Vol. 41 No. 2 Transactions of Information Processing Society of Japan Feb. 2000

Regular Paper

MobileSocket: Session Layer Continuous Operation Support for Java Applications

Tadashi Okoshi,[†] Masahiro Mochizuki,[†] Yoshito Tobe[††] and Hideyuki Tokuda[†††,†]

This paper proposes the session layer communication continuity support for Java applications toward a continuous operation for the users. In a mobile computing environment, mobile hosts move around the different network segments even during applications communicate with the remote endpoint. In such a situation, maintenance of the communication continuity between the applications is significant. In order to retain the communication continuity, not only the mobility support but the virtual circuit continuity support is required for applications. Existing approaches on the network, the transport and the session layers do not provide the complete mobility and virtual circuit continuity for applications, although they require the complicated implementation. "MobileSocket" is a user-level enhanced socket library written in Java, and provides the library-based session layer mobility and virtual circuit continuity support for applications. Two mechanisms, Dynamic Socket Switching (DSS) and Application Layer Window (ALW) enforce MobileSocket and enable the implementation simplicity. MobileSocket applications can be used in the Java mobile applications and the agents, as well as in ordinary network applications. In this paper, after we clarify the communication continuity and existing approaches, we present the MobileSocket design, mechanism, and evaluation results.

1. Introduction

Several types of computing entities, such as users, hosts, applications [1], or even users' desktops [2] can "rove" around in the mobile computing environment. Carrying their own notebook computers, users rove around between the places. Their mobile hosts are disconnected from and reconnected to the different network segments any number of times, even during applications such as remote log-in, video conferences, etc., are active.

For instance, firstly a user uses his/her notebook computer in the office with TELNET application to log-in the remote host and the video conference application with friends in other places. Due to the 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 the different network segment in the meeting room. Even in such a situation, this 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 endpoint or behave continuously. The user needs to reconfigure the applications manually, reconnecting the session to the remote endpoint or restarting the applications.

Particularly for the network applications, maintenance of their **communication continuity** with the companion applications on the remote hosts is significant, in order to enable applications to maintain continuous behavior and to provide the users with continuous operation. Not only the **mobility** support often enabled by the transparent network identifier but the **virtual circuit continuity** support which retains the byte stream consistency of the virtual circuit sessions are the requirements for communication continuity.

Although some related works [3)~5)] on the network, transport, and session layers address either of mobility and virtual circuit continuity for applications, they require extra software components such as proxies, agents, and the modifications to the existing protocols.

In this paper, we present **MobileSocket**, which is the user-level, pure Java [6]-based, enhanced Socket interface library. It provides both mobility and virtual circuit continuity for any Java applications which use

† Graduate School of Media and Governance, Keio University
†† Keio Research Institute at SFC, Keio University
††† Faculty of Environmental Information, Keio University

`java.net.Socket` class [7] as their Inter Process Communication (IPC). By using MobileSocket, existing Java applications can obtain communication continuity without any modifications in their source code, while the Java-based adaptation scheme for the mobility event allows 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, serializable Java class library allows even applications with the active MobileSocket connections to be mobile.

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 the related works in Section 3. Section 4 shows the design overview of MobileSocket and Section 5 describes 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. Finally, we address our future work and conclude this paper in Section 8.

2. 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.

2.1 Mobility

Figure 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.

> **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.

2.2 Virtual Circuit Continuity

Figure 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 a 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 communicating despite the location changes of the host or applications themselves. Reliability and order accuracy of 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.

2.3 Communication Continuity

Figure 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 connection. Each type of communication has different requirements for communication continuity, (1) Datagram Communication Continuity and (2) Virtual Circuit Communication Continuity.

2.3.1 Datagram Communication Continuity

For applications which use connection-less datagram communications, including the network video conference application or the netphone application, the mobility support is enough for datagram communication continu-

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.

ity. Those types of applications do not need reliability and ordered data packet in their communication. The lack of those two characteristics is not critical, although 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 reconnects 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.

2.3.2 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. For instance, Mobile-IP provides IP layer mobility, but does not provide complete virtual circuit continuity for the applications with TCP/IP connections. In this case, the retransmission timer and the keep alive timer of TCP protocol cause problems and they limit the virtual circuit continuity provision of Mobile-IP.

Virtual circuit protocol typically guarantees the byte stream consistency of the connection. 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 has not arrived from the remote before the timer expiration. The retransmission timeout period is fixed in the protocol stack and cannot be modified by applications in most of major TCP implementations [8], although RFC 1122 [9] 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 over this period.

2.4 Connection Redirection Schemes

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

2.4.1 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 a source code are not required. Hence, the existing applications can obtain communication continuity without any modification or compilation. A drawback of Implicit Redirection is the lack of adaptability in the behavior of the applications.

2.4.2 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 to 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 associated with Explicit Redirection.

3. Issues and Related Approaches

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

3.1 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 the 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.

3.2 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.

3.2.1 Network Layer Approach

A network layer protocol provides a global node identifier and an addressing scheme in the network and 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 [3]**:** 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 the same IP address 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 2.3.2. A network layer approach requires an additional mechanism for virtual circuit continuity at the upper layer.

3.2.2 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 [5]**:** 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 the 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 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 to TCP, and the TCP connection continuity is 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.

3.2.3 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 [4]**:** 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 wells 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 the network interface switching.

The advantage of the session layer approach is the unnecessity of the modifications of underlying protocols, such as TCP or IP. The approach allows to implement the mobility support mechanism only at user-level, such as 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.

3.3 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 3.1. It provides effective virtual circuit continuity for applications. We describe our solution in detail in the following section.

4. MobileSocket

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

4.1 Design Overview

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, as described previously.

MobileSocket realizes the session layer communication continuity support, providing the applications with the 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.

4.2 Java Library Implementation

MobileSocket is implemented as a class library in Java language. We use Java Development Kit (JDK)[7] 1.1.6 on FreeBSD 2.2.1R.

```
public class MobilityEvent
                        extends AWTEvent{}

public interface MobilityListener
                    extends EventListener{
 public void MSSuspended(MobilityEvent e)
 public void MSResumed(MobilityEvent e)
}
```

Fig. 4 Overview of Java-event based adaptation interface.

The TCP MobileSocket implementation consists of approximately 1,800 lines of Java source code.

MobileSocket class has the upper compatibility to the `java.net.Socket` class of JDK. Modifying the CLASSPATH environment variable, existing Java applications with the `java.net.Socket` can use MobileSocket class without any modification to them.

4.3 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 ordinary Socket originally. In this case, if the mobile host is disconnected from the network, 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.

4.4 Adaptation Interface for Applications

Figure 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.

4.5 MobileSocket Application

Figure 5 shows an example of MobileVideoPlayer with `MobileSocket` class. The name of 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

```
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){
     int len
       = sock.getInputStream()
                .read(VideoImage);
     drawVideoFrame(VideoImage);
    }
  }

  /*Java Event Handler Methods*/
  public void MSSuspended(
                        MobilityEvent e){
    Dialog.setText("EVENT: suspended!");
  }
  public void MSResumed(MobilityEvent e){
    Dialog.setText("EVENT: resumed!");
  }

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

Fig. 5 MobileSocket example application.

`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 MobileSocket class is "serializable". Object serialization [11] is one of the major characteristics of Java language. If one Java object is an instance of the class which implements "serializable" interface,

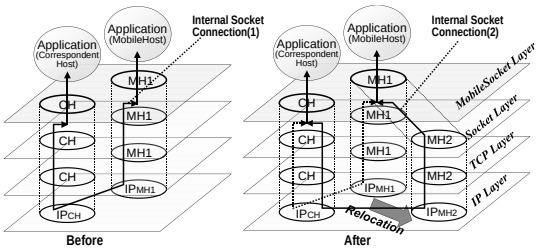


Fig. 6 Concept of DSS.

this object can be translated into 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 deserialized.

5. MobileSocket Mechanism

In this section, we present the mechanism of MobileSocket. After we describe the DSS and ALW, we detail the MobileSocket state diagram and DSS time sequence.

5.1 Dynamic Socket Switching (DSS)

Figure 6 shows the concept of 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 in the both sides of the socket can read and write the byte stream of each other with one lasting socket object, even after the mobile host's relocation. In contrast, inside the MobileSocket library, a new socket connection between both applications is created every time after the mobile host's relocation, and switched dynamically to preserve the virtual circuit connection between the libraries.

5.2 Application Layer Window (ALW)

Figure 7 shows the 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 in both ends of the connection communicate with each other with ALW-ACK, the acknowledgment for ALW. As each 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.

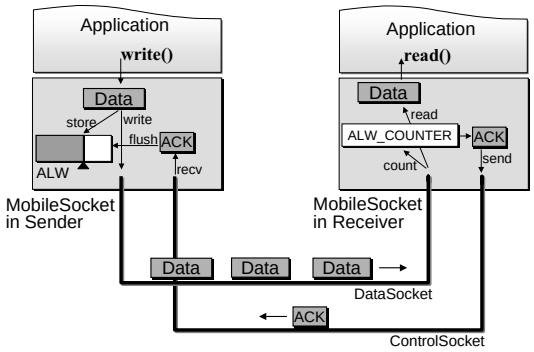


Fig. 7 Application layer window.

In the case of Implicit Redirection, it is possible that some user data already sent by the sender cannot be read by the receiver side. To maintain the same TCP socket semantics for the applications as the normal socket interface, it is necessary to provide the byte stream consistency. In the phase of DSS implicit resuming, both MobileSocket libraries exchange the number of bytes they individually have read. If there is difference between the number of bytes the host wrote and the number of bytes the remote read, it means that the user data is lost. By sending the user data stored in ALW to the remote again, MobileSocket achieves the byte stream consistency.

5.3 MobileSocket State Transition

Figure 8 shows the state transition of MobileSocket. There are mainly four states in MobileSocket, "Closed", "Established", "ImplicitlySuspended", and "ExplicitlySuspended".

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 communi-

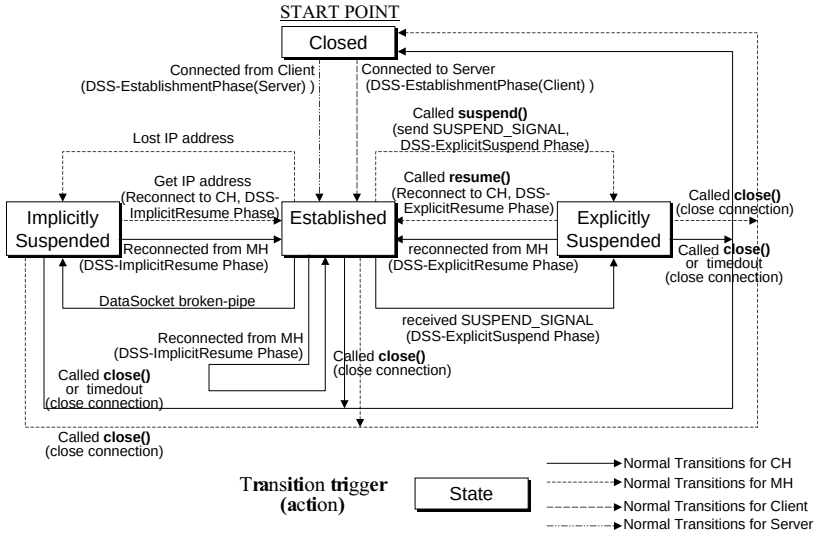


Fig. 8 MobileSocket state transition diagram.

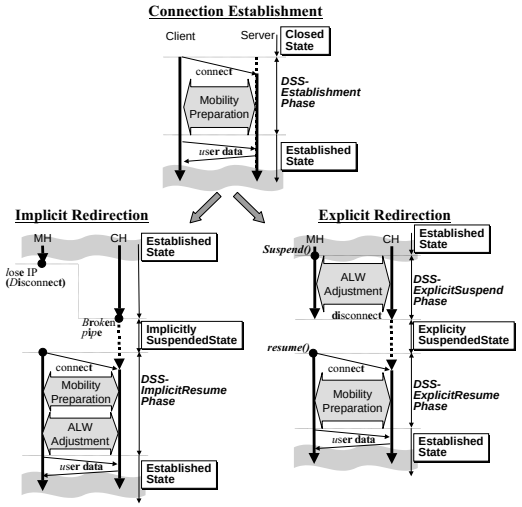


Fig. 9 DSS time sequence.

cate with each other through the MobileSocket. In "ExplicitlySuspended" 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.

5.4 DSS Time Sequence

In DSS, there are four distinguished phases, "DSS-Establishment Phase", "DSS-Explicit SuspendPhase", "DSS-Explicit ResumePhase", and "DSS-Implicit ResumePhase". **Figure 9** shows the overview of DSS time sequence at the connection establishment, suspending, and resuming.

5.4.1 DSS-EstablishmentPhase

DSS-EstablishmentPhase is performed whenever the MobileSocket connection is being established. **Figure 10** shows DSS-EstablishmentPhase.

DSS-EstablishmentPhase is described as follows.

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

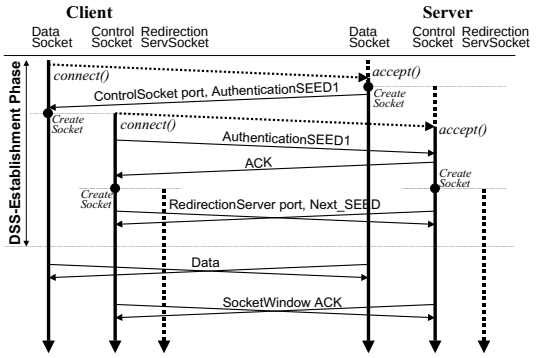


Fig. 10 DSS-EstablishmentPhase.

(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 client just received.

(4) After the authentication has succeeded, the 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 the mobility.

5.4.2 DSS-ExplicitSuspendPhase

DSS-ExplicitSuspendPhase 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 connection.

Figure 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

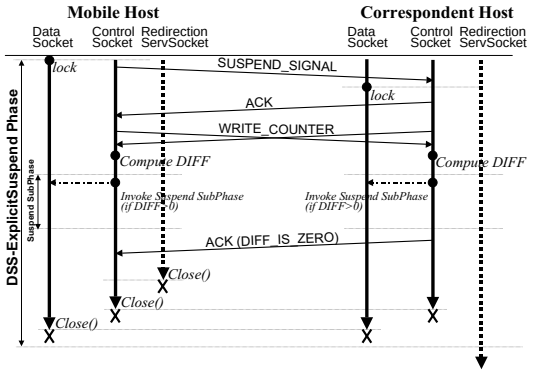


Fig. 11 DSS-ExplicitSuspendPhase.

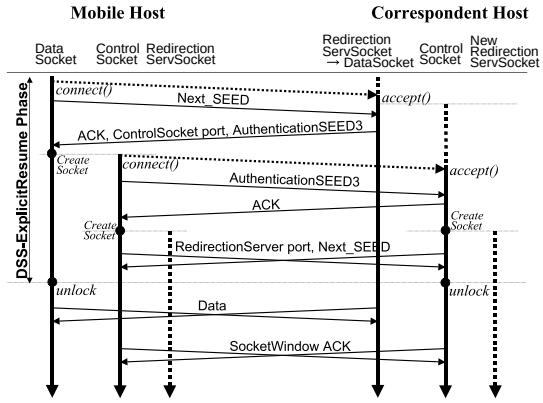


Fig. 12 DSS-ExplicitResumePhase.

sending SUSPEND_SIGNAL through the ControlSocket.

(2) After both sides of 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 unlocked reading from the socket once if there is any difference because it means that the host should read this "difference" of bytes additionally. Confirming that the application has read the 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.

5.4.3 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. Except the initial authentication checking, this phase is just like DSS-EstablishmentPhase. **Figure 12** shows the time sequence of 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 has succeeded, 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.

5.4.4 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. **Figure 13** shows the time sequence of the 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) As 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 starts ControlSocket server.

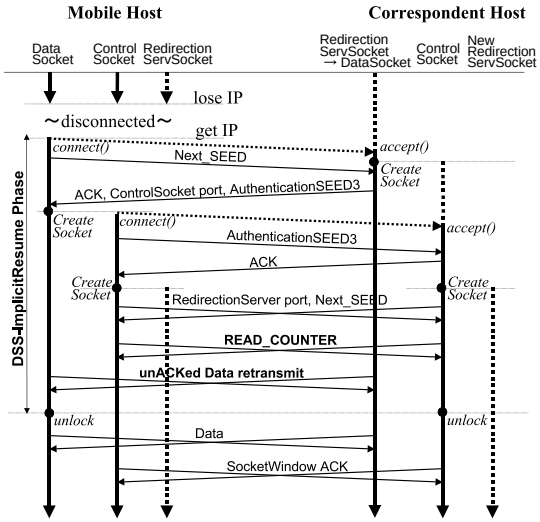


Fig. 13 Implicit Suspending and DSS-ImplicitResumePhase.

(4) After the authentication checking of ControlSocket, the 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 to communicate with the new socket.

6. Performance Measurement

In this section, we present the performance evaluation of the connection redirection in the MobileSocket library. We can see some overheads which can be reduced more by source code optimization, while the performance of internal socket depends on the Java environment.

6.1 Evaluation Environment

Table 1 shows the platform we evaluated MobileSocket. The mobile host and the correspondent host are connected through an isolated 10Mbps Ethernet. In both of these hosts, we use FreeBSD 2.2.1-RELEASE version with PAO-970616 [12], PC Card support package, and Java Development Kit (JDK) 1.1.6. The following results are the mean values of 100 times measurements.

Table 1 Specification of hosts for performance evaluation.

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

Table 2 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 3 Detail of DSS-ExplicitResume phase.

Steps	Time (msec)	Percentage (%)
make new DataSocket	80.75	29.88
switch Socket in Stream	0.36	0.13
Authentication 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
Authentication 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

6.2 Evaluation 1: Explicit Suspending and Resuming

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

Result

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. `suspend()` consumes 46.67 milli-seconds, and `resume()` consumes 270.28 milli-seconds.

In the DSS-ExplicitSuspend Phase, except the waiting for ACK from the correspondent host, locking of Socket and killing of sub thread spends relatively higher ratio of 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 it causes overhead. Thread termination in Java depends on the implementation

Table 4 Detail of DSS-ImplicitResume phase.

Steps	Time (msec)	Percentage (%)
make new DataSocket	78.39	24.67
Authentication Check for DataSocket	3.34	1.05
receive port of ControlSocket	1.10	0.35
receive Authentication Seed	1.82	0.57
make new ControlSocket	79.57	25.04
Authentication Check for ControlSocket	3.53	1.11
make new NextServerSocket	58.47	18.40
exchange of Next-port and AuthSeed.	6.68	2.10
exchange of READ_COUNTER	3.69	1.16
resend unACKed Data from ALW	0.21	0.07
restart Sub-Thread1	48.03	15.12
restart Sub-Thread2	25.91	8.16
manage Phase Transition	0.89	0.28
switch Socket in Stream	0.22	0.07
Miscellaneous	5.93	1.85
Total	317.78	100.00

of Java Virtual Machine. Concerning about the waiting for the acknowledgment from the correspondent host, two MobileSocket libraries in the both ends of the connection need to confirm that all data bytes written into the socket as been already read by the remote library. Therefore, time for synchronization is needed in the both libraries.

In the DSS-ExplicitSuspend Phase, establishments of three internal sockets have large overhead and consume 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 Java compiler and the Java Virtual Machine (VM).

6.3 Evaluation 2: Implicit Resuming

We also measured the time consumed in the implicit resuming phase of MobileSocket. In this case, only DSS-ImplicitResumePhase is performed between the two hosts at the period of mobile host's reconnection.

In our measurement, by removing PC-Card Ethernet card from the PC-Card slot, we disconnected the mobile host from network after the initial MobileSocket connection was established. Then, we measured the time of the operation when we connected the mobile host to the network again.

Result

Table 4 shows the result of this measurement. DSS-ImplicitResumePhase needs 317.78 milli-seconds. Also in this phase, three internal sockets' creation consumes 216.43 milli-seconds, approximately 68% of all. And two sub-threads creation consumes 73.94 milli-seconds, approximately 23%. These creation of sub-threads can be reduced by optimizing the source code and the class structure, while we cannot avoid the sockets' creation in MobileSocket and performance of making socket depends on the performance of Java environment.

7. Discussion

In this section, we present our functional comparison between MobileSocket and some related works described in Section 3.2. **Table 5** and **Table 6** show the results of functional comparisons. We compared these works from the view points of (1) mobility, (2) virtual circuit continuity, (3) implementation, and (4) application.

(1) Mobility

When we consider the mobility of hosts, a pair of a mobile server and a correspondent client as well as a pair of a mobile client and a correspondent server should be taken into consideration. To accommodate these, MSOCKS requires modification to the proxy and the library in the correspondent client. On the other hand, our MobileSocket and TCP-R are able to accommodate these with Mobile IP.

When both the server and the client are mobile hosts, the simultaneous relocation of the either hosts is possible. Mobile-IP, TCP-R with Mobile-IP, and MobileSocket with Mobile-IP can handle this situation, while MSOCK with Mobile-IP requires the modification in the proxy.

(2) Virtual Circuit Continuity

On Virtual circuit continuity, MobileSocket and TCP-R provide the TCP connection continuity without the limitation of the TCP timers and MobileSocket also allows the applications both implicit and explicit redirection. Virtual circuit continuity of MSOCKS and Mobile-IP is limited to the TCP timers.

(3) Implementation

One of the design features of MobileSocket is the simplified and minimized user-level implementation. MobileSocket realizes its functionalities with only libraries in the mobile and the correspondent hosts, while others need the modifications to the existing protocol and the additional software components such as the proxy or the agents.

Although MSOCKS and MobileSocket have two similarities, (1) user-level implementation with the layer above TCP protocol, (2) user-level sliding window mechanism, they have different implementation approaches, the external proxy server and the library at the correspondent host. As the background of this difference, there are the granularity differences of the disconnection period and the relocation scale between them. MSOCKS focuses mainly on the

Table 5 Functional comparison.

Name	(1) Mobility			(2) Virtual Circuit Continuity	
	Layer	Mobile Server Situation	Simultaneous Relocation		Redirection Scheme
Mobile-IP	IP	Yes	Yes	Limited(*3)	N/A
TCP-R	TCP	Limited(*1)	Limited(*2)	Yes	Implicit
MSOCKS	Socket	No	No	Limited(*3)	Implicit
MobileSocket	Socket Upper	Limited(*1)	Limited(*2)	Yes	Implicit / Explicit

Table 6 Functional comparison (Cont'd).

Name	(3) Implementation			(4) Application		
	MH	Additional Software	CH	Application Modification	Adaptation Interface	for Mobile Application
Mobile-IP	M (IP), A (daemon)	HA,FA	unnecessary	No	N/A	N/A
TCP-R	M (TCP)	unnecessary	M (TCP)	No	No	N/A
MSOCKS	A (library)	Proxy (M (TCP))	unnecessary	No	N/A	N/A
MobileSocket	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.

situation of (1) the disconnection shorter than the TCP retransmission timeout and (2) the rather local relocation such as the roaming at the wireless LAN. In such a situation, the proxy approach which does not require the modification in the correspondent host is effective and the affection of the triangle routing may not be critical. MobileSocket, in contrast, focuses mainly on the situation of (1) longer disconnection than the TCP retransmission timeout and (2) relocation in the wide area network such as the ones between the campus and office. In this situation, proxy approach causes the wide area triangle routing problem, and the MSOCKS cannot be used for the disconnection longer than the retransmission timeout. MobileSocket with the peer-to-peer library approach does not cause the routing problem, and can keep the connection longer than MSOCKS.

(4) Application

MobileSocket provides its functionalities without any modification to applications as same as the others. Also, MobileSocket has adaptation interfaces for applications. This function enables appropriate behavior of application towards the dynamic change of the computing environment. Our Java library implementation allows Java mobile applications or the agents with the MobileSocket connections to migrate to another host using the Java Object Serialization, even with the active MobileSocket connection to the remote.

8. Conclusion and Future Work

In this paper, we have presented MobileSocket, a user level library-based solution of the communication continuity support to the applications. Session layer approach and the user-level library installation in the mobile and the correspondent hosts simplify and minimize the implementation. The combination of DSS and ALW achieves the byte stream consistency for the TCP Socket connection. Java event-based adaptation interfaces of MobileSocket realize the application level adaptation toward 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 plan two future works. The first one 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 second one 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.

Acknowledgments The authors would like to thank Keio Media Space Family (KMSF) group, R3 Project group in Keio University, Dr. Antony Rowstron, and Open Media Research Group, Department of Engineering, University of Cambridge for valuable discussions and comments.

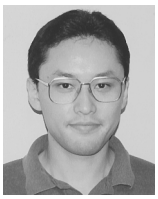
References

1) Bharat, K. and Cardelli, L.: Migratory Applications, *ACM Symposium on User Interfaces Software and Technology* (1995).
2) Richardson, T., Stafford-Fraser, Q., Wood, K. and Hopper, A.: Virtual Network Computing, *IEEE Internet Computing*, Vol.2, No.1, pp.33–

38 (1998).
3) Perkins, C.: IP mobility support (1996). RFC 2002, Internet Request For Comments.
4) Maltz, D. and Bhagwat, P.: MSOCKS: An Architecture for Transport Layer Mobility, *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pp.1037–1045 (1998).
5) Funato, D., Yasuda, K. and Tokuda, H.: TCP-R: TCP Mobility Support for Continuous Operation, *Proceedings of IEEE International Conference on Network Protocols 97*, pp.229–236 (1997).
6) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison Wesley, Reading, MA (1996).
7) Sun Microsystems inc.: Java Developpers Kit (JDK) Version 1.1.6 (1997). http://www.javasoft.com/.
8) Wright, G.R. and Stevens, W.R.: *TCP/IP Illustrated*, Volume 2, Addison Wesley, Reading, MA (1995).
9) Branden, R.: Requirements for Internet Hosts – Communication Layers, RFC 1122, Internet Request For Comments (1989).
10) Postel, J.: Internet Protocol, RFC 791, Internet Request For Comments (1981).
11) Sun Microsystems Inc.: Object Serialization Specification (1996).
12) Hosokawa, T.: PAO: FreeBSD Mobile Computing Package. http://www.jp.freebsd.org/PAO/.

(Received April 30, 1999)
(Accepted October 7, 1999)

Tadashi Okoshi received his B.S. degree in Environmental Information from Keio University in 1998. He is a master's course student at graduate school of Media and Governance, Keio University. He is currently studying adaptive middleware in mobile computing environments. He is a member of the ACM and the IEEE Computer Society.



Masahiro Mochizuki received his B.A. degree in Policy Management from Keio University in 1994. He received his M.A. degree in Media and Governance from Keio University in 1996. He is a Ph.D. Candidate at graduate school of Media and Governance, Keio University. He is currently studying mobile and adaptive middleware and applications. He is a member of the ACM and the JSSST.

Yoshito Tobe is a research staff at Keio Research Institute at SFC, where he is currently studying QoS-aware protocol processing implementation. He is a member of the IEEE Communication Society, the ACM, and the IEICE.

Hideyuki Tokuda received his B.S. and M.S. degrees in electrical engineering from Keio University in 1975 and 1977, respectively; a Ph.D. degree in computer science from the University of Waterloo in 1983. He joined the School of Computer Science at Carnegie Mellon University in 1983, and is an Adjunct Associate Professor from 1994. He joined the Faculty of Environmental Information at Keio University in 1990, and is a professor since 1996. His current interests include distributed operating system and computer networks. He is a member of the ACM, the IEEE, the IEICE, and the JSSST.