architecture meets most of the requirements we derived. The remaining requirements are evaluated in Chapter 5 with a prototype.

## 4.2 Requirements

In order to be suitable for MANET environments a distributed SIP SE must satisfy a set of requirements. In this section we derive and explain the requirements imposed by MANET operating conditions on the distributed SIP SE.

First, distributed SIP SE components must be deployable on mobile, handheld devices with limited capabilities. This requirement stems from the fact that a typical MANET participant node is a mobile device with limited capabilities (as explained in Section 2.2.9). Therefore, the components of a distributed SIP SE should be of small size, and be light on runtime memory consumption.

Second, a distributed SIP SE must support on-the-fly service deployment. This is because the distributed SIP SE does not load services when it is formed by its component nodes. It neither stores any SIP service archives itself, nor has any knowledge of the servlet repository's location. Instead the distributed SIP SE loads a servlet application from a servlet repository [20] when it is instructed to do so. The location of the SIP service command must be provided by the issuer of that command.

Third, components of a distributed SIP SE must bootstrap to discover each other's presence. They must achieve a distributed SIP SE formation without the aid of an external node. Information on participant nodes' locations (e.g., hostnames, ports) must not be preconfigured. Instead, such information should be acquired dynamically as participant nodes come within transmission range of each other.

Fourth, changes in the network topology should, ideally, have no impact on service delivery. This requires support for redundancy. If a SIP SE node becomes unavailable the remaining nodes should attempt to keep the distributed SIP SE functional. This can be achieved either by replacing the disappeared node with a node having similar capabilities, or by allowing the service to be delivered with fewer nodes. Also, a distributed SIP SE should allow several nodes to run an instance of the same servlet. In this case, the SE should choose one of those nodes to service an arriving protocol message that should be handled by that servlet.

Fifth, the distributed SIP SE should be able to handle constant changes in connectivity characteristics. The changes in connectivity result from user mobility and from the inherent characteristics of wireless media. The SIP SE should be able to cope with unpredictable delays, packet losses and disconnects.

Sixth, the distributed SIP SE should scale well to the increase of the number of servlets deployed, the number of requests it can serve, and the number of participant nodes (e.g., if more nodes are used, say, to provide redundancy).

## 4.3 Evaluation of the state-of-the-art with regard to the requirements

Commercial SIP SEs available today are part of global application servers. Examples of such application servers are: IBM Websphere [28], and SIPMethod [29]. These application servers are representative of the ones currently available on the market. System requirements for these servers make them unsuitable for deployment over mobile devices with limited capabilities, as stated by our first requirement. For example, the minimal requirements for IBM Websphere on Windows 2000 are: 512 MB of RAM, Intel® Pentium® processor (500 MHz), and 990 MB of disk space.

The available SIP SEs support on-the-fly deployment of SIP applications, thus satisfying our second requirement.

Our third requirement that the components of a distributed SIP SE bootstrap to discover each other is not satisfied by any of the existing SIP SEs. This is because existing SIP SEs have been developed for networks with fixed, centrally managed networks. In these types of networks the need for automatic discovery of different nodes does not exist.

The fourth requirement is to tolerate changes in network topology without an impact on service delivery is not satisfied by any existing SIP SE implementations. Network topology is assumed to be stable. Redundancy was implemented for fault-tolerance and load-balancing purposes. However, these redundancy schemes use a "Load Director" node which remains a centralized entity and, therefore, cannot be used in MANET environments.

The fifth requirement (support for changes in connectivity and user mobility) is not met by existing SIP SEs because this requirement is not applicable for infrastructure networks.

Most existing SIP SEs are highly scalable, thus satisfying our sixth requirement on scalability.

## 4.4 Business Model

Business Models describe the different actors involved in service provisioning as well as their relationship to each other [30]. Once a business model is defined, it serves as a starting point for identifying interfaces between the actors and specifying the service architecture.

The work done in [25] proposes a business model for service provisioning in MANET with SIP servlets. This model consists of four roles: *end-user* (E.U.), *service provider* (S.P.), *service capabilities provider* (S.C.P.), and *service execution environment provider* (S.E.P). Figure 4.1 below depicts the business model.

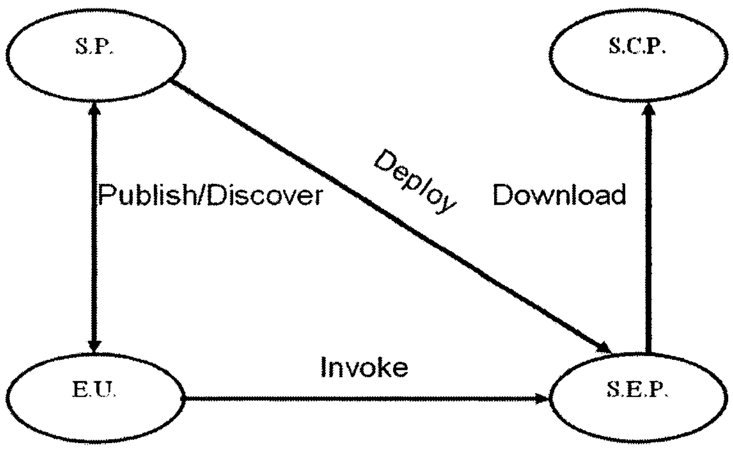


**Figure 4.1: Business model for service provisioning with SIP servlets in MANET**

An **end-user** is the entity that requests a service to consume. A **service provider** is the entity that advertises and offers services to end-users, and owns service logic. The logic on how a SIP servlet service is built from its building blocks and conditions that trigger service execution are listed in the *deployment descriptor* for the service. Even though a service provider owns service logic, it does not, necessarily, own the resources required to build and execute the service. Therefore, a service provider ensures that both service building blocks and an environment to host the service are available. Services are constructed from building blocks called *capabilities*. SIP servlets act as capabilities for SIP servlet services. A **service capabilities provider** is an entity that acts as a repository where capabilities are stored. This entity is also referred to as a "servlet repository". Finally, a **service execution environment provider** provides an environment to deploy and execute services. In other words it is the provider of the SIP SE that loads and executes the SIP servlet applications. Service provider issues a "deploy" command to the SIP SE instructing it to download a service from the servlet repository and deploy the service. After the service is operational, SIP SE becomes ready to receive "invoke" commands from the end-user that wishes to consume the service. It should be noted that in service provisioning using a SIP SE only the "invoke" interaction has to occur over SIP protocol.

## 4.5 The Proposed Distribution Scheme

### 4.5.1 Overall View

In this section we propose an architecture to distribute a SIP SE across multiple nodes: *connector*, *controller*, *wrapper*, and *session repository*. This is equivalent to substituting the role of a service execution environment provider with four roles of: *connector* *provider*, *controller provider*, *wrapper provider*, and *session repository provider*. Figure 4.2 shows the four new roles within the boundary of the distributed SIP SE and how these roles interact with the three roles that are not part of the distributed SIP SE.

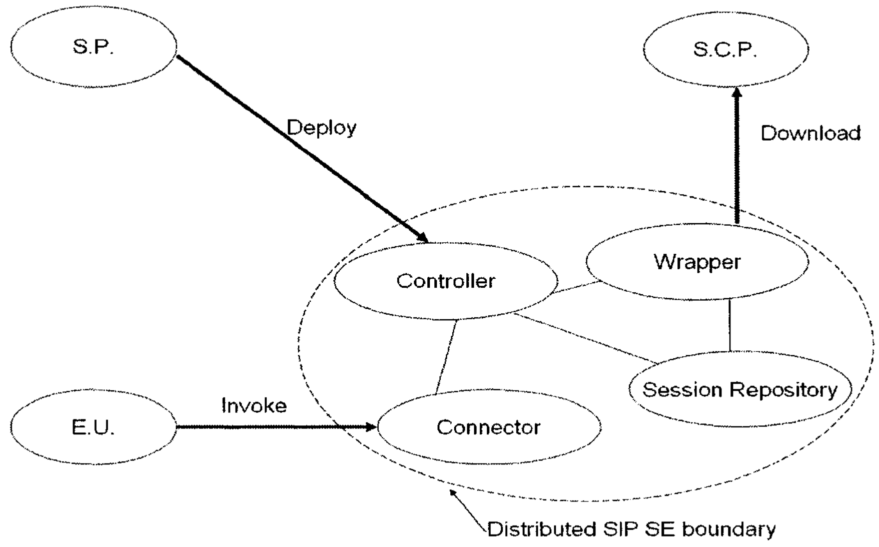


**Figure 4.2: An overall view of service provisioning with a distributed SIP SE**

### 4.5.2 Components of the distributed SIP SE

#### 4.5.2.1 Connector

A connector provides transport layer connectivity to a SIP SE by managing listen points. A listen point is a transport connection at which SIP messages are sent and received. This means that all SIP messages that a SIP SE sends or receives must traverse the connector. A transport connection of the connector may use TCP, UDP or any other transport protocol. The connector parses received SIP messages and validates their format. For every correctly-formatted SIP message that a connector receives, it creates an object representation of the message. Then, the connector forwards the object representation of the SIP message to the controller via intra-node communication as shown in Figure 4.3.

Inversely, a connector receives object representations of SIP messages from the controller, parses it into a SIP message and sends the message to its destination.

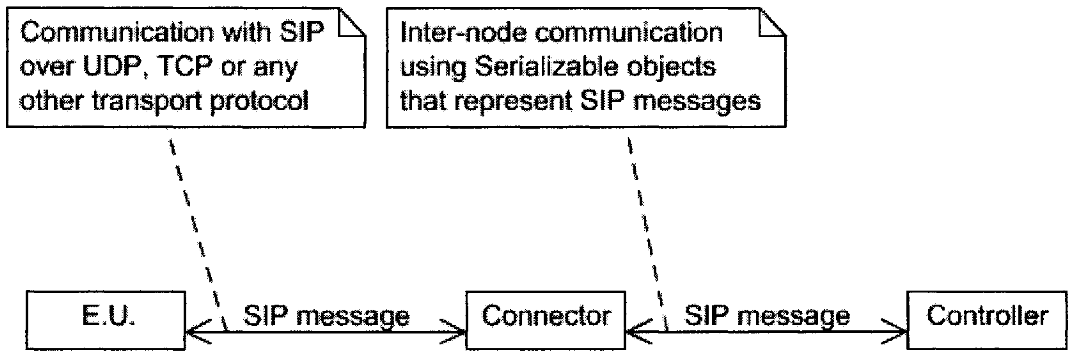

Communication with SIP over UDP, TCP or any other transport protocol
Inter-node communication using Serializable objects that represent SIP messages
E.U.
SIP message
Connector
SIP message
Controller

**Figure 4.3: Connector's interfaces**

It should be noted that the Connector performs two distinct functionalities: managing transport connectivity, and parsing SIP messages to and from a format used for intra-node communication between SIP SE nodes. A question arises on whether these functionalities should be performed by the same node or be split across separate nodes. On the one hand, splitting the functionalities between a transport node and a parsing node would enable multiple transport nodes to reuse the same parsing node. On the other hand, the output of the transport node would be an array of bytes representing a SIP message. This is, essentially, equivalent to what the transport node receives from the network. The array of bytes will have to be sent to the parsing node over inter-node communication that, in turn, is implemented over some transport mechanism. In our view, this introduces an overhead that is not justified. Therefore, we opted for combining the functionalities of transport connectivity and message parsing in one node.

The benefits of introducing a Connector stem from the fact that no other SIP SE node has to parse SIP messages, ensure their validity, or manage transport connections. The

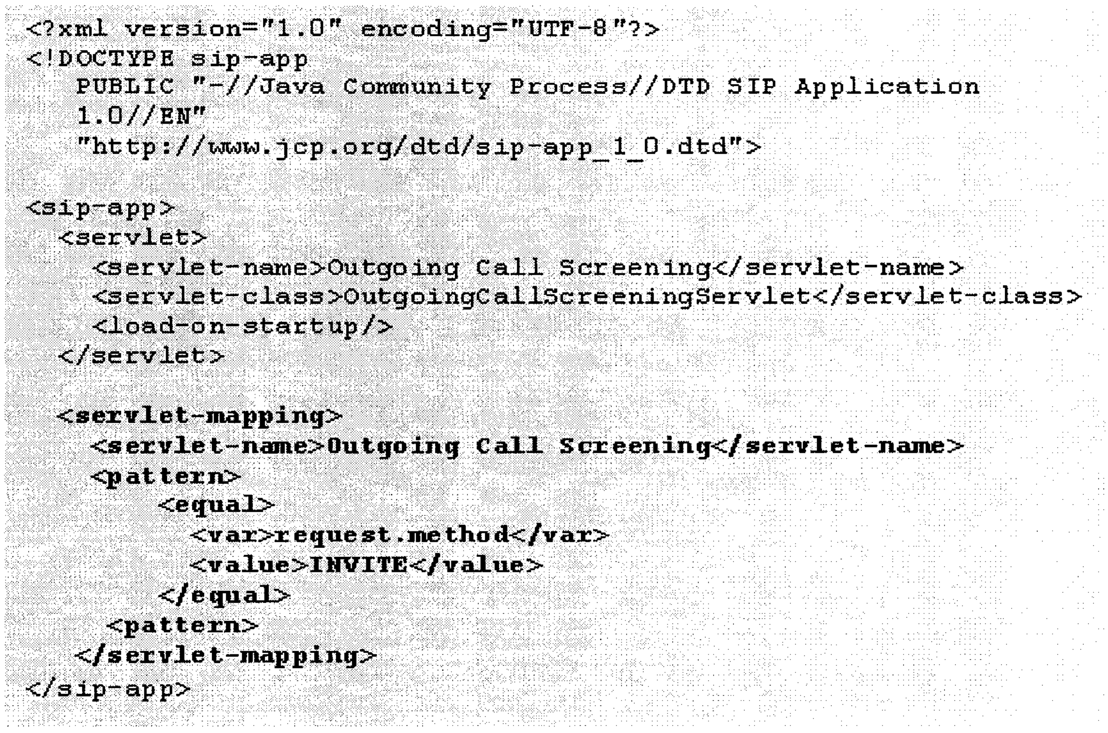
Connector prevents malformed SIP messages from propagating to the other SIP SE nodes. Instead, the other SIP SE nodes operate on an object representation of a SIP message that enables them to easily access and modify message content (its headers and body). It should be noted that the Connector functionality does not overlap with any other node's functionality.

#### 4.5.2.2 Controller

A controller is a central node that coordinates all other nodes of a distributed SIP SE. Furthermore the Controller is responsible for: receiving commands to deploy an application, handling SIP transaction layer functionality, and SIP message routing (i.e., deciding which servlet gets to serve an incoming SIP message).

First, the Controller is the entity which a service provider instructs to deploy a SIP application. The Controller receives a deployment descriptor and creates rules mappings that specify the conditions that trigger SIP servlets execution. The Controller does not load and instantiate the servlets itself. Instead, it forwards the deploy command to the Wrapper for further processing. Figure 4.4 shows (in bold) the portion of a deployment descriptor that a Controller uses to create a rule that triggers a servlet execution. The Controller will use this information to decide where to route incoming initial requests (i.e., requests that are not part of an established SIP dialog).

Second, the Controller manages SIP server and client transactions. A server transaction is created for each incoming SIP request. It is used to correlate a received SIP request with its responses. If no SIP responses are generated within a given amount of time a server transaction mechanism returns a SIP response that indicates a timeout [5]. A client transaction is created for each outgoing SIP request. It is used to correlate received responses to sent requests. Additionally, client transactions control request retransmissions and handle timeouts.



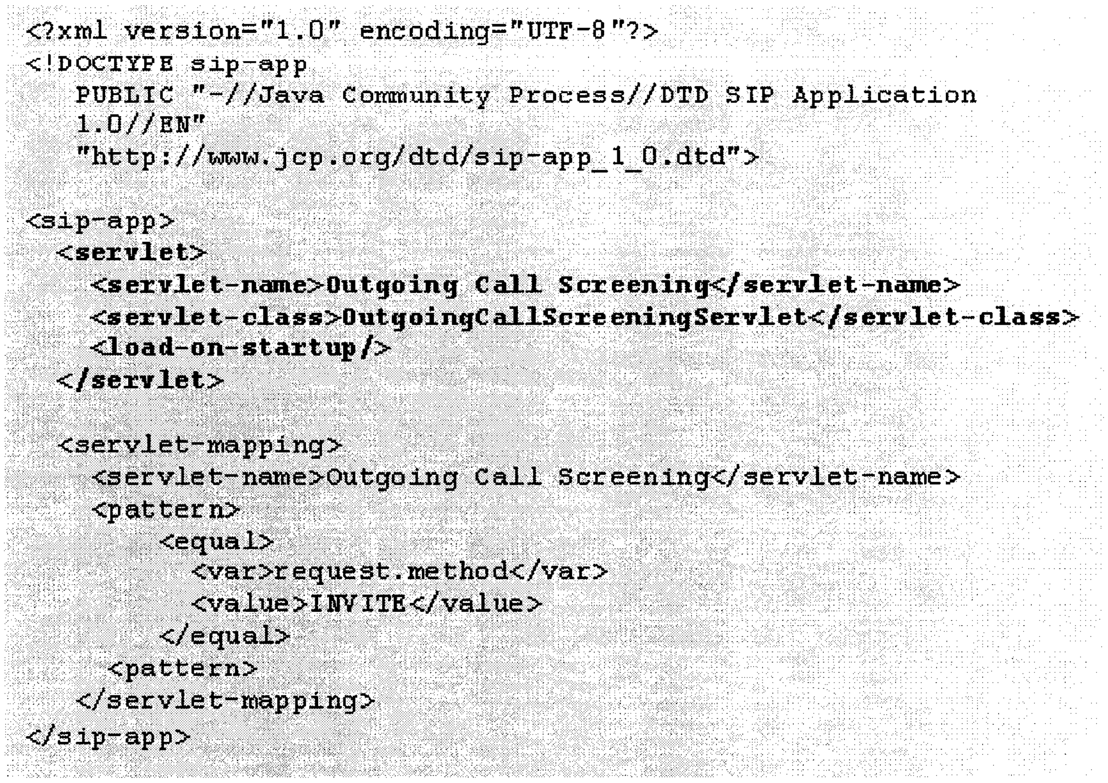
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app
   PUBLIC "-//Java Community Process//DTD SIP Application
   1.0//EN"
   "http://www.jcp.org/dtd/sip-app_1_0.dtd">

<sip-app>
  <servlet>
    <servlet-name>Outgoing Call Screening</servlet-name>
    <servlet-class>OutgoingCallScreeningServlet</servlet-class>
    <load-on-startup/>
  </servlet>

  <servlet-mapping>
    <servlet-name>Outgoing Call Screening</servlet-name>
    <pattern>
        <equal>
          <var>request.method</var>
          <value>INVITE</value>
        </equal>
     <pattern>
   </servlet-mapping>
</sip-app>
```

**Figure 4.4: Deployment descriptor portion processed by the controller**

Third, when the controller receives a SIP message from the connector it needs to decide on the servlet that should serve the message. Initial requests are routed according to the rules specified in the deployment descriptor. For example, the deployment descriptor in Figure 4.4 specifies that a received initial SIP request of type INVITE should be handled by the "Outgoing Call Screening" servlet. For all SIP messages that are not initial SIP requests the controller computes a SIP session id and retrieves a SIP session from the session repository. SIP sessions contain information on what application they belong to and what servlet should serve the incoming message. After the controller determines the servlet that should serve the incoming message it inserts information containing the servlet's name to the message and forwards the message to the wrapper on which the servlet is hosted. The information on the servlet's name is necessary because a Wrapper may run a servlet application that consists of multiple servlets. If the name of the servlet is not provided, then the Wrapper will have to determine which servlet gets to serve the message. This, in turn, will result in an overlap of functionality between the Controller and the Wrapper.

#### 4.5.2.3 Wrapper

A wrapper is the entity that provides an execution environment for SIP servlets. It downloads a servlet from a repository. Then, it loads the servlet's class and calls the servlet's `init()` method. For each SIP message that should be served by the servlets the wrapper calls the servlet's `service()` method. Finally, when the servlet should be phased-out the wrapper calls the servlet's `destroy()` method. Figure 4.5 shows (in bold) the portion of a deployment descriptor processed by the wrapper. This portion creates a mapping between a servlet's name and its class. The name of the servlet that should serve an incoming SIP message is provided to the Wrapper by the Controller. The Wrapper, then, matches the servlet name to an instance of a servlet class, and calls `service()`method on that instance. In the example shown in Figure 4.5, when a SIP request should be served by "Outgoing Call Screening" servlet the wrapper will call `service()` method of "OutgoingCallScreeningServlet" class.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app
   PUBLIC "-//Java Community Process//DTD SIP Application
   1.0//EN"
   "http://www.jcp.org/dtd/sip-app_1_0.dtd">

<sip-app>
  <servlet>
    <servlet-name>Outgoing Call Screening</servlet-name>
    <servlet-class>OutgoingCallScreeningServlet</servlet-class>
    <load-on-startup/>
  </servlet>

  <servlet-mapping>
    <servlet-name>Outgoing Call Screening</servlet-name>
    <pattern>
        <equal>
          <var>request.method</var>
          <value>INVITE</value>
        </equal>
     <pattern>
   </servlet-mapping>
</sip-app>
```

**Figure 4.5: Deployment descriptor portion processed by the wrapper**

#### 4.5.2.4 Session Repository

Session Repository stores state information for SIP servlet applications. The motivation for introducing this node is to separate servlet hosting from application state storage. This achieves several benefits. First, creating a node that is solely responsible for storing state information reduces the sizes of the other nodes. Then, it introduces redundancy which makes it easier for a distributed SIP SE to operate in MANET environments. This is due to the fact that servlet hosting nodes become stateless. Therefore, different messages directed to a particular servlet may be served by different instances of that servlet; each instance running on a different host. Finally, from a conceptual point of view this separation introduces a node with high cohesion because its responsibilities have a well-defined and narrow scope; storing state information.

Application state can be classified into two types: protocol session local, and application wide. For SIP, protocol session local state is stored per each SIP point-to-point relationship between two User-Agents (i.e., per each SIP session). It is possible to store local state information for other protocols as well (e.g., HTTP sessions); but this is beyond the scope of this thesis. Application wide state encloses all protocol sessions created by the application in addition to application data in the form of name/value pairs. Session Repository provides storage and retrieval for both protocol session local (SIP session), and application wide (SIP application session) state information. It also ensures the integrity of the data by resolving issues that arise when multiple entities require access to the same piece of state information.

## 4.6 Communication between SIP SE nodes and external actors

This section describes interactions between the nodes of a distributed SIP SE on one side and the nodes representing the other business roles (service provider, end-user, and service capabilities provider) on the other. As previously depicted in Figure 4.1, the three types of interactions between SIP SE nodes and external actors are: `Deploy`, `Download`, and `Invoke`. Figure 4.2 expands the SIP SE to reveal the nodes that it consists of. It also reveals which nodes of the distributed SIP SE are involved in executing the aforementioned interactions. Next, we discuss those three interactions in detail.

### 4.6.1 Deploy

Deploy is the interaction that an S.P. issues in order to instruct a distributed SIP SE to put a SIP application into service. As shown in Figure 4.2 this interaction occurs between the S.P. and the Controller. Three conditions must be satisfied prior to issuing this command.

First, a distributed SIP SE formation must be in place. Then, the S.P. must be aware of the location of the S.C.P. where the application is stored. Finally, the S.P. must discover the Controller node of the SIP SE. After the deploy interaction completes successfully the distributed SIP SE becomes ready to receive invocations of the service from the E.U. Table 4.1 summarizes Deploy interaction.

| | |
|---|---|
| Name | Deploy |
| Purpose | Instruct a SIP SE to put a SIP application into service. |
| Parameters | 1. Service logic (deployment descriptor).<br>2. Location of the Capability Provider |
| Response | Connector's SIP listen-point location or an error if deployment fails. |

**Table 4.1: Deploy interaction**

### 4.6.2 Download

Download is the interaction that the Wrapper issues in order to download an archive containing a SIP servlet application from an S.C.P. The application archive contains servlet classes and other resources necessary to run a service (such as image or audio files). After this interaction completes successfully the Wrapper will extract the servlets from the archive, and load and initialize the servlet classes. Table 4.2 summarizes Download interaction.

| | |
|---|---|
| Name | Download |
| Purpose | Obtain service capabilities and other resources required to run a service. |
| Parameters | The URI where the capabilities are located. |
| Response | Servlet application archive is returned to the Wrapper. |

**Table 4.2: Download interaction**

### 4.6.3 Invoke

After a successful completion of Deploy and Download interactions, Invoke interaction allows the E.U. to trigger execution of a service deployed on the SIP SE. This interaction is conducted by sending a SIP request to the Connector node of the SIP SE. The SIP SE will process the incoming request and execute service logic responsible for processing that request. Table 4.3 summarizes Invoke interaction.

| | |
|---|---|
| Name | Invoke |
| Purpose | Invoke service logic on an incoming request from the End-User. |
| Parameters | - |
| Response | Depends on the service logic. |

**Table 4.3: Invoke interaction**

## 4.7 Communication within the distributed SIP SE

This section describes interactions between the different nodes of a distributed SIP SE. Figure 4.6 below emphasizes the links between SIP SE nodes.

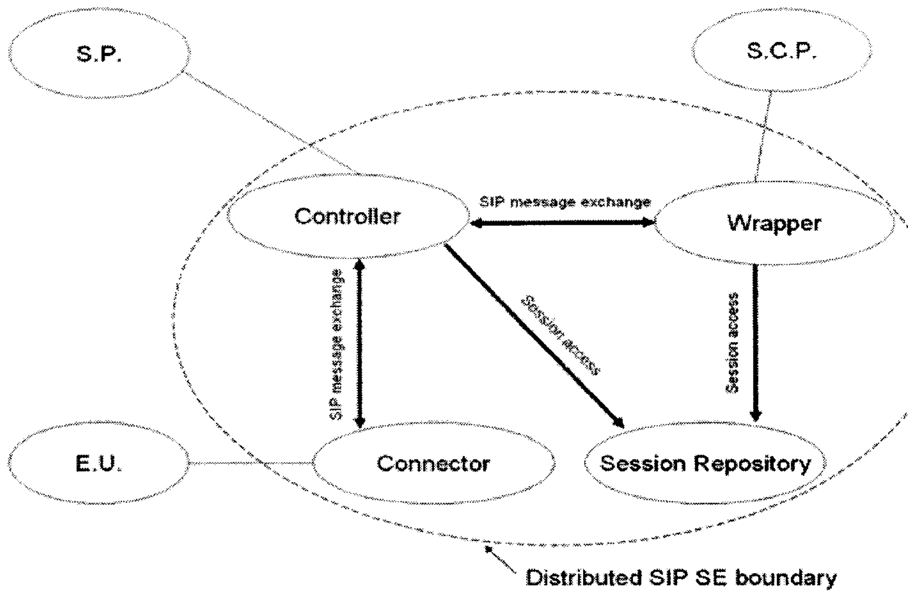


**Figure 4.6: Communication within the distributed SIP SE**

Two types of interactions occur between SIP SE nodes: *SIP message exchange* interactions, and *session access* interactions. The rest of this section discusses the two interaction types, and defines abstractions used for conducting those interactions.

### 4.7.1 SIP message exchange

This interaction is used to pass a representation of a SIP message between SIP SE nodes. Two abstractions are introduced to achieve this:

1. SIP_REQUEST: represents SIP request parts: start line, headers, body.
2. SIP_RESPONSE: represents SIP response parts: start line, headers, body.

Those abstractions carry information found in SIP requests and SIP responses respectively. Consider the case of a SIP request received by the Connector. The Connector parses the received request creating a SIP_REQUEST. Then, it passes the SIP_REQUEST to the Controller. The Controller, in turn, passes the SIP_REQUEST to the Wrapper. The Wrapper applies service logic and creates a SIP_RESPONSE. The

SIP_RESPONSE traverses the same path as the SIP_REQUEST, but in the opposite direction – from Wrapper, to the Controller, and then, to the Connector. The Connector parses the SIP_RESPONSE into a byte array and sends it to its destination.

### 4.7.2 Session access

Session access interactions are used to: create, modify, delete, and access both SIP sessions and application sessions. First, we define the structures that represent a SIP session and a SIP Application Session. Then, we define the different types of session access interactions.

The first data structure primitive used for session access interactions is `SIP_APP_SSN`. This primitive is used to represent a SIP Application Session. It consists of a unique session id field, a list of all protocol session ids that are part of this application session, and a map of application session's attributes (see Table 4.4).

| **Parameter** | **Value** |
|---|---|
| App_ssn_id | SIP application session id |
| Protocol_ssn_ids | A list of protocol session ids that are part of the application session. |
| Attribs | A map of application session's attributes. The map keys are attribute names. The map values are the attribute values. |

**Table 4.4: Abstraction for a SIP application session (`SIP_APP_SSN`)**

The second data structure primitive used for session access interactions is `SIP_SSN` (see Table 4.5). It represents one point-to-point relationship between a servlet and a User Agent. This primitive consists of three fields. First, a session id field that is unique. The second field is the SIP application session the SIP session belongs to. This field is of type of SIP_APP_SSN. The third field is a map of the SIP session's attributes.

| **Parameter** | **Value** |
|---|---|
| Sip_ssn_id | SIP session id |
| Sip_App_Ssn | SIP application session the SIP session belongs to. |
| Attribs | A map of application session's attributes. The map keys are attribute names. The map values are the attribute values. |

**Table 4.5 - Abstraction for a SIP session (SIP_SSN)**

After discussing the primitives used in session access interactions we, now, discuss the different types of session access interactions. A new SIP session is created with a SIP_SSN_CREATE interaction. A SIP session is created before the actual SIP dialog that it represents is established. Therefore, a SIP session is, first, created with a temporary session id. When the SIP dialog is established this temporary session id should be replaced with the dialog id. The interaction responsible for changing the session id is SIP_SSN_CHANGE_ID. The other session access interactions are concerned with SIP session retrieval, deletion, and returning a modified SIP session to the repository. These primitives are: SIP_SSN_GET, SIP_SSN_DELETE, and SIP_SSN_PUT, respectively. These primitives are shown in Table 4.6.

| | | |
|---|---|---|
| • SIP_SSN_CREATE | | |
| | Purpose | Creates a SIP session and adds it to the repository. |
| | Parameters | • Sip_Session |
| • SIP_SSN_CHANGE_ID | | |
| | Purpose | Instructs the repository to modify SIP session id. |
| | Parameters | • Old SIP session id.<br>• New SIP session id. |
| • SIP_SSN_GET | | |
| | Purpose | Gets SIP session from the repository. |
| | Parameters | • SIP session id. |
| • SIP_SSN_PUT | | |
| | Purpose | Returns a SIP session to the repository. |
| | Parameters | • Sip_Session. |
| • SIP_SSN_DELETE | | |
| | Purpose | Permanently removes a SIP session from the repository. |
| | Parameters | • SIP session id. |

**Table 4.6: Session access primitives**

## 4.8 Distribution scheme vs. requirements

First, we proposed distributing SIP SE functionality across multiple nodes with each node performing part of the SIP SE functionality. Therefore, each participant node is expected to require fewer resources than a node that hosts a SIP SE in its entirety. The exact amount of resources required for each node will be determined by the implementation of the prototype which is discussed in the next chapter.

Second, the proposed distribution scheme supports on-the-fly deployment through the "Deploy", and "Download" interactions.

Third, the proposed distribution scheme does not assume a presence of an entity that the distributed SIP SE nodes can query in order to perform discovery. Therefore, the SIP SE nodes must bootstrap to discover each other; thus meeting the requirement for bootstrap discovery.

Fourth, the requirement for tolerating changes in network topology is met by allowing multiple SIP SE components of the same type to be part of a distributed SIP SE formation. For example, multiple Wrapper nodes running the same SIP service may be present in order to ensure service continuity should one of the Wrapper nodes become unavailable.

The requirement for handling changes in connectivity and user mobility can be met by implementing node communication using middleware targeted at MANETs. Such a middleware will be responsible for user mobility and changes in link characteristics.

Finally, the proposed distribution scheme satisfies the scalability requirement. This is achieved by letting multiple nodes of the same type to coexist in a distributed SIP SE formation (in a way similar to meeting the requirement on tolerating changes in topology).

## 4.9 Conclusions

In this chapter we derived requirements for a scheme to distribute a SIP SE for MANET. We showed that existing SIP SE implementations fail to satisfy most of these requirements. Therefore, we introduced an architectural scheme for distributing a SIP SE for MANET and showed how it met the requirements. The distribution scheme includes four component roles for a SIP SE: controller, connector, session repository and wrapper. Furthermore, the distribution scheme describes models of communication used among the SIP SE nodes and between the distributed SIP SE and external actors. Finally, we established that the distribution architecture meets the requirements that we derived with the exception of the requirement that the nodes of the SIP SE should be deployable on mobile, handheld devices. This requirement will be validated in the next chapter with a distributed SIP SE prototype.

# Chapter 5: Prototype and Proof of Concept

This chapter describes a prototype of the distributed SIP SE and two services used as a proof-of-concept for service deployment on the prototype. First, the communication model with the middleware of choice is explained. Then, the main control flows that occur in the prototype are discussed. Finally, analysis of results obtained with two conferencing services is presented.

## 5.1 Communication with LIME

We chose LIME [26] as the middleware for communication between the distributed nodes of the prototype. LIME was also chosen as means of communication between the distributed SIP SE prototype and Service Provider. The rest of this section starts by a brief outline of the main concepts of LIME. Then, a justification for using LIME is provided by contrasting it with another middleware protocol, Java RMI. Finally, a description of how LIME is used by the prototype's nodes is provided.

### 5.1.1 Introduction to LIME

LIME uses a model where information communicated between entities is carried in tuples. A *tuple* is an ordered sequence of typed fields. A source sends a message to a destination by adding a tuple to the destination's *tuple space.* This operation is referred to as *out(t)* where *t* is the tuple being communicated. In order for the destination to retrieve a tuple written into its tuple space, the destination must register a *reaction* with a *template* for the tuples it expects to receive. A template is a special tuple that provides logic used to match received tuples against those expected. For example, consider a tuple space that expects to receive tuples that consist of three fields: a string with a predefined value "message", followed by an arbitrary integer value, followed by an arbitrary string value.

A template used in the reaction for such tuples would be <"message", int, String.class>. The first field, an *actual* with a value "message". Actual fields specify the exact value of a field. The second and the third fields are *formals*. Formal fields specify the type of a field; but not its value. In the aforementioned example the second and the third fields of the received tuples must be of types integer and string, respectively.

Each node of the distributed SIP SE prototype maintains its tuple space together with reactions to the tuples it expects to receive. In order for a node to send a message to another node the source node writes a tuple to the destination node's tuple space. The destination node reacts to the tuple, reads it from its tuple space and applies processing logic to the received information.

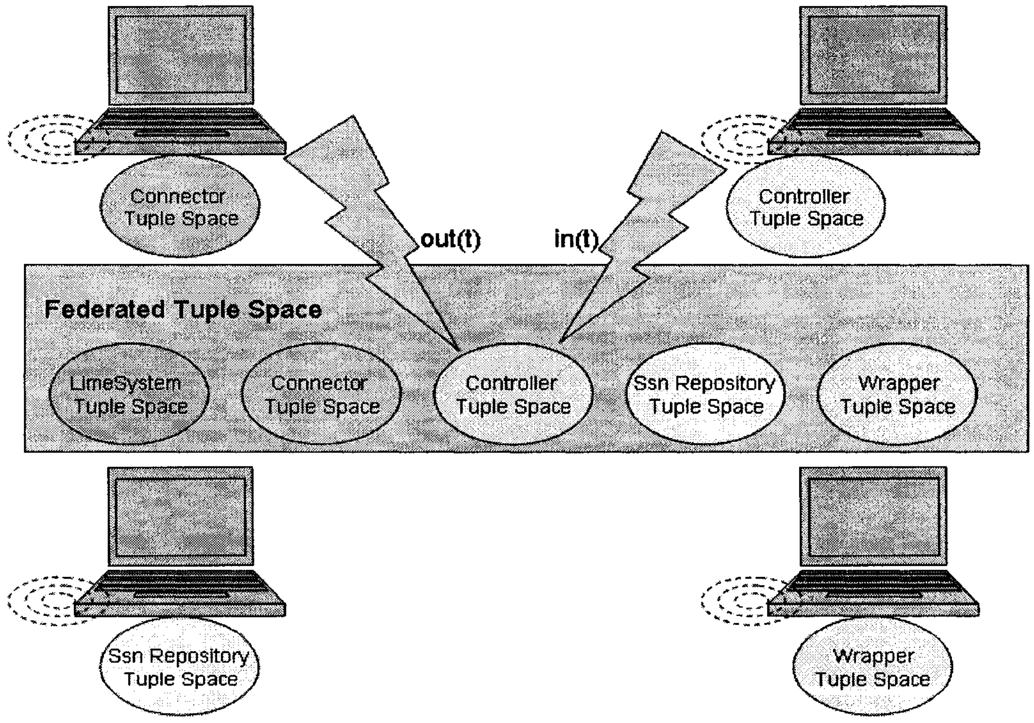


**Figure 5.1: SIP SE communication with LIME**

### 5.1.2 Mobility support

LIME was specifically designed for mobile environments where participants constantly adapt to dynamic changes in their context (e.g., node mobility and changes in link characteristics). LIME facilitates mobility by decoupling operations through which nodes communicate from how communication details are implemented. All nodes located within range of each other share a common *LimeSystem* tuple space. This tuple space contains context information, such as accessible mobile nodes and their tuple spaces. Nodes can detect context changes by reacting to LimeSystem events and adapt accordingly. The context itself is represented using a *Federated Tuple Space*. This tuple space provides each node with an illusion that all tuple spaces belonging to all connected nodes are located locally.

For example, when the Connector needs to send a message to the Controller it performs out(t) operation (where "t" is the tuple carrying the message) to the Controller tuple space. The Connector has access to the Controller tuple space through the Federated tuple space as shown in Figure 5.1.

On the other hand, existing middlewares such as Java RMI were designed for client/server environments. Therefore, they exhibit strong coupling between clients and servers, as well as predefined roles of: server, client, and registry. Furthermore, adapting dynamically to context changes is made difficult by relying on the notion of addresses for interaction between entities. In order to deliver a service over RMI a server must create remote objects and register them with an RMI registry. In order to access a remote object a client must query an RMI registry and obtain a *stub* to the remote object. The stub acts as a proxy to the remote object; a client calls a method on the stub and the call is relayed to the remote object via RMI.

### 5.1.3 LIME tuples of the prototype

This section describes tuples used for the distributed SIP SE prototype's communication model. First, tuples used for SIP message exchange are discussed. Then, tuples used for Session Repository access are presented. Finally, a tuple used for service deployment is presented.

#### 5.1.3.1 SIP message exchange

SIP messages transmitted in message exchange tuples are represented by instances of either *SimpleSipRequest* or *SimpleSipResponse* classes. Both of these classes extend a common, abstract parent, *SimpleSipMessage*. In order to be transmitted across the network the parent class implements *java.io.Serializable* interface. This class hierarchy is shown in Figure 5.2 below.

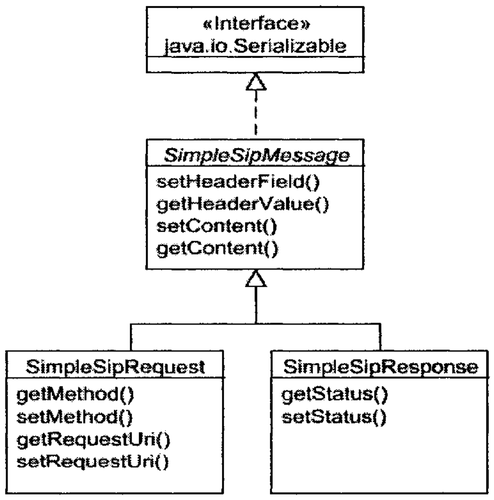


**Figure 5.2: SimpleSipMessage hierarchy**

Tuples that carry representations of SIP messages consist of three fields: a predefined tuple name, an instance of a *SimpleSipMessage* subclass, and an originator node's type (as a string). Therefore, templates used by the reactions on the destination node's side also consist of three fields. First, an actual field of type string to denote the tuple's name. Second, a formal whose type is a subclass of *SimpleSipMessage*. Third, a formal of type string to denote the originator node's type. Message exchange tuple templates' structure is shown in the following two tables: Table 5.1 and Table 5.2.

| **Field** | **Actual or Formal** | **Field type or value** |
|---|---|---|
| Tuple Name | Actual | "SimpleSipRequest" |
| Payload | Formal | SimpleSipRequest.class |
| Originator Node | Formal | String.class |

**Table 5.1: A tuple template to transmit a SIP request**

| **Field** | **Actual or Formal** | **Field type or value** |
|---|---|---|
| Tuple Name | Actual | "SimpleSipResponse" |
| Payload | Formal | SimpleSipResponse.class |
| Originator Node | Formal | String.class |

**Table 5.2: A tuple template to transmit a SIP response**

### 5.1.3.2 SIP session access

Communication with Session Repository occurs using the tuple depicted in Table 5.3.

| Field | Actual or Formal | Field type or value |
|---|---|---|
| Tuple Name | Actual | " LimeSessionRepository" |
| Command Type | Formal | SsnRepCommands.class |
| Session Id | Formal | String.class |
| New Dialog Id | Formal | String.class |
| Sip Session | Formal | SipSession.class |
| Originator Tuple Space | Formal | String.class |

**Table 5.3: A tuple template to access Session Repository**

A SIP session access tuple carries a session access command and its arguments. Session access command types are provided in the `SsnRepCommands.class` enumeration. Commands supported by the prototype are:

- ADD: adds a new session to the repository.
- CHANGE: modifies session id.
- GET: retrieves a session from the repository.
- PUT: returns a session to the repository.

New dialog id field is set to an empty string for all commands except the CHANGE where it is set to the session's new id, the dialog id. Sip session field is set to an empty session for all commands except the ADD and the PUT. The only command for which the Session Repository provides a response is the GET command. The response tuple has the same structure as the request tuple. The value of Sip Session field is set to the actual sip session retrieved from the repository. The response if written to the tuple space of the originator specified in ”Originator Tuple Space” field.

### 5.1.3.3 SIP application deployment

This section defines the deployment command tuple. This tuple is issued by the Service Provider which writes it to Controller’s tuple space. The Controller processes the command and then forwards it to the Wrapper. The deploy tuple template structure is shown in Table 5.4 below:

| **Field** | **Actual or Formal** | **Field type or value** |
|---|---|---|
| Tuple Name | Actual | ”DEPLOY_COMMAND” |
| Service URL | Formal | String.class |
| Deployment Descriptor | Formal | byte[] |
| Service Name | Formal | String.class |

**Table 5.4: A tuple to transmit a deploy command**

A deployment tuple provides a URL from which to download an archive containing a SIP service. It also provides the deployment descriptor for the service (sip.xml) and the service name.

## 5.2 Control flows

This section describes the main control flows of the prototype. First, the flow of the deploy command is explained. Then, the control flow of serving initial and subsequent requests is provided

## 5.2.1 Deploy command flow

Deploy action is performed by the Service Provider to instruct the Controller to load a SIP servlet application. The command flow for this action is shown in Figure 5.3 below.

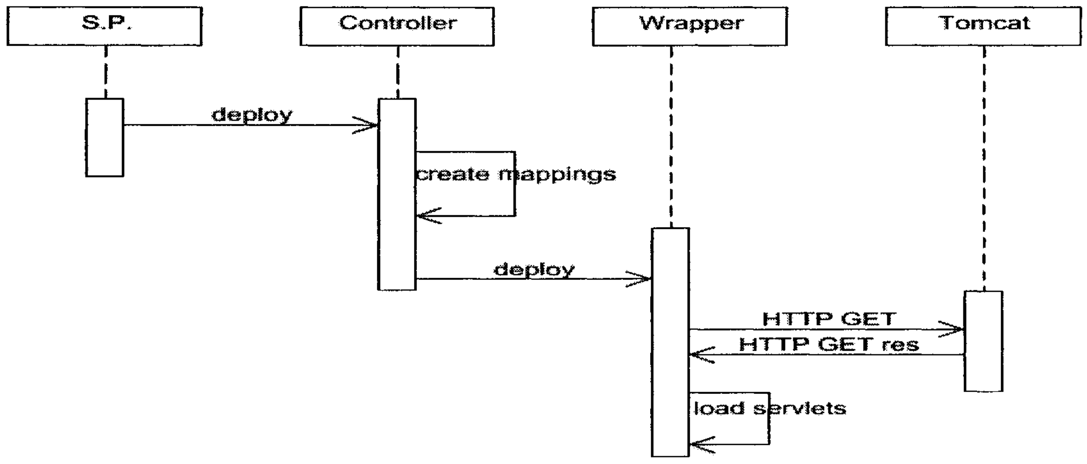


**Figure 5.3: Service deployment control flow**

First, the Service Provider instructs the Controller to deploy a SIP servlet application by writing a deploy tuple to Controller's tuple space. As specified in section 5.1.3.3 the deploy tuple contains the application deployment descriptor and the URL from which to the download the application archive. The Controller parses the deployment descriptor and creates mapping rules that specify conditions that trigger the application to be invoked. Then, the Controller writes the deploy tuple to Wrapper's tuple space. The Wrapper obtains the URI of the application from the tuple. Then, the Wrapper downloads the application's archive by issuing an HTTP Get command. The application's archive is passed to the Wrapper in the HTTP response. The Wrapper extracts the archive's contents onto a local directory, loads and initializes the servlet classes contained in the archive. Finally, the Wrapper parses the deployment descriptor to create mappings between servlet names and classes.

## 5.2.2 SIP message flow

Figure 5.4 illustrates the flow that occurs when an initial request is received.

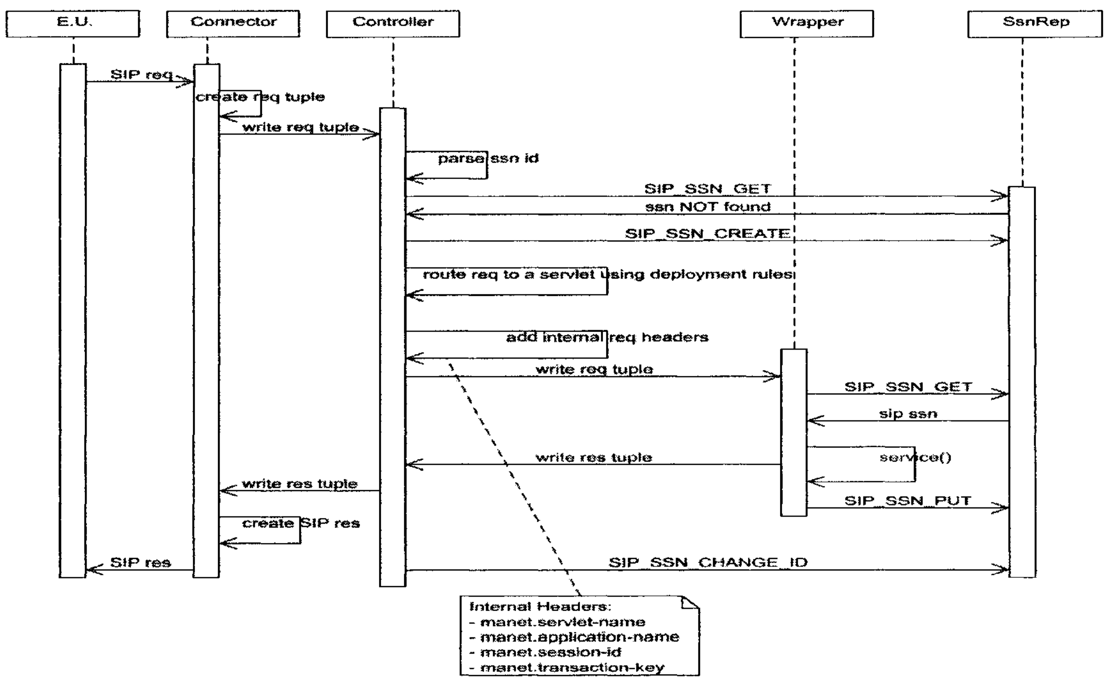


**Figure 5.4: Initial request serving control flow**

The Connector receives a SIP request, creates a tuple representation of the request and writes the tuple to Controllers tuple space. The Controller obtains the request session id and contacts Session Repository to retrieve the session. Since, the request is an initial request, no corresponding sessions are found. The Controller routes the request to a servlet by applying rules stated in the deployment descriptor. A rules matching result specifies the name of the application and the name of the servlet that should serve the request. Then, the Controller proceeds to create a new SIP session, and adds it to the repository. Since, the session is in its *initial* state a temporary session id is used. This temporary id is replaced when a response is generated and the SIP session moves to its *confirmed* state. Before writing the request tuple to Wrapper's tuple space the Controller adds a number of SIP headers to the requests. These headers take part in the communication between the Controller and the Wrapper. They are explained in detail later in this section.

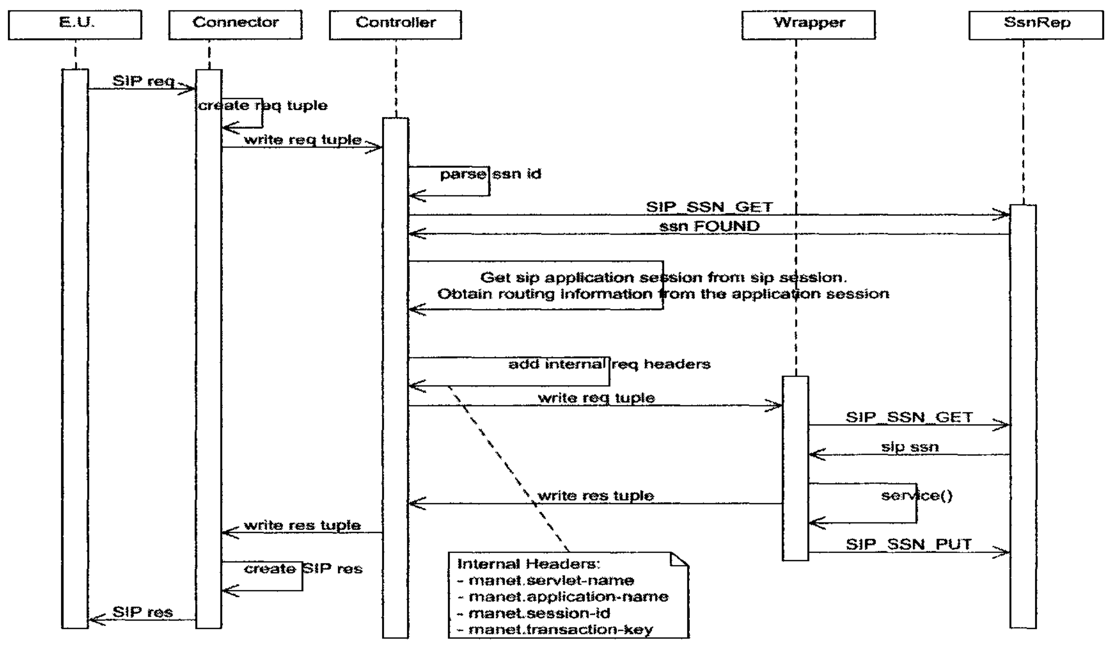


**Figure 5.5: Subsequent request serving control flow (also applies to serving responses)**

Control flow for subsequent requests is shown in Figure 5.5. It differs from the flow of initial requests in the way the request is routed. The Controller obtains the request session id and contacts Session Repository to retrieve the session. Since, the request is a subsequent request a corresponding session is retrieved from the Session Repository. Information regarding the name of the application and the servlet that the request should be routed to is found in the application session enclosed inside the sip session. This information is retrieved and placed into the specialized SIP headers of the request. Then, the request tuple is written to the Wrapper tuple space.

Specialized SIP headers assist in providing communication between SIP SE nodes. In order to convey a piece of information, a SIP SE node may insert a specialized header into a SIP message before passing it to another SIP SE node. The specialized headers we propose to introduce are not currently defined in SIP. Therefore, the SIP SE must remove those headers before sending a SIP message to a node that is not part of a SIP SE. Table 5.5 below enumerates the proposed specialized headers and their meaning.

| **Header Name** | **Header Value** | **Example value** |
|---|---|---|
| `manet.servlet-name` | Servlet to invoke (as specified in the deployment descriptor). | `conferencingServlet` |
| `Manet.application-name` | Servlet Application name. | `Conferencing` |
| `Manet.session-id` | SIP session id. | `3uydip` |
| `Manet.transaction-key` | SIP transaction key and type. | `z9hG4bKegq478;server` |

**Table 5.5: Specialized SIP headers**

All the proposed specialized headers are used by the Controller to convey information to the Wrapper. When a SIP message arrives at the Controller, the Controller determines the application, the servlet within that application and the id of the SIP session the message belongs to. Therefore, the Controller forwards this information to the Wrapper so that the latter does not have to redo the effort. Additionally, the Controller passes the key of the SIP transaction the received request belongs to. This information is not required by the Wrapper. Instead, it is forwarded back to the Controller when the Wrapper generates a response. This is due to the fact that the Controller needs to match the response to an existing SIP transaction in order to handle the response properly.

## 5.3 Prototype characteristics

The prototype nodes had the footprints shown in Table 5.6 below:

| **Component** | **Footprint with no servlets deployed (Kbyte)** |
|---|---|
| Controller | 462 |
| Wrapper | 462 |
| Connector | 417 |
| Session Repository | 340 |

**Table 5.6: SIP SE node footprints**

The nodes of the distributed SIP SE and other actors involved in running the scenarios were deployed on three laptops. The centralized SIP SE that we used to compare the distributed prototype against was deployed on a desktop with characteristics shown in Table 5.7. The characteristics of the laptops were identical (shown in Table 5.8).

| | |
|---|---|
| **Processor** | Pentium 4, 3 GHz |
| **RAM** | 1 GB |
| **Operating System** | Windows XP Professional |
| **Connectivity** | Ethernet |

**Table 5.7: Desktop characteristics for the scenarios for the centralized SIP SE**

| | |
|---|---|
| **Processor** | Mobile Pentium 4 |
| **RAM** | 512 MB |
| **Operating System** | Windows XP |
| **Connectivity** | WLAN 802.11g |

**Table 5.8: Laptop characteristics for the scenarios for the distributed SIP SE**

The node placement on the laptops was done is such a way that no SIP SE actor is collocated with another SIP SE node it directly communicates with. The actor distribution is shown in Table 5.9 below.

| | | |
|---|---|---|
| **Laptop 1** | Controller | SIP SE node |
| | Servlet Repository (Apache Tomcat) | Non SIP SE node |
| **Laptop 2** | Session Repository | SIP SE node |
| | Connector | SIP SE node |
| | Service Provider | Non SIP SE node |
| **Laptop 3** | Wrapper | SIP SE node |
| | End-Users | Non SIP SE node |

**Table 5.9: Scenario actors' locations**

## 5.4 Interest-based Conferencing with N participants

### 5.4.1 Service description

In this scenario a service provider instructs the distributed SIP SE to initiate a new conference between two or more participants. As a precondition, it is required that no such conference exists prior to the execution of this scenario. The call-flow for this service is shown in Figure 5.6 below. In this scenario the Service Provider sends an INVITE with a special header "Manet.Conference" that lists the addresses of conference participants.

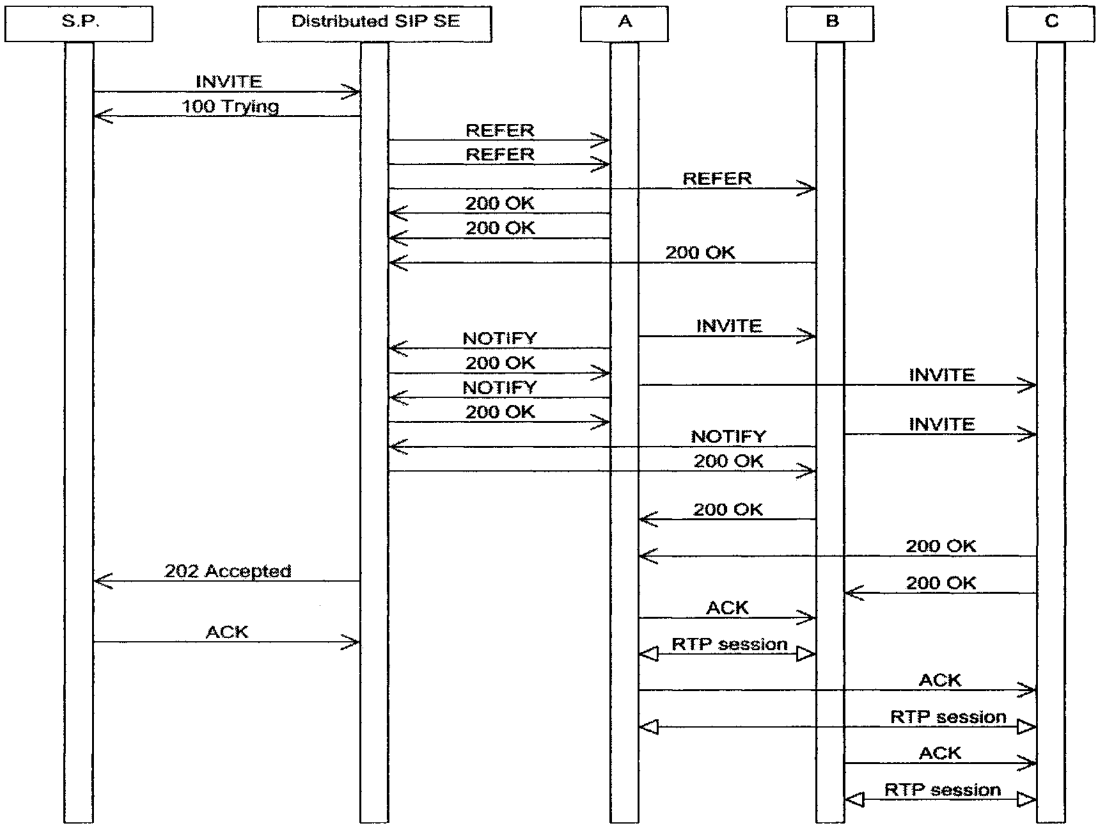


**Figure 5.6: Establishing a new conference between N nodes**

In this example, the value of the Manet-Conference header would be :

Sip:a@winter:7080;transport=tcp~sip:b@winter:8090~sip:c@winter:9010

(note that the "~" character is used as a separator). The servlet generates a necessary number of REFER requests in order to establish a fully-meshed conference. This number is equal to $\sum_{i=1}^{N} N-i = \frac{1}{2}N(N-1)$. This number is calculated by noting that for a fully-meshed conference to be established the first node sends an INVITE to the remaining (N-1) nodes. The second node sends invites to all nodes except the first node (i.e., (N-2)

INVITEs). The last node does not send any INVITEs. When a client receives a REFER it sends an INVITE to the referred node. Upon sending the INVITE the client immediately sends a NOTIFY back to the servlet to indicate that it is trying to contact the referred party. When the servlet receives back a number of NOTIFYs that is equal to the number of REFERs the servlet returns a 202 Accepted response back to the Service Provider.

### 5.4.2 Statistics and analysis

This scenario was executed five times on both centralized and distributed SIP SE. The results for the scenario execution are shown in Table 5.10 below. The results depict average response times of service execution and the standard deviation obtained for the response times. Service execution times constitute experimental data related to the scalability requirement. This data demonstrates whether the service can be delivered within reasonable amounts of time with a distributed SIP SE.

| *N* | **Distributed** | | **Centralized** | |
|---|---|---|---|---|
| | **Response time** | **Standard Deviation** | **Response time** | **Standard Deviation** |
| 2 | 4.69 | 1.97 | 1.00 | 0.34 |
| 3 | 17.16 | 6.07 | 7.45 | 0.74 |
| 4 | 24.05 | 6.97 | 13.4 | 3.24 |

**Table 5.10: Interest-based conferencing with N participants: execution results**

Figure 5.7 depicts the execution results for both the distributed and centralized SIP SEs side by side.

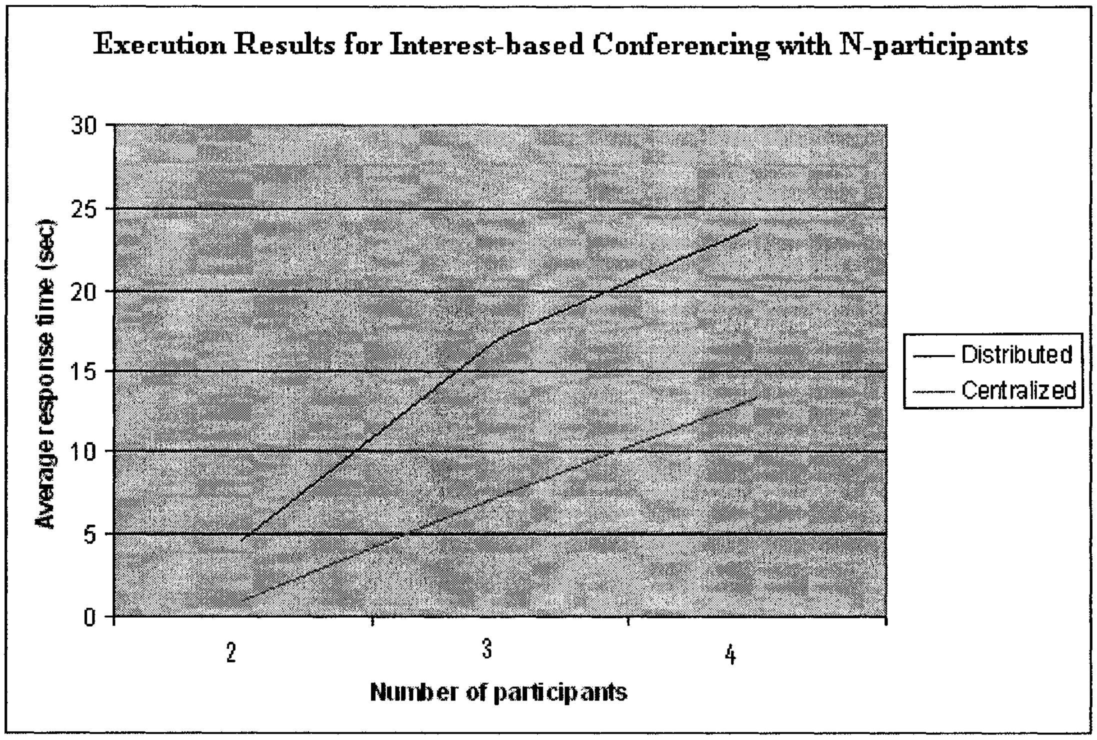


**Figure 5.7: Interest-based conferencing with N participants: response times comparisons**

It can be noted the average response time increased as the number of participants increased for both centralized and distributed cases. The centralized SIP SE was 4.69 times faster than the distributed SIP SE for 2 participants. This gap decreased as the number of participants grew to 3 (2.30 times difference) and 4 (1.84 times difference). With increasing the number of participants from two to four the increase in response time was much steeper in the case of the centralized SIP SE, 13.4 times as opposed to 5.1 times on the distributed SIP SE. This may be attributed to the higher runtime overhead due to Java's garbage collection on the centralized SIP SE. The larger overhead is due to a larger number of processing tasks performed on the centralized SIP SE compared to the processing on any node of the distributed SIP SE.

The standard deviation also increased for both cases. However, its increase was much more significant for the distributed SIP SE. This indicates high variance in response times between trials. This may be attributed to wireless link characteristics.

## 5.5 Adding a participant to an existing conference

### 5.5.1 Service description

In this scenario a service provider instructs the distributed SIP SE to add a new participant to an existing conference with *N* participants. The resulting conference will have *N+1* participants when the SIP SE finishes processing the Service Providers request. In this scenario the Service Provider sends an INVITE with a special header "Manet.Conference" that lists the addresses of *current* conference participants. Address of the new participant is provided in "Manet.New-Participant" header. For example, to invite client "C" to a conference that has "A" and "B" as participants the value of the two will be as shown below:

- Manet.Conference: <sip:a@winter:7080;transport=tcp>~sip:b@winter:8090
- Manet.New-Participant: sip:c@winter:9010headers: sip:c@winter:9010

The call-flow for this service is shown in Figure 5.8.

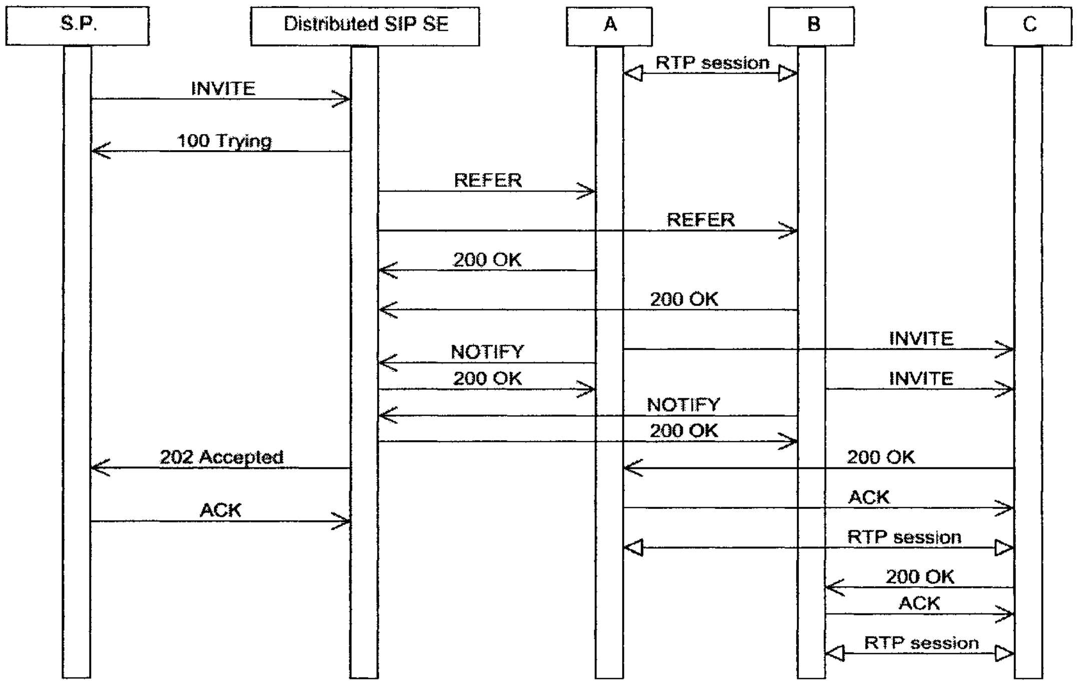


**Figure 5.8: Adding a new participant to an existing conference**

Note that "A" and "B" are already engaged in an RTP session. The servlet adds the new participant by sending *N* REFERs (one REFER per existing participant). Upon receiving *N* NOTIFYs the servlet returns a 202 Accepted response to the Service Provider.

The trials proceeded in four stages:

1. Creating a conference with two participants ("A" and "B").
2. Adding a third participant "C" to the conference.
3. Adding a fourth participant "D" to the conference.
4. Adding a fifth participant "E" to the conference.

### 5.5.2 Statistics and analysis

This scenario was executed ten times on both centralized and distributed SIP SE. The results for the scenario execution are shown in Table 5.11 below. Experimental data is presented in a way similar to the one in Section 5.4.2 and is also related to the scalability requirement.

| Phase | Distributed | | Centralized | |
|---|---|---|---|---|
| | Response time | Standard Deviation | Response time | Standard Deviation |
| 1. | 5.75 | 1.13 | 0.91 | 0.21 |
| 2. | 6.96 | 2.45 | 3.57 | 1.20 |
| 3. | 9.97 | 3.62 | 5.96 | 1.00 |
| 4. | 14.05 | 4.81 | 14.57 | 3.79 |

**Table 5.11: Adding a participant to an existing conference: execution results**

Figure 5.9 depicts the execution results for both the distributed and centralized SIP SEs.

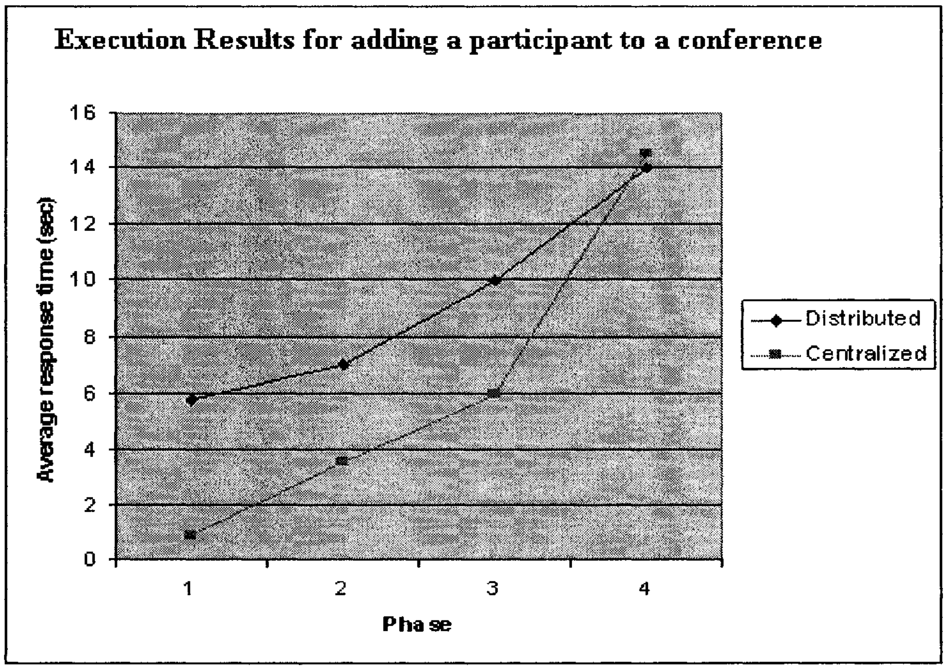


**Figure 5.9: Adding a participant to an existing conference: response times comparisons**

It can be noted that the gap in response times decreases with increasing number of participants. For the fourth category the distributed SIP SE slightly outperformed the centralized SIP SE. In distributed SIP SE runs the standard deviation increased consistently with the increase of number of participants. This is an indicator that response times obtained during the trials varied considerably from each other and from the average values. This may be attributed to varying wireless link conditions as well as to Windows operating system process scheduling.

## 5.6 Conclusions

In this chapter we introduced a SIP SE prototype as a proof of concept for the proposed architecture. We discussed LIME as our choice of middleware. LIME characteristics were presented and contrasted to those of Java RMI. Then, we examined the control flows for serving SIP messages. The control flows highlighted the interaction between the distributed SIP SE nodes as well as processing that occurs. Then, we presented the prototypes characteristics. The resulting footprints of prototype nodes did not exceed 500KB which makes them deployable on a large number of mobile, handheld devices available on the market today, thus satisfying our first requirement. The average execution times between the distributed SIP SE and a centralized one were compared. The biggest difference (6.3 times) was obtained with two participant nodes. This difference was less significant with a larger number of conference participants. In one scenario setup the distributed SIP SE response slightly surpassed that of the centralized SIP SE. Distributed SIP SE trials exhibited much higher standard deviation than those of the centralized SIP SE. This was attributed to the characteristics of the wireless links.

# Chapter 6: Conclusions and Future Work

## 6.1 Summary of Contributions

Mobile Ad Hoc Networks (MANET) will constitute a major part of Next Generation Networks. SIP protocol and SIP servlet technology will play a central role in signaling and service provisioning in MANET environments. In this thesis, applicability of SIP Servlet technology to service provisioning in MANET was investigated. This resulted in the introduction of a novel architecture for distributing a SIP SE for MANET. It was established that the novel architecture meets requirements imposed by MANET environments and by the limitations of participating devices. A prototype of the distributed SIP SE was implemented. The footprints of the prototype's nodes did not exceed 500 KB which makes them suitable for deployment on a large number of mobile, handheld devices. The prototype was evaluated with two conferencing services: establishing an interest-based conference between *N* participants, and adding a new participant to an existing multiparty service. The average execution times between the distributed SIP SE and a centralized one were compared. It was found that the difference in execution times became less significant as the number of conference participants increased. Due to the characteristics of wireless links, distributed SIP SE trials exhibited higher standard deviation than those of the centralized SIP SE.

## 6.2 Future Work

The SIP SE prototype discussed in Chapter 5 proved useful in evaluating the proposed distribution architecture. However, it can be enhanced with further features such as:

proxying support, lighter XML processor (to reduce its size and the amount of processing), event notification support and security.

The proposed architecture introduced a distribution scheme. However, it did not provide schemes for how the distributed nodes may advertise their capabilities and discover each other's presence [27].

The proposed architecture allows for a number of redundant nodes to be present in a distributed SIP SE. For example, multiple Wrapper nodes running the same SIP servlet application may be part of the same distributed SIP SE. Furthermore, multiple Session Repositories may provide storage and sharing of the applications' state information. Mechanisms for implementing such redundancies as well as component sharing between multiple distributed SIP SEs provide opportunities for future research.

## References

[1] Crow, B.P.; Widjaja, I.; Kim, L.G.; Sakai, P.T., "IEEE 802.11 Wireless Local Area Networks," Communications Magazine, IEEE , vol.35, no.9 pp.116-126, Sep 1997.

[2] R. Ramanathan and J. Redi, "A Brief Overview of Ad Hoc Networks: Challenges and Directions," IEEE Communications Magazine, vol. 40, pp. 20-22, May 2002.

[3] P. Mohapatra and S. Krishnamurthy, AD HOC NETWORKS: Technologies and Protocols, Springer, 2005.

[4] S. Basagni, M. Conti, S. Giordano and I. Stojmenovic, Mobile Ad Hoc Networking, IEEE Press and John Wiley & Sons, Inc., 2004.

[5] J. Rosenberg et al., "SIP: Session Initiation Protocol," Internet Eng. Task Force RFC 3261, June 2002; www.ietf.org/rfcrfc3261.txt.

[6] Fielding, R., Gettys, J., Mogul, J., Frysyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[7] A. Johnston, Ed., "Session Initiation Protocol Service Examples", draft-ietf-sipping-service-examples-11 (work in progress), October 2006.

[8] G. Camarillo, "SIP Demystified", McGraw-Hill, 2002.

[9] "Sip Servlet API specification, Version 1.0," Anders Kristensen. Release: February 4, 2003.

[10] James Gosling, et al., "The Java Language Specification", Third Edition, Prentice Hall PTR, 2005.

[11] "Java Servlet Specification, Version 2.4"; Release: November 24, 2003;

[12] Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C Recommendation 16 August 2006, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau eds.

[13] Defense Advanced Research Projects Agency (DARPA); http://www.darpa.mil

[14] Jubin, J.; Tornow, J.D., "The DARPA packet radio network protocols," Proceedings of the IEEE , vol.75, no.1pp. 21- 32, Jan. 1987.

[15] "BLUETOOTH - The universal radio interface for ad hoc, wireless connectivity," Ericsson Review, No. 3, 1998.

[16] M. Frodigh, P. Johansson and P. Larsson, "Wireless Ad Hoc Networking: The Art of Networking without a Network," Ericsson Review, No. 4, 2000.

[17] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. 2002. "A survey on sensor networks", IEEE Communications Magazine 40, 8 (Aug.), 102-114.

[18] M. Handley et al., "SIP: Session Initiation Protocol," Internet Eng. Task Force RFC 2543, Mar. 1999; www.ietf.org/rfcrfc2543.txt.

[19] Peter Granström, Sean Olson and Mark Peck, "The future of communication using SIP," Ericsson Review, No. 1, 2002.

[20] R. Glitho, R. Hamadi and R. Huie, "An Architectural Framework for Using Java Servlet in a SIP environment," ICN 2001, July 2001, Colmar, France.

[21] W. V. Leekwijck and D. Brouns. "Siplets: Java-based service programming for IP telephony." ICIN 2000, pp. 22-27.

[22] "How Tomcat works: A Guide to Developing Your Own Java Servlet Container", Budy Kurniawan and Paul Deck. First Edition: April 2004.

[23] "Enterprise Servlets and J2EE", Jason Hunter and William Crawford.

http://www.onjava.com/pub/a/onjava/excerpt/java_servlets_ch12/index.html

[24] "Take control of the servlet environment", Thomas Davis and Craig Walker (*JavaWorld*) Part 2: Alternatives to servlet session management (December 21, 2000). http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-servlets.html

[25] B. Slimane, R. Glitho and R. Dssouli, A Business Model with a LINDA -- Based Broker for Service Provisioning in Mobile Ad Hoc Networks, 10th International Conference on Intelligence in Networks (ICIN 06), 29 May - 1 June, 2006, Bordeaux, France.

[26] Amy L. Murphy, Gian Pietro Picco and Gruia-Catalin Roman. "Lime: A Middleware for Physical and Logical Mobility", ICDCS-21, Phoenix, AZ, USA, April 16-19 2001, pp. 524-233.

[27] S. Abranowski et al., "I-centric Communications: Personalization, Ambient Awareness, and Adaptability for Future Mobile Services", IEEE Communications Magazine, pp. 63-69, September 2004.

[28] "WebSphere Application Server", http://www-306.ibm.com/software/webservers/appserv/was/

[29] "SIPMethod Application Server – SIP Application Server", http://www.micromethod.com/products/runtime.htm

[30] Telecommunication information networking architecture consortium (TINAC), http://www.tinac.com

[31]. Slimane Bah, Basel Ahmad, Roch Glitho, Ferhat Khendek, Rachida Dssouli, "A SIP Servlet Framework for Service Provisioning in Stand-Alone Mobile

Ad Hoc Networks", Submitted to IEEE Communications Magazine, Ad-hoc and Sensors Networks Series.

[32] CGI: Common Gateway Interface, http://www.w3.org/CGI

[33] Active Server Pages, http://msdn.microsoft.com/library

*Note: All the Web Pages were last accessed on February 10, 2007.*