MobileSocket: Toward Continuous Operation for Java Applications

Tadashi Okoshi[1], Masahiro Mochizuki[1], Yoshito Tobe[2], and Hideyuki Tokuda[1 3]
[1]Graduate School of Media and Governance, [2]Keio Research Institute at SFC,
[3]Faculty of Environmental Information, Keio University
5322 Endo, Fujisawa, Kanagawa 252–8520, Japan

Abstract—**This paper proposes "MobileSocket" which realizes session layer communication continuity support for Java Applications towards the continuous operations for mobile applications. In the mobile computing environment where mobile hosts moves around the network even during applications are communicating with the remote, maintenance of the communication continuity between the applications is significant. Not only the mobility support but the virtual circuit continuity support is required for communication continuity. Existing approaches have not provided the complete communication continuity for applications. "MobileSocket" is a user-level enhanced socket library written in Java, and provides library-based session layer mobility and virtual circuit continuity support for applications. Two mechanisms, Dynamic Socket Switching (DSS) and Application Layer Window (ALW) were developed for MobileSocket and enable a simple implementation. MobileSocket applications can be used in Java mobile applications and the agents, as well as for ordinary network applications. In this paper, after we clarify the communication continuity and existing approaches, we present the MobileSocket design, mechanism, and results of evaluation.**

I. Introduction

Several types of computing entities, such as users, hosts, applications[1], or even users' desktops[12] can "rove" around in the mobile computing environment. Carrying their own notebook computers, users rove around among several places. Their mobile hosts are disconnected from and reconnected to the different network segments any number of times, even while applications such as remote log-in or video conferences are active.

For instance, firstly a user may use his/her notebook computer in the office, with a TELNET application to log-in to the remote host and the video conference application with friends in another place. According to his/her schedule, he/she moves from the office to a meeting room with the notebook computer, disconnecting the host from the network once, and reconnecting it to a different network segment. Even in such a situation, the user may want to use TELNET and video conference applications continuously, after the host has moved to the different network. Without any mobility support in the host, both TELNET and video conference application cannot maintain their communication with the remote host or behave continuously. The user needs to reconfigure the applications manually, reconnecting sessions to the remote hosts or restarting the applications.

Particularly for network applications, maintenance of their **communication continuity** with the companion applications on the remote host is important in order to enable applications to maintain continuous behavior and to provide the users with continuous operation. Not only the **mobility** but the **virtual circuit continuity** support which retains the byte stream consistency of the virtual circuit sessions is the requirement for communication continuity.

Although some related works[9], [6], [3] on the network, transport, and session layers address either of mobility or virtual circuit continuity for applications, they require extra software components such as proxies, agents, and modifications to the existing protocols.

In this paper, we present **MobileSocket**, which is the user-level, pure Java[4]-based, enhanced Socket interface library. It provides both mobility and virtual circuit continuity for any Java applications which use `java.net.Socket` class[15] as their Inter Process Communication (IPC). By using MobileSocket, existing Java applications can obtain communication continuity without any source code modifications, while a Java-based adaptation scheme for mobility event allows the applications to behave adaptively.

MobileSocket is enforced by two special mechanisms: Dynamic Socket Switching (DSS) and Application Layer Window (ALW). Our library-based session layer approach allows MobileSocket to provide communication continuity with only user-level implementation. Moreover, as a serializable Java class library, it allows even applications with the active MobileSocket connections to be mobile across hosts.

In the remainder of this paper, we present our definitions and clarifications of mobility, virtual circuit continuity and communication continuity in Section 2. We describe the issues for the communication continuity realization and several approaches of related works in Section 3. Section 4 shows the design overview of MobileSocket and Section 5 describes the MobileSocket mechanism. Section 6 presents the performance evaluations of our implementation and Section 7 discusses the results of the functional comparisons with the related works.

II. Continuous Operation

In this section, we describe our definition of "mobility" and "virtual circuit continuity" for clarification of "communication continuity," and explain two connection redirection schemes: Explicit and Implicit Redirection.

A. Mobility

Fig.1 shows the definition of mobility. With "mobility", the mobile host can maintain the transparent host identifier in network protcol architecture, even after the host has been disconnected from a network and reconnected to a different network. The mobile host can be identified transparently from other hosts in the wide area network at a certain layer of the network structure, with any framework which supports mobility.

Several different approaches for mobility are possible at each layer of the layered network structure because there are multiple different identifiers in each layer, such as IP address, TCP connection (a pair of an IP address and a port number) or a socket descriptor.

Mobility: capability of the protocol functionality in the both communication endpoints to identify each other independent of the location changes of the endpoints

Fig. 1. Definition of Mobility

B. Virtual Circuit Continuity

Fig.2 shows the definition of virtual circuit continuity. With "continuity", applications in the mobile host can preserve their activities and can offer their own services to users, after the host moves to the different network (or even after the application moves to another host). Particularly in the case of network communication aspects, with "virtual circuit continuity", network applications using virtual circuit connections with the remote applications can maintain their connections and continue communication despite the location changes of the host or applications themselves. Reliability and accuracy of ordering each byte of data stream (we describe both of them as "byte stream consistency") in the connection between the endpoints are maintained for the applications.

Virtual Circuit Continuity: capability of keeping a virtual circuit connection between the applications alive retaining reliability and the order of the byte stream of the virtual circuit when the location of the host changes

Fig. 2. Definition of Virtual Circuit Continuity

C. Communication Continuity

Fig.3 shows the definition of communication continuity. Communication between the applications are mainly classified to datagram communication such as a UDP flow and a reliable virtual circuit communication such as TCP[11] connection. Each type of communication has different requirements for communication continuity, (1)Datagram Communication Continuity and (2)Virtual Circuit Communication Continuity.

Communication Continuity: capability of maintaining the communication between the applications despite the location changes of the host

(1)Datagram Communication Continuity: Communication continuity support for applications using a datagram communication is enabled only by the mobility support.

(2)Virtual Circuit Communication Continuity: Communication continuity support for applications using a virtual circuit connection is enabled by both the mobility and the virtual circuit continuity support.

Fig. 3. Definition of Communication Continuity

1) Datagram Communication Continuity: For applications which use connection-less datagram communications, including the network video conference application or the net-phone application, the mobility support is enough for datagram communication continuity. These types of applications do not need reliability and ordered data packets. The lack of these two characteristics is not critical, although the Quality of Service (QoS) they provide to users may be affected. For example, Mobile IP provides not only mobility but datagram communication continuity. When the mobile host is reconnected to a foreign network after the disconnection period, UDP/IP communication between the mobile and the correspondent hosts can be redirected, although the packets sent by the correspondent host to the mobile host during the mobile host's disconnection are lost in the network.

2) Virtual Circuit Communication Continuity: In contrast, mobility does not always imply virtual circuit communication continuity in the case of applications with virtual circuit communications. In this case, virtual circuit communication continuity can be achieved only with both mobility and virtual circuit continuity. In order to realize both characteristics at the same time, it is required to support not only the mechanism for mobility support but an additional mechanism which supports virtual circuit continuity.

Virtual circuit protocols typically guarantee the byte stream consistency of the connection. The TCP protocol, as an instance of virtual circuit connection protocol, uses acknowledgment packets for this functionality and exploits the retransmission timer which retransmits the data if the acknowledgment does not arrive from the remote host before the timer expires. The retransmission timeout period is fixed in the protocol stack and cannot be modified by applications in most major TCP implementations[13], although RFC 1122[2] requires the ability of modification. As a result, when the correspondent host sends data to the mobile host during the mobile host's disconnection, a TCP connection will be torn down after twelve times of retransmission or nine-minutes idle time.

In addition to the retransmission timer, if the TCP keep alive timer option is enabled, the TCP connection between the mobile and the correspondent hosts cannot be alive for greater than or equal to 2 hours 10 minutes. Thus, the mobile host cannot be disconnected for longer than this period.

D. Connection Redirection Schemes

There are two kinds of connection redirection schemes that provide applications with virtual circuit continuity: Implicit and Explicit Redirection.

1) Implicit Redirection: Implicit Redirection is a mechanism by which a connection is automatically redirected such that the application of the connection is unaware of the relocation of the host. Thereby additional lines for redirection in the source code are not required. Hence the existing applications can obtain communication continuity without any modification or re-compilation. A drawback of Implicit Redirection is the lack of adaptability in the behavior of the applications.

2) Explicit Redirection: Explicit Redirection is a mechanism by which the application programmers can explicitly specify where the redirection takes place. The programmers need to insert additional lines into their source code for the redirection. For instance, `suspend` and `resume` are used to specify temporal disconnection of a connection and resumption of the disconnected connection, respectively. A benefit with Explicit Redirection is accommodating an adaptive behavior of the applications with signals or events from the underlying mechanism.

III. Issues and Related Approaches

In this section, we describe issues for communication continuity and classify several related works.

A. Issues

We define four issues for the achievement of communication continuity support for applications toward the continuous operation.

1) Effective virtual circuit continuity: Virtual circuit continuity with byte stream consistency support for the applications should not depend on the specific protocol mechanism. Applications should be able to be disconnected from the network for the period they have configured without limitation of the underlying protocol.

2) Simplified and minimized implementation: Modification to existing protocol stacks usually in kernels and their reconfiguration in the hosts or the necessity of additional software components like servers and agents must be simplified and minimized.

3) Avoidance of modification in applications: It is effective for numerous applications to avoid modification, insertion of additional APIs into their source code or even re-compilation.

4) Interfaces for application adaptation: Despite the importance of compatibility with the existing applications, the schemes and the interfaces for the explicit redirection and adaptation for applications are also required for the adaptive behavior of the applications.

B. Related Approaches

There are some related works which intend to realize communication continuity. We classify them by the layer of the OSI reference model they use.

1) Network Layer Approach: A network layer protocol provides a global node identifier and an addressing scheme in the network as the basic unit of the end-to-end communication. Movement with the global node identifier enables end-to-end transparent reachability independent of the mobile host's relocation.

Mobile IP[9], [7], [8] Mobile IP is a mobile extension to IP. Using IP tunneling mechanism through Home Agent (HA) and Foreign Agent (FA), a mobile and a correspondent host can communicate with each other with a pair of the same IP addresses even after the mobile host's relocation.

Mobile IP, however, does not provide effective virtual circuit continuity in the case of TCP/IP applications because of the TCP functionality described in Section 2. A network layer approach requires an additional mechanism for virtual circuit continuity at the upper layer.

2) Transport Layer Approach: A transport layer approach is effective for communication continuity because a transport connection protocol can provide the end-to-end communication byte stream consistency.

TCP-R[3] TCP-R is a modification to TCP with mobility support. In TCP-R, a mobile host sends its new IP address to its correspondent host after the relocation, and then both hosts change IP destination address and port number inside the TCP control block. The TCP connection is kept alive even after the mobile host relocates, thereby the mobility in TCP layer is retained. Furthermore, TCP-R provides continuity for the TCP/IP connection. In TCP-R, the TCP state transition diagram is modified and "reconnect-timer" is introduced in addition to the retransmission timer. Using the reconnect-timer, applications can set the appropriate reconnection time-out for TCP, and TCP connection continuity is thus offered to the applications.

TCP-R itself does not guarantee network layer mobility, thus combination of TCP-R and Mobile IP provides more effective mobility such as establishment of a new connection after the mobile host's relocation.

3) Session Layer Approach: We here review an approach that is above the transport protocol or the session layer. It can be referred to as an application layer approach in TCP/IP suites[10].

MSOCKS[6] MSOCKS is the architecture for transport layer mobility. MSOCKS consists of a MSOCKS library in the mobile host and a proxy server which splits a TCP connection between the mobile host and the correspondent host.

MSOCKS requires neither the kernel implementation at the mobile host nor the modification in applications, by using the linked library replacement. However, it needs a proxy server with a modification in kernel, as well as MSOCK library in the mobile hosts. Since MSOCKS does not consider the retransmit timer in TCP, it is not possible to provide TCP/IP virtual circuit continuity for a period longer than a temporary disconnection from the network such as during network interface switching.

The advantage of the session layer approach is the unnecesity of the modifications of underlying protocols, such as TCP or IP. The approach allows implementation of the mobility support mechanism only at user-level: in servers or libraries. It is possible to accomplish communication continuity only with the libraries at the both endpoints, although MSOCK exploits the proxy aided implementation.

C. Discussion on Approaches

A network layer approach is suited for the mobility support, but it cannot provide complete connection continuity because of the semantics of the layered network architecture. A transport layer approach provides effective connection continuity for applications. But both approaches require the modification inside the existing protocol stack and complicate the implementation.

Our MobileSocket exploits the session layer approach for the communication continuity support with solving the issues described in Section 3. It provides effective virtual circuit continuity for applications. We describe our solution in detail in the following section.

IV. MobileSocket

In this section, we present the design overview, the functionalities and the applications of the MobileSocket.

A. Design Overview

The design goals of MobileSocket are (1)effective virtual circuit connection, (2)simplified and minimized implementation, (3)avoidance of modification in applications, and (4)interfaces for application adaptation.

MobileSocket realizes the session layer communication continuity support, by providing the applications with one persistent socket connection, while it switches the multiple actual socket connections internally. Two mechanisms, Dynamic Socket Switching (DSS) and Application Layer Window (ALW) described in the next section, support MobileSocket's functionality.

B. Java Library Implementation

MobileSocket is implemented as a class library in Java language. We use Java Development Kit (JDK)[15] 1.1.6 on FreeBSD 2.2.1R. The TCP MobileSocket implementation consists of approximately 1,800 lines of Java source code.

The MobileSocket class has backwards compatibility to the `java.net.Socket` class of JDK. By modifying the

CLASSPATH environment variable, existing Java applications with the `java.net.Socket` can use MobileSocket class without any modifications.

C. Redirection Support

MobileSocket offers both implicit and explicit operations of connection redirection to applications.

Implicit redirection scheme is prepared for the compatibility with the existing Java applications which use original Sockets. In this case, if the mobile host is disconnected from the network, the MobileSocket library detects it and invokes the implicit redirection and the application does not need to be aware of the movement of the host.

On the other hand, explicit redirection schemes, `MobileSocket#suspend` and `MobileSocket#resume` methods, are prepared for the mobility-aware applications. Using these methods, applications are able to suspend and resume their MobileSocket connections explicitly.

D. Adaptation Interface for Applications

Fig.4 shows the overview of MobileSocket Java event based adaptation interface for applications. This interface enables the adaptive application behavior triggered by the `MobilityEvent` from the MobileSocket object.

```
public class MobilityEvent extends AWTEvent{}

public interface MobilityListener
                          extends EventListener{
  public void ConnectionSuspended(MobilityEvent e)
  public void ConnectionResumed(MobilityEvent e)
}
```

Fig. 4. Overview of Java Event based Adaptation Interface

E. MobileSocket Application

Fig.5 shows an example of MobileVideoPlayer using the `MobileSocket` class. The name of the MobileSocket class is expressed as `MobileSocket` for clarification in the example. In the constructor and the `play()` method, there is no difference in the case of `java.net.Socket` class. Event handler methods and explicit redirection methods are optional and for the adaptive application behavior.

MobileSocket provides yet another communication continuity for Java mobile applications because our implementation of the MobileSocket class is "serializable". Object serialization[14] is one of the major characteristics of the Java language. If one Java object is an instance of the class which implements "serializable" interface, this object can be translated into a byte stream using `ObjectInput/OutputStream` classes, and can be sent to the remote host across the network through the ordinary byte stream socket connection. MobileSocket object itself or the application which uses MobileSocket internally can be sent from a host to another. With this characteristic, mobile Java applications can maintain their MobileSocket connection to the remote applications even after they are sent to another host by the object serialization. In this case, the application does not need to call explicit `suspend()` method before the object serialization because the explicit `suspend()` and `resume()` methods are called internally when the MobileSocket object is serialized and de-serialized.

```
import jp.ac.keio.sfc.ht.mobilesocket.*;

public class MobileVODPlayer
                     implements MobilityListener{

  public MobileVODPlayer(String hostname,
                         int port){
    makeGUIInterface();
    MobileSocket sock
       = new MobileSocket(hostname, port);
    sock.addMobilityListener(this);
    play();
  }

  public void play(){
    while(true){
     len = sock.getInputStream().read(VideoImage);
     drawVideoFrame(VideoImage);
    }
  }

  /*Java Event Handler Methods*/
  public void ConnectionSuspended(MobilityEvent e){
    Dialog.setText("Connection is suspended!");
  }
  public void ConnectionResumed(MobilityEvent e){
    Dialog.setText("Connection is resumed!");
  }

  /*Methods fo Explicit Redirection Operation*/
  public void suspend(){
    sock.suspend();
  }
  public void resume(){
    sock.resume();
  }
}
```

Fig. 5. MobileSocket Example Application

V. MobileSocket Mechanism

In this section, we present the mechanism of MobileSocket. After we describe the Dynamic Socket Switching (DSS) and Application Layer Windows (ALW), we detail the MobileSocket state diagram and DSS time sequence.

A. Dynamic Socket Switching (DSS)

Fig.6 shows the concept of the Dynamic Socket Switching (DSS) mechanism inside the MobileSocket library.

DSS allows the MobileSocket library to provide one persistent socket connection to the applications. Once a MobileSocket connection is established between the mobile and the correspondent hosts, the applications on both sides of the socket can read and write the transmitted byte streams with one persistent socket object, even after the mobile host's relocation. In contrast, inside the MobileSocket library, a new socket connection between the applications is created every time after the mobile host's relocation, and switched dynamically to preserve the virtual circuit connection between the libraries.

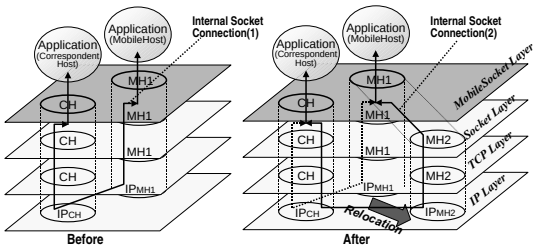

Fig. 6. Concept of DSS

B. Application Layer Window

Fig.7 shows the Application Layer Window (ALW) mechanism. ALW is a user-level sliding window implemented in the MobileSocket library and maintains the byte stream consistency of the MobileSocket connection. After the mobile host's reconnection with the implicit redirection operation, the user data already written by the application can remain in the lost socket connection between two MobileSocket libraries, in the buffers of the local protocol stacks, in the network, and in the buffers of protocol stacks in the remote host. This causes byte stream inconsistency of the MobileSocket connection. ALW keeps the byte stream consistency of the MobileSocket by re-sending the lost data after the reconnection.

While the MobileSocket connection is established, the libraries at both ends of the connection communicate with each other with ALW-ACK, the acknowledgment for ALW. As the user data is sent from the sender to the receiver, the data is stored in the ALW of the sender. On the other hand, in the receiver, the number of bytes the library read from the DataSocket is stored in ALW_COUNTER. When the value of ALW_COUNTER becomes equal to the ALW length, the receiver sends ALW_ACK to the sender over ControlSocket. At the sender, after ALW is filled up with the user data, library waits for ALW_ACK from the receiver. The sender is able to write more user data only after it receives ALW_ACK.

In the phase of DSS implicit resuming during Implicit Redirection, both MobileSocket libraries exchange the number of bytes they individually have read. If there is difference between the number of bytes the host written and the number of bytes the remote read, it means that the user data has been lost. By sending the user data stored in ALW to the remote again, MobileSocket maintains the byte stream consistency.

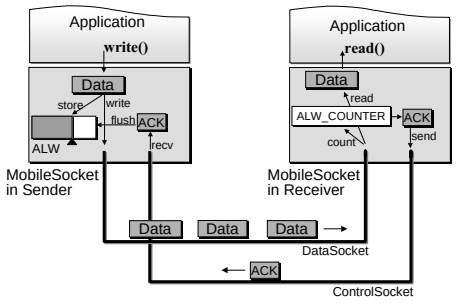

Fig. 7. Application Layer Window

C. MobileSocket State Transitions

Fig.8 shows the state transitions of MobileSocket. There are mainly four states in MobileSocket, "Closed", "Established", "ImplicitlySuspended", and "Explicitly-Suspended".

In "Closed" state, the MobileSocket connection is not connected to the remote host. In "Established" state, the connection between two MobileSocket libraries is established and applications at the both ends can communicate with each other through the MobileSocket. In "Explicitly-Suspended" state, the connection between the libraries is disconnected after the explicit `suspend` API is called by the application. The applications cannot communicate with each other unless they call `resume` API of MobileSocket. In "ImplicitlySuspended" state, the MobileSocket connection is disconnected implicitly by the libraries itself without any explicit API called from the applications.

In the state transition of MobileSocket, Closed state transits to Established state by connecting the initial socket connection. State transitions between Established and Explicitly Suspended are triggered by calling `suspend()` and `resume()` interfaces at the mobile host. Transitions between Established and Implicitly Suspended are triggered by the mobile host's sensing of the IP address reconfiguration.

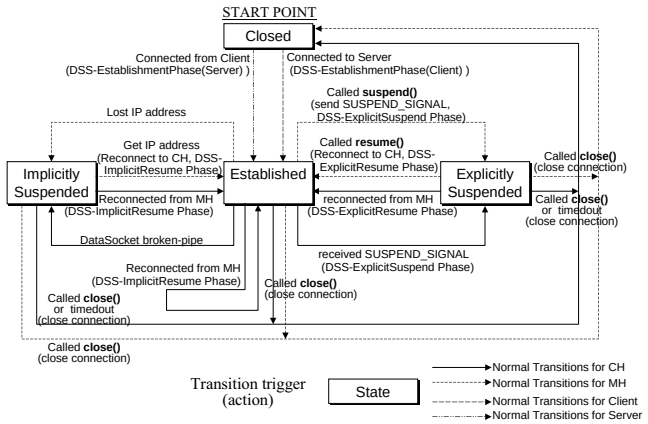

Fig. 8. MobileSocket State Transition Diagram

D. DSS Time Sequence

In Dynamic Socket Switching (DSS), there are four distinguished phases, "DSS-EstablishmentPhase", "DSS-ExplicitSuspendPhase", "DSS-ExplicitResumePhase", and "DSS-ImplicitResumePhase". Fig.9 shows the overview of the DSS time sequence at the connection establishment, suspending, and resuming.

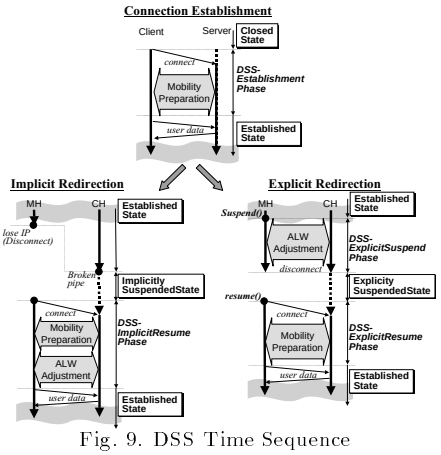

Fig. 9. DSS Time Sequence

1) DSS-EstablishmentPhase: DSS-EstablishmentPhase is performed whenever the MobileSocket connection is being established. Fig.10 shows DSS-EstablishmentPhase.

DSS-EstablishmentPhase is described as follows.

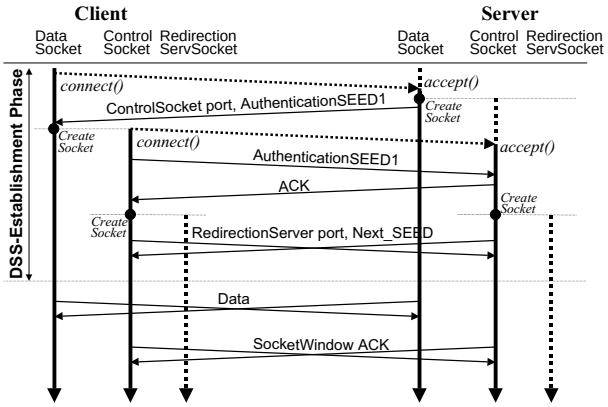

Fig. 10. DSS-EstablishmentPhase

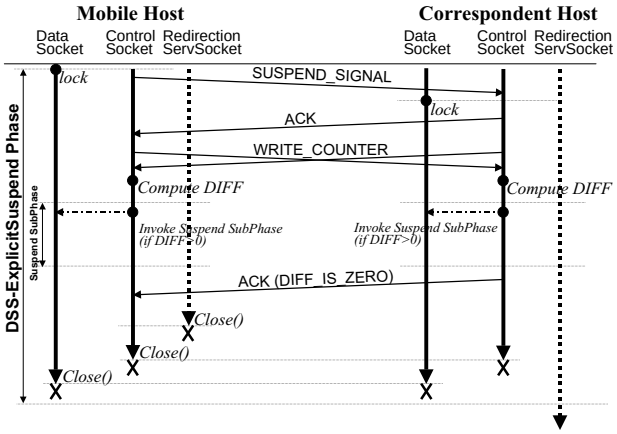

Fig. 11. DSS-ExplicitSuspendPhase

(1) The client connects a DataSocket connection to the server.

(2) The server starts ControlSocket, a server socket, after the DataSocket acceptance, and sends its port number and a seed for authentication to the client.

(3) The client makes a ControlSocket connection to the server with the port number and seed the client just received.

(4) After the authentication has succeeded, both sides create RedirectionServerSocket, which is a server socket for the next connection after the mobile host relocation.

(5) The client and the server exchange the port numbers and the authentication seeds of RedirectionServerSockets.

(6) Actual byte stream communication between applications starts.

Relation between the client and the server does not depend on which side will be the Mobile Host (MH) that suspends and resumes connection, and which side will be the Correspondent Host (CH) that is suspended and resumed connection by the MH. Therefore the libraries at both sides create RedirectionServerSocket for mobility.

2) DSS-ExplicitSuspendPhase: DSS-Explicit SuspendPhase is triggered by `suspend()` API (Java method) called from the application at the MH. In this phase MobileSocket locks writing and reading to and from the socket, confirms that all of byte stream data was read by remote host, and closes the connection. Fig.11 shows the time sequence of DSS-ExplicitSuspendPhase.

DSS-ExplicitSuspendPhase is described as follows.

(1) As `suspend()` API is called by the application on the MH, the MH informs the CH about the explicit suspend phase by sending SUSPEND_SIGNAL through the ControlSocket.

(2) After both sides of the connection have locked the stream, they exchange WRITE_COUNTER which indicates the number of bytes the host wrote to the socket.

(3) Each side calculates the difference between its own READ_COUNTER and the WRITE_COUNTER from the remote.

(4) The library unlocks reading from the socket once if there is any difference because it means that the host should read this "difference" of bytes more. Confirming that the application has read appropriate bytes of data, the library locks reading again.

(5) After the MH makes sure that both the MH and the CH have locked the stream finally, it close both DataSocket and ControlSocket connection.

3) DSS-ExplicitResume Phase: DSS-ExplicitResumePhase is triggered by `resume()` API called from the application at the MH, when the MH is in "ExplicitlySuspended" state.

In this phase, the MH reconnects to the RedirectionServerSocket of the CH with a new DataSocket connection. Apart from the initial authentication checking, this phase is identical to the DSS-EstablishmentPhase. Fig.12 shows the time sequence of DSS-ExplicitResumePhase.

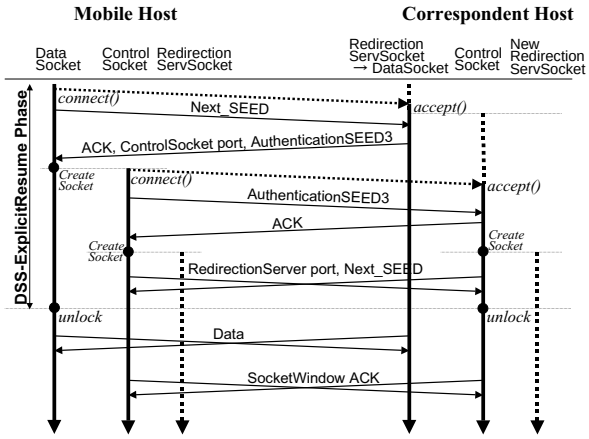

Fig. 12. DSS-ExplicitResumePhase

DSS-ExplicitResumePhase is described as follows.

(1) The MH creates new DataSocket connection to the RedirectionServerSocket of the CH.

(2) If the authentication succeeds, the port number of the ControlSocket server and the seed for ControlSocket is sent to the MH from the CH.

(3) With these port and seed, the MH establishes a ControlSocket connection to the CH.

(4) After the authentication checking, the MH and the CH exchanges their next RedirectionServerSocket's port number and seed.

(5) Applications at the both ends restart their communication after MobileSocket unlocked the connection.

4) Implicit Suspending and DSS-ImplicitResumePhase: When MobileSocket detects that the host has lost its IP address, the library transits into "ImplicitlySuspended" state. And the DSS-ImplicitResumePhase is triggered by sensing the host's reconnection to the network. In DSS-ImplicitResumePhase, after the MH obtains a new IP address, the MH connects to the RedirectionServerSocket of CH and reconstructs the MobileSocket connection, supported by ALW retransmission. Fig.13 shows the time sequence of the implicit suspending and DSS-ImplicitResumePhase.

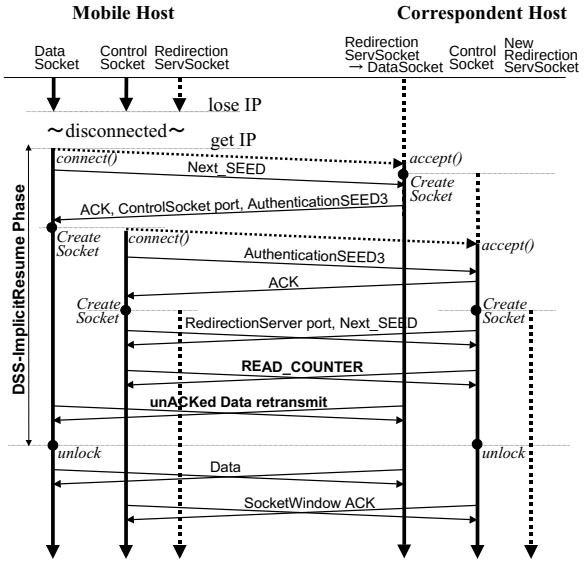

Fig. 13. Implicit Suspending and DSS-ImplicitResumePhase

(1) After MobileSocket in the MH senses obtaining a new IP address, the MH establishes a new DataSocket connection to the CH's RedirectionServerSocket.

(2) When the CH accepts this connection, the CH switches the socket and treats this socket as a new DataSocket.

(3) After the authentication checking, the CH sends the port number of ControlSocket and the next seed back to the MH as well as starting the ControlSocket server.

(4) After the authentication checking of ControlSocket, both sides exchange READ_COUNTERs, which indicate the number of bytes each host already read from the last internal socket connection.

(5) Both of the MH and the CH calculate the difference between their own WRITE_COUNTER and the READ_COUNTER from remote individually and retransmit the "difference" bytes of data to the remote from their own ALW.

(6) Both libraries unlock the DataSockets and applications restart communicate with the new socket.

TABLE I
Specification of Hosts for Performance Evaluation

Host	Mobile Host	Correspondent Host
PC	Dynabook SS-R590 (TOSHIBA)	VAIO PCG-737 (SONY)
CPU	Pentium 90MHz	MMX Pentium 233MHz
Memory	40MB	40MB
OS	FreeBSD 2.2.1-RELEASE with PAO-970616	
JavaVM	JDK 1.1.6.V98-7-21 for FreeBSD	

VI. Performance Measurement

In this section, we present the performance evaluation of the connection redirection in the MobileSocket library.

A. Evaluation Environment

Table 1 shows the platforms on which we evaluated MobileSocket. The mobile host and the correspondent host are connected through an isolated 10-Mbps Ethernet. In both of these hosts, we use FreeBSD 2.2.1-RELEASE version with PAO-970616[5], PC Card support package, and JDK 1.1.6. The following results are the mean values of 100 measurements.

B. Evaluation: Explicit Suspending and Resuming

We measured the time consumed by the `MobileSocket.suspend()` method, the explicit API to suspend MobileSocket connection, and the `MobileSocket.resume()` method, the explicit connection resuming API. After the two Java applications establish a MobileSocket connection, we measured the time with the `suspend()` and `resume()` method at the mobile host.

Table 2 shows the detailed times which are consumed in each process of DSS-ExplicitSuspend Phase, and Table 3 shows those of DSS-ExplicitResume Phase. The `suspend()` method consumes 46.67 milliseconds, and `resume()` consumes 270.28 milliseconds.

In the DSS-ExplicitSuspend Phase, except for waiting for an ACK from the correspondent host, locking of the Socket and killing of sub thread consumes a relatively high propotion of the time for the whole operation. The mutual exclusion class, used in the locking part, is made for the serializable class, in order to make MobileSocket class serializable, and this results in an overhead. Thread termination in Java depends on the implementation of the Java Virtual Machine. Concernin the wait for the acknowledgment from the correspondent host, the two MobileSocket libraries at both ends need to confirm that all data bytes written into the socket have already been read by the remote library. Therefore, the time for synchronization is needed in both libraries.

In the DSS-ExplicitSuspend Phase, the establishment of three internal sockets is a large overhead and consumes 82.14% of the whole operation. In contrast, we can optimize the rest (approximately 20% of operation) by polishing our implementation, while the socket performance depends on the Java compiler and the Java Virtual Machine (VM).

VII. Discussion

In this section, we present our functional comparison between MobileSocket and some related research described in Section 3. Table 4 shows the result of functional comparisons. We compared these works from the view points of (1)mobility, (2)virtual circuit continuity, (3)implementation, and (4)application.

MobileSocket provides Socket upper layer mobility and virtual circuit continuity without the limitation of the TCP timers. The MobileSocket library simplifies and minimizes the implementation for the relocation in a

TABLE II
Detail of DSS-ExplicitSuspend Phase

Steps	Time (msec)	Percentage (%)
manage Phase Transition	1.76	3.77
lock Socket	7.40	15.86
kill Sub-Thread	8.12	17.40
send SUSPEND_SIGNAL	1.17	2.50
send WRITE_COUNTER	5.35	11.46
receive ACK from CH (wait for process in CH)	11.01	23.59
receive port number	1.11	2.38
receive Authentication Seed	1.85	3.96
close Socket	3.28	7.03
prepare Info. of Next Socket	1.02	2.19
Miscellaneous	4.60	9.86
Total	46.67	100.00

TABLE III
Detail of DSS-ExplicitResume Phase

Steps	Time (msec)	Percentage (%)
make new DataSocket	80.75	29.88
switch Socket in Stream	0.36	0.13
Auth. Check for DataSocket	2.95	1.09
receive port of ControlSocket	1.11	0.41
receive Authentication Seed	1.89	0.70
make new ControlSocket	80.80	29.90
Auth. Check for ControlSocket	3.30	1.22
make new NextServerSocket	60.44	22.36
exchange of Next-port and AuthSeed.	6.62	2.45
restart Sub Thread	26.56	9.83
manage Phase Transition	0.90	0.33
Miscellaneous	4.60	1.70
Total	270.28	100.00

wide area network and for longer disconnection, while MSOCKS focuses on the local relocation and short disconnection periods. MobileSocket has interfaces for the adaptive application behavior and also can be used for the Java mobile applications.

VIII. Conclusion and Future Work

In this paper, we have presented MobileSocket, a user level library-based solution for application communication continuity support. The session layer approach and the user-level library installation in the mobile and the correspondent hosts simplify and minimize the implementation. The combination of Dynamic Socket Switching and Application Layer Window achieves the byte stream consistency for the TCP Socket connection. The Java event-based adaptation interfaces of MobileSocket realize the application level adaptation to the mobile host's relocation. According to our functional comparison between some related works and MobileSocket, MobileSocket provides applications the communication continuity and the adaptation interface despite its simple implementation.

We have two future works. The first is the optimization of the implementation. Performance of socket creation in Java, which is the current serious overhead at the redirection phase of MobileSocket, should be optimized and improved. The other is the application of the user level approach for the communication continuity to other resources, such as the file descriptor or the host specific devices. This will be effective for mobile applications and agents, which dynamically migrate between the hosts with the IPCs or local resources left opened.

TABLE IV
Functional Comparison

Name	(1)Mobility		
	Layer	Mobile Server Situation	Simultaneous Relocation
Mobile IP	IP	Yes	Yes
TCP-R	TCP	Limited(*1)	Limited(*2)
MSOCKS	Socket	No	No
MobileSocket	Socket Upper	Limited(*1)	Limited(*2)

Name	(2)Virtual Circuit Continuity		(3)Implementation			(4)Application		
		Redirection Scheme	MH	Additional Software	CH	Application Modification	Adaptation Interface	for Mobile Application
Mobile IP	Limited(*3)	N/A	M(IP), A(daemon)	HA,FA	unnecessary	No	N/A	N/A
TCP-R	Yes	Implicit	M(TCP)	unnecessary	M(TCP)	No	No	N/A
MSOCKS	Limited(*3)	Implicit	A(library)	Proxy(M(TCP))	unnecessary	No	N/A	N/A
MobileSocket	Yes	Implicit / Explicit	A(library)	unnecessary	A(library)	No	Yes	Yes

"M(x)"... modify x, "A(x)"... add x
Limited(*1)... Only after the connection is established, Server can relocate, otherwise it needs Mobile IP.
Limited(*2)... Only when it works with Mobile IP.
Limited(*3)... Limited to the TCP timers.

Acknowdledgment

The authors are grateful to members of the R3/RT-HDI Project for valuable comments and criticisms, especially Nobuhiko Nishio, Kazunori Sugiura, and Jin Nakazawa supplied ideas for extending this research.

We thank anonymous IC3N'99 reviewers for several comments and suggestions that helpd improve the quality of this paper. Our special thanks to Antony Rowstron, Tim Edmonds, Radina (Jiny) Stefanova, and Sheng Feng Li, who gave us valuable comments and suggestions on this research and the draft of this paper.

References

[1] K. Bharat and L. Cardelli, "Migratory applications," in *ACM Symposium on User Interfaces Software and Technology*, Nov. 1995.

[2] R. Braden, "Requirements for internet hosts – communication layers," 1989, RFC 1122, Internet Request For Comments.

[3] D. Funato, K. Yasuda, and H. Tokuda, "TCP-R: TCP mobility support for continuous operation," in *Proceedings of IEEE International Conference on Network Protocols 97*, pp. 229–236, Oct. 1997.

[4] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison Wesley, Reading, Massachusetts, 1996.

[5] T. Hosokawa, "PAO: FreeBSD mobile computing package ," http://www.jp.freebsd.org/PAO/.

[6] D. A. Maltz and P. Bhagwat, "MSOCKS: an architecture for transport layer mobility," in *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 1037–1045, 1998.

[7] C. Perkins, A. Myles, and D. B. Jonson, "IMHP: A mobile host protocol for the internet," *Computer Networks and ISDN System*, vol. 27, pp. 479–491, 1994.

[8] C. Perkins, A. Myles, and D. B. Jonson, "The Internet Mobile Host Protocol(IMHP)," in *Proceedings of INET'94/JENC5*, 1994.

[9] C. Perkins, "IP mobility support," Oct. 1996, RFC 2002, Internet Request For Comments.

[10] J. Postel, "Internet Protocol," 1981, RFC 791, Internet Request For Comments.

[11] J. Postel, "Transmission Control Protocol," 1981, RFC 793, Internet Request For Comments.

[12] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33–38, May 1998.

[13] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated, Volume 2*, Addison Wesley, Reading, Massachusetts, 1995.

[14] Sun Microsystems Inc., "Object serialization specification," 1996.

[15] Sun Microsystems inc., "Java Developpers Kit (JDK) version 1.1.6)," 1997, http://www.javasoft.com/.