A Reliable and Secure Connection Migration Mechanism for Mobile Agents

Xiliang Zhong, Cheng-Zhong Xu, and Haiying Shen
Department of Electrical & Computer Engg.
Wayne State University, Detroit, Michigan 48202
{xlzhong, czxu, shy}@wayne.edu

Abstract

Connection migration in mobile systems is to support continuous and transparent communication operations between mobile agents. This paper presents a reliable connection migration mechanism that provides exactly-once delivery for all transmitted data during agent migration. It integrates with an agent-based access control mechanism that controls the access to network ports. To avoid frequent agent authentication and permission checking due to agent migration, a secret session key is associated with each connection. We present the design and implementation of the mechanism, named NapletSocket in Naplet mobile agent system. It is a pure middleware implementation, requiring no modification of Java virtual machines. Evaluation results show that the NapletSocket system incurs a moderate overhead in connection setup, mainly due to security checking. Once a secure connection is established, only a marginal cost is needed to pay for reliable communication during agent migration.

1. Introduction

In agent-oriented programming, agents often communicate with each other via mailbox-like *asynchronous persistent* communication mechanisms due to the requirement for agent autonomy [2]. That is, an agent can send messages to others no matter its communication parties exist or not. The messages in transmission are often forwarded in support of agent migration.

Asynchronous persistent communication plays a key role in many distributed applications and is widely supported by existing mobile agent systems; see [13] for a recent comprehensive review of location independent communication protocols between mobile agents. However, it is not appropriate or sufficient for certain applications that require agents to cooperate closely. For example, in the use of mobile agents for parallel computing [16], cooperative agents need to be synchronized frequently during their lifetime. A *synchronous transient* communication mechanism would keep the agents work more closely and efficiently. Socket over TCP is an example that ensures instantaneous communication in distributed applications. Since agents tend to move from one server to another for various reasons, it is desirable that an established socket connection would migrate with the agent continuously and transparently.

The traditional TCP protocol has no support for mobility because it has been designed with the assumption that the communication peers are stationary. There are recent studies on mobile TCP/IP in both network and transport layers to support the mobility of physical devices in the arena of mobile computing. We refer to this type of mobility as physical mobility, in contrast to logical mobility of codes. Representatives of the protocols include Mobile IP [7] in network layer, MSOCKS [8],TCP-R [6], M-TCP [12] and Migrate [11] in transport layer. Although these protocols provide feasible ways to link moving devices to network, they have no control over the logical mobility. Mobile agents may also be run on wired networks that have no support for mobile TCP/IP.

Mobile agent systems are usually organized as a middleware. Agent connection migration requires support of session-layer implementations in the middleware. In the past, a few session-layer connection migration mechanisms were proposed. Examples include Persistent Connection [18], Mobile TCP [10], and MobileSocket [9]. However, none of them were targeted at agent mobility. Agent related connection migration involves two unique reliability and security problems. Since both the end points of a connection would move around, a reliable connection migration needs to do more for exactly-once delivery for all transmitted data. Security is a major concern in agent-oriented programming. Socket is one of the critical resources and its access must be fully controlled by agent servers. Agent-oriented access control must be enforced during the setup of connections. Connection migration is vulnerable to eavesdropper attacks. Additional security measures are needed to protect transactions on an established connection from any malicious attacks due to migration.

In this paper, we present the design and implementation of an integrated mechanism that deals with security and reliability in connection migration. It provides an agent-oriented socket programming interface for location-independent socket communication. The interface is implemented by a socket controller that guarantees exactly-once

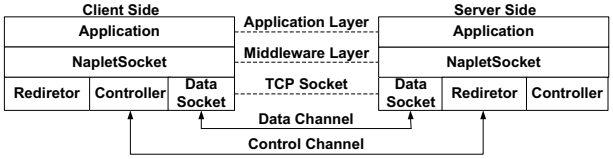

Figure 1: NapletSocket Architecture.

delivery of data, including data in transmission when the communicating agents are moving. To assure secure connection migration, each connection migration is associated with a secret session key created during connection setup.

We prototyped the mechanism as a NapletSocket component in Naplet mobile agent system. Naplet [14] is a featured mobile agent system we developed in house for educational purposes. It supports a mailbox-based PostOffice mechanism with asynchronous persistent communication. NapletSocket provides a complementary mechanism for synchronous transient communication.

The remainder of the paper is organized as follows. Section 2 gives an overview of the design of NapletSocket. Reliability and security concerns and measures are discussed in Section 3. Section 4 presents performance evaluation results. Related works are summarized in Section 5. Section 6 concludes the paper with remarks on future work.

2. NapletSocket Design

2.1. NapletSocket Architecture

A basic requirement of connection migration is location-independent. The NapletSocket connection migration mechanism provides an interface similar to Java Socket. It comprises of two classes *NapletSocket(agent-id)* and *NapletServerSocket(agent-id)*. They resemble Java Socket and ServerSocket in semantics, except that the NapletSocket connection is agent oriented. It is known that Java Socket/ServerSocket establish a connection between a pair of endpoints in the form of (Host IP, Port). For security reasons, an agent is not allowed to specify a port number for its pending connection. Instead, it is the underlying support system that allocates ports to the connection based on resource availability and access permissions. Naplet system contains an agent location service that maps an agent ID to its physical location. This ensures location transparent communication between agents.

For connection migration, NapletSocket system provides two new methods *suspend()* and *resume()*. They can be either called by agents for explicit control over connection migration, or by Naplet navigator for transparent migration.

Fig. 1 shows the NapletSocket architecture. It comprises of three components: data socket, controller and redirector. The component of data socket is the actual channel for data transfer. It contains a pair of send/recieve buffers to keep undelivered data. The controller is used for management of connections and operations that need access right to socket resources. The redirector is used to redirect socket connection from a remote agent to a local resident agent. Both controller and redirector can be shared by all NapletSockets so that only one pair is necessary.

Table 1: States in NapletSocket transitions.

State	Description
CLOSED	Not connected
LISTEN	Ready to accept connections
CONNECT_SENT	Sent a CONNECT request
CONNECT_ACKED	Confirmed a CONNECT request
ESTABLISHED	Normal state for data transfer
SUS_SENT	Sent a SUSPEND request
SUS_ACKED	Confirmed a SUSPEND request
SUSPENDED	The connection is suspended
RES_SENT	Sent a RESUME request
RES_ACKED	Confirmed a RESUME request
CLOSE_SENT	Sent a CLOSE request
CLOSE_ACKED	Confirmed a CLOSE request

To open a connection, the controller of the client agent sends a request to the counterpart at the server. After the request is acknowledged, the client connects to the redirector at the server side and the connection is then handed to the desired agent. After a connection is established, the two agents communicate with each other through accessing the socket, no matter where their communication parties are located. Under the hood are a sequence of operations by the NapletSocket library. The underlying data socket is first closed, when the NapletSocket takes a suspend action before agent migration. After the agent lands on the destination, the NapletSocket system resumes the connection by connecting to the server-side redirector. The data sockets of both client and server are then updated and new input/output streams are re-created atop of the socket.

2.2. State Transitions

The design of NapletSocket can be described as a finite state machine, extended from the TCP protocol. It contains 12 states, as listed in Table 1. The states in bold are newly added to the standard TCP state transitions diagram. A NapletSocket connection is in one of these states. The NapletSocket system takes an action when a certain event occurs, according to the current state of the connection. There are two types of events: calls from local agents and messages from remote agents. Actions include sending messages to remote agents and calling local functions.

Fig. 2 shows the state transitions of NapletSocket. The solid lines show the transitions of clients connecting to servers and the dotted lines are for the servers. Details of the open/suspend/resume/close transactions are as follows.

Open a connection Both client and server are initially at the CLOSED state. When an agent does an active open, a CONNECT request is sent to the server and the state of the connection changes to CONNECT_SENT. If the request is accepted, the client side NapletSocket receives an ACK and a socket ID to identify the connection. Then it sends back its own ID and the state changes to ESTABLISHED.

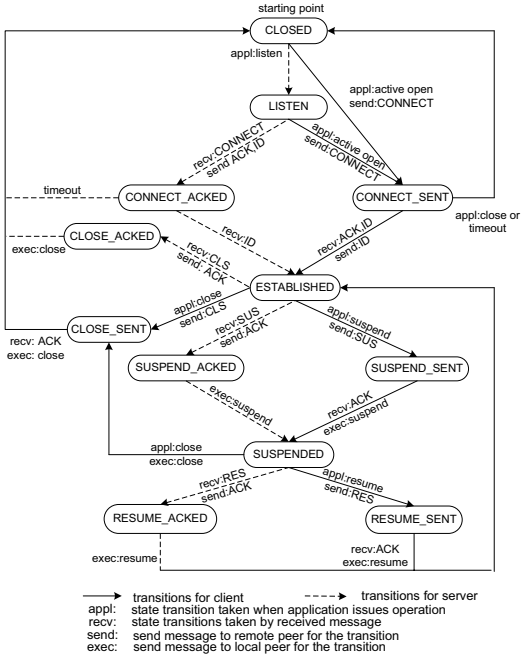

Figure 2: NapletSocket state transitions diagram.

Connection in server side switches to the LISTEN state once an agent does a listen. When a CONNECT request comes from a client, the server acknowledges it by sending back an ACK and a socket ID. It then changes to the CONNECT_ACKED state. After the socket ID of the client side is received, it switches to ESTABLISHED and the NapletSocket connection is established. Now the data can be transferred between the two peers as normal socket connection.

Suspend/Resume a connection After a connection is established, either of the two parts may suspend it. The one who wants to suspend a connection invokes the suspend interface and a SUS is sent to the peer. If the request is acknowledged, an ACK is sent back and triggers the action of closing underlying input/output streams and data socket. The connection state then switches to SUSPENDED.

When the other side of NapletSocket receives the SUS message, it sends back an ACK if it agrees to suspend. Then it closes the input/output streams and the data socket under NapletSocket. After that, the state for this peer also changes to SUSPENDED. Now connections at both peers are suspended. No data can be exchanged in this state.

At the SUSPENDED state, when either of the agents decides to resume the connection, it invokes the resume interface. The resume process first sets up a new connection to the remote redirector and sends a RES message. After an ACK is received, it then resumes the connection and the state switches back to ESTABLISHED. For the remote peer in the SUSPENDED state, once it receives a resume request, it first sends back an ACK. Then the redirection server hands its connection to the desired NapletSocket and new input/output streams are created. After that, both peers change back to the ESTABLISHED state.

Close a connection In either the ESTABLISHED or the SUSPENDED state, if an agent decides to close the current connection, it invokes the close interface and the NapletSocket does an active close by sending a CLS(CLOSE) request to the peer. After the request is acknowledged, local data socket is closed. For the other side of the connection, it closes passively after receiving a CLS request. It first acknowledges the request and then closes the underlying socket and streams. At the time, data socket at both sides is closed and the state changes to CLOSED.

3. Design Issues

3.1. Transparency and Reliability

If there is any connection setup before agent migration, the connection should be migrated transparently. The main approach for connection migration is to use a data socket under NapletSocket. Each time the agent migrates, the underlying data socket is closed before migration and updated to a new data socket after migration. But for a connection between two freely migrated agents, it is possible that migration happens at the same time when there are data being transferred and in this case, the data may fail to reach their destinations. So the presence of mobility causes a problem for reliable communication. Furthermore, two agents may migrate simultaneously which makes it more difficult to achieve reliability .

In order to guarantee that messages can be finally delivered, we added an input buffer to each input stream and wrapped them together as a NapletInputStream. To suspend a connection, the operation retrieves all currently undelivered data into the buffer before it closes the socket. The data in the NapletInputStream migrate with the agent. When migration finishes and the connection is resumed at the remote server, a read operation first reads data from the input buffer of its NapletInputStream. It doesn't read data from socket stream until all data from the buffer have been retrieved.

In the case both agents of a connection want to move simultaneously, there is a problem since the resume operation from one agent only remembers the previous location of the other agent. A simple solution to this problem is to give priority to one side of the connection and delay the other. For example, we can assign server-side NapletSocket a higher priority than client-side NapletSocket so as to ensure server-side migration occurs before client-side. However, this could lead to a deadlock problem if more than two agents that are connected to each other in a dual role of client and server are moving simultaneously. For example, in a configuration of three agents X, Y, and Z which are clients of Y, Z, and X, respectively, the system gets into a deadlock state if all the agents want to move at the same time. To prevent deadlock in simultaneous migration, we can determine migration priority based on ordered agent IDs. Although the agent ID based priority policy guarantees deadlock-free, it may cause another fairness problem in migration. It deserves further studies at least in theory.

3.2. Security

Security is always a basic and direct concern in mobile agent system. Here we address security issues in two aspects. First, the agent should not be able to cause any security problems to the host it resides, either at the original host or at the host it migrates to. Second, the connection itself should be secure from possible attacks like eavesdropping. More specifically, a connection can only be suspended/resumed by the one who initially establishes it.

Regarding the first issue, it is safe enough if we deny any requests to create a Socket or ServerSocket from an agent. Permissions are only granted to requests from the NapletSocket system. So the only way for an agent to use socket resources is through the service provided by the mobile agent system. Now the problem becomes whether we can deny permission if a request is from an agent and grant it if it is from the system.

This problem can be solved by user-based access control introduced in the latest JDK security mechanism. It allows permissions to be granted according to who is executing the piece of code (subject), rather than where the code comes from (codebase). A subject represents the source of a request such as a mobile agent or NapletSocket controller. By denying access requests from the subject of agents for socket resources and granting them to local users like administrators, we achieve our security goal in a simple and clear manner. In mobile agent applications, an agent subject has no permissions to access local socket resources by default. When it needs access to a socket resource, it submits a request to a proxy service in NapletSocket controller. The proxy authenticates the agent and checks access permissions. After the security check passes, a NapletSocket or NapletServerSocket will be created by the proxy and returned to the agent. More details about agent-oriented access control in Naplet system can be found in [15].

Regarding the second issue, connection migration can be realized by the use of a socket ID. However, a plain socket ID couldn't prevent a third party from intercepting the information and exercising eavesdropping attacks. To this end, we applied Diffie-Hellman key exchange protocol [4] to establish a secret session key between the pair of communicating agents at the setup stage of their connection. Any subsequent requests for suspend, resume, and close operations on the connection must be accompanied with the secret key. Such requests will be denied unless their keys are verified by remote peers. Since the key generated by Diffie-Hellman protocol is hard to break, NapletSocket connections are protected from eavesdropping attacks.

3.3. Socket Handoff

As we know in NapletSocket, when a client connects to a server, it uses an agent ID to specify which server to connect to. So we need to find both host name and port number from the ID. For host name, we can keep records of traces of agents and locate the host according to its ID. For port number, we need to send a query message to the host where the

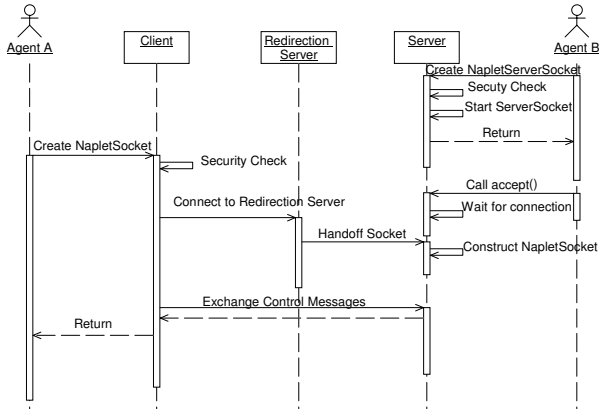

Figure 3: Socket handoff in connection setup.

NapletServerSocket is. The server has to maintain a table indicating which agent uses which port and return the port to client. Then the client can start a connection. In fact, it is not necessary for the client to know which port to connect to. By connecting to the redirector at the server and indicating which agent it wants to connect to. The server looks up which NapletServerSocket this request is for and redirects the current socket to it. Then the NapletServerSocket is waken up from waiting and constructs a NapletSocket according to the data socket it receives. We call this socket handoff and it can save at least one round trip time in querying host name and port number and there is no need for the server to maintain a table mapping ports to agents. Fig. 3 shows the sequence diagram for socket handoff in connection setup.

Similar mechanism also applies when resuming a connection. In this case, the client connects to redirection server at the other side and sends a request to resume . The server then hands the socket connection to the suspended NapletSocket and wakes it up. Finally the notified NapletSocket updates its underlying data socket.

3.4. Control Channel

It is necessary to exchange control messages during state transitions of a NapletSocket connection. From a performance perspective, we used a separate channel for control messages and chose UDP as the transport layer protocol. For the omission failures and ordering problems caused by UDP, we adopted a retransmission mechanism to provide reliable delivery on top of UDP. The basic idea is to use retransmission in case of failure. After sending a control message, the sender starts a retransmission timer and waits for an ACK from the receiver. If an ACK is received before timeout, the timer is cancelled. If no ACK is received after timeout, the message is retransmitted and a new timer for the message is set. Sequenced numbers are used to relate a reply to the corresponding request.

4. Evaluation

In this section, we present the performance of NapletSocket and compare it with Java Socket. The experiments

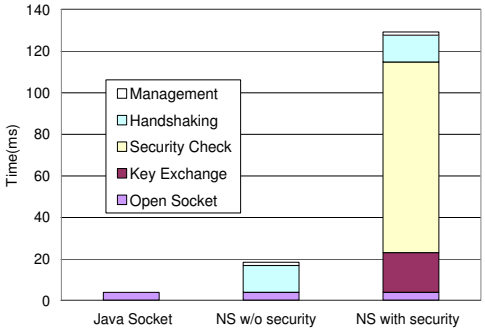

Figure 4: Breakdowns of the latency to open a connection.

Table 2: Latency to open/close a connection

Connection Type	Open (ms)	Close (ms)
Java Socket	3.7	0.6
NapletSocket w/o security	18.2	12.5
NapletSocket with security	134.4	12.6

were conducted in a group of Sun Blade workstations connected by a 100M ethernet. Two primary performance metrics were used in evaluation: latency and throughput.

4.1. Open and Close a Connection

The first experiment was on the latency to open and close a NapletSocket connection. The connection has an option to enable/disable security checking. We performed open and close operations for 100 times and calculated the average time for each operation. Table 2 shows the results. For comparison, the latencies to open and close a Java Socket connection are also included. It is known that opening a connection involves a number of operations: authentication, authorization, secret key exchange, handshaking and socket establishment. Breakdown of the latencies for opening a connection is shown in Fig. 4.

From Table 2, we can see that opening a connection with migration and security support costs almost 40 times as much as that of Java Socket. Fig. 4 shows that more than 80% of the time was spent on key establishment, authentication and authorization. The latency would reduce to 18.2ms without security support. In both cases, NapletSocket involves some latencies. But with the reliable communication mechanism provided for mobile agents, open a connection is a one-time operation and the connection remains alive once established, which means that cost of opening a connection can be amortized over agent migration.

4.2. Suspend and Resume a Connection

We measured the latencies for suspend and resume operations and recorded 27.8ms and 16.9ms respectively. The delays mainly come from the exchange of control messages (handshaking), which makes up about 50% for suspending and 70% for resuming. Suspending a connection also requires to check if there are any undeliver data in the input stream. Resuming a connection also needs to set up a data socket and create I/O streams.

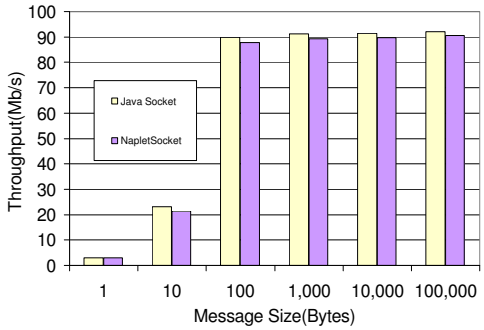

Figure 5: Throughput of NapletSocket vs. Java Socket.

The benefit of provisioning a reliable connection can be seen by comparing the time required for re-opening a connection with that of suspend/resume. If we close a NapletSocket before migration and reopen a new one after that, the total latency involved is about 147ms. However, if we use suspend and resume instead, the latency is less than one third of the cost for close and reopen operations. When an agent travels around the network, the total time saved by using suspending and resuming is considerable.

In summary, both of these operations bring latencies, but with the continuous connection it offers, the latency is reasonable and acceptable.

4.3. Throughput

In the next experiment, we measured the throughput of NapletSocket. The benchmark chosen for this experiment was TTCP [3], a well-known program for measurement of Java Socket throughput. We measured throughput by changing the message size (product of package number and package size). Fig. 5 shows the results. The throughput of Java Socket is included for comparison.

From the figure, we can see the throughput of NapletSocket degrades slightly (less than 5%), in comparison with Java Socket. This degradation is mainly due to synchronization required in accessing I/O streams. With the increase of message size, the performance gap between NapletSocket and Java Socket becomes almost negligible.

5. Related Work

There are a number of techniques proposed for connection migration in different contexts.

The conventional technique is from network-layer, enabling the same IP address to be used even when users change to another network attachment point. An example of this technique is Mobile IP [7] which works on a concept of home agent associated with the mobile host. Every package destined to a mobile host by its home address is intercepted by its home agent and forwarded to it.

Network layer implementations are not appropriate or sufficient for some applications [1]. There are many studies focusing on transport layer support for mobility. One of the representative work is due to Snoeren [11, 1]. It uses an end-to-end mechanism to handle host mobility, by extending the TCP protocol with a TCP migrate option. The se-

mantics of TCP remains unchanged. There are other similar work, such as TCP-R [6], M-TCP [12]. Although most of them work well, they require to change OS kernels. This hinders the protocols from wide deployment.

Connection migration in network and transport layers is mainly for the support of physical mobility. Connection migration due to code mobility requires support at session layer. Zhang and Dao introduced a persistent connections model [18]. They described connection end points in terms of location-independent IDs, which are stored in a centralized host. When an end point changes its attachment point, it notifies and the host and then the host notifies all others in the system. By using a centralized host, their approach may suffer from poor performance.

Qu, *et al.* later proposed a similar mechanism to preserve upper layer unbroken connections [10]. They used OS-specific kernel interface to access the kernel buffer for data not yet delivered.

Okoshi, *et al.* presented a library solution called MobileSocket on top of Java Socket [9]. They used dynamic socket switch to update connection of MobileSocket and application layer window to keep all outgoing data at user level buffers so that any data in the buffers can be recovered from broken connections.

Similar mechanisms were used for robust TCP connections due to mask server crash or communication failures. Robust TCP connections [5] addresses reliable communication problem in the area of fault tolerant distributed computing. The authors used similar approach when designing reliable communication and implemented it as a reliable socket on top of Java Socket. Reliable Sockets (Rocks) [17] allows TCP connections to support changes in attachment points. It emphasizes reliability over mobility, viewing mobility as just another cause of network connection failure. It has support for automatic failure detection and a protocol for safely inter-operates with end points that do not support Rocks.

6. Conclusions and Future Work

Mailbox-based asynchronous persistent communication mechanisms in mobile agent systems are not sufficient for certain distributed applications like parallel computing. Synchronous transient communication provides complementary services that make cooperative agents work more closely and efficiently. Connection migration is necessary in response to agent migration. This paper presents a reliable connection migration mechanism that guarantees exactly-once message delivery. The mechanism uses agent-oriented access control and secret session keys to deal with security concerns arising in connection migration. A prototype of the mechanism, NapletSocket, has been developed in Naplet mobile agent system, which shows a reasonable overhead in support of connection migration.

We note that current implementation is limited to agent migration at one endpoint of a connection. As part of ongoing work, we are developing solutions in support of concurrent connection migration where agents at the both endpoints migrate at the same time.

Acknowledgement This research was supported in part by NSF grants CCR-9988266 and ACI-0203592.

References

[1] M. Alexander and C. Snoeren. *A Session-Based Architecture for Internet Mobility*. PhD thesis, MIT, February 2003.

[2] J. Cao, X. Feng, J. Lu, and S. Das. Mailbox-based schedule for mobile agent communication. *IEEE Computer*, 35(9):54–60, 9 2002.

[3] Chesapeake Computer Consultants. Tools - Test TCP (TTCP). http://www.ccci.com/tools/ttcp/.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[5] R. Ekwall, P. Urb'an, and A. Schiper. Robust TCP connections for fault tolerant computing. In *Proc. of Int'l Conf. on Parallel and Distributed Systems*, 2002.

[6] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. In *Proc. of IEEE Int'l Conf. on Network Protocols, pages 229-236*, October 1997.

[7] J. Ioannidis, D. Duchamp, and G. Q. Maguire. IP-based protocols for mobile internetworking. In *Proc. of ACM SIGCOMM*, April 2002.

[8] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proc. of IEEE INFOCOM*, pages 1037–1045, 1998.

[9] T. Okoshi, M. Mochizuki, and Y. Tobe. Mobilesocket: Toward continuous operation for java applications. In *Int'l Conf. on Computer Communications and Networks, pages 50-57*, October 1999.

[10] X. Qu, J. X. Yu, and R. P. Brent. A Mobile TCP Socket. In *Proc. of IASTED Int'l Conf. on Software Engineering*, November 1997.

[11] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th Int'l Conf. on Mobile Computing and Networking (MobiCom)*, 2000.

[12] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. In *Proc. of 22nd IEEE Int'l Conf. on Distributed Computing Systems*, July 2002.

[13] P. T. Wojciechowski. Algorithms for location-independent communication between mobile agents. In *Proc. of AISB Symposium on Software Mobility and Adaptive Behaviour*, pages 10–19, March 2001.

[14] C.-Z. Xu. Naplet: A flexible mobile agent framework for network-centric applications. In *Proc. of the Second Workshop on Internet Computing and e-Commerce (In conjunction with IPDPS)*, April 2002.

[15] C.-Z. Xu and S. Fu. Privilege delegation and agent-oriented access control in naplet. In *Proc. of Int'l Workshop on Mobile Distributed Computing (In conjunction with ICDCS)*, pages 493–497, May 2003.

[16] C.-Z. Xu and B. Wims. Mobile agent based push methodology for global parallel computing. *Concurrency: Practice and Experience*, 14(8):705–726, July 2000.

[17] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proc. of 8th Annual ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, September 2002.

[18] Y. Zhang and S. Dao. A persistent connection model for mobile and distributed systems. In *Proc. of Int'l Conf. on Computer Communications and Networks*, September 1995.