A Layer Enabling Transport Independent Connection

Yuichi Ueno
Fuji Xerox Co., Ltd.
430 Sakai, Nakai-machi, Ashigarakami-gun
Kanagawa 259-0157
Japan
Yuichi.Ueno@fujixerox.co.jp

Masayuki Ida
Aoyama Gakuin University
4-4-25 Shibuya, Shibuya
Tokyo 150-8366
Japan
ida@gsim.aoyama.ac.jp

***Abstract* – The TCP/IP technology intrinsically enables symmetric communication model. However, today's communication services adopt asymmetric model. And the model is "hard-coded" to simplify implementation and management. Usual middle-ware technologies provide mechanism to enable transport layer independence. However, these technologies can not handle both transport independence and quality of service simultaneously. We have developed a new layer enabling transport independent connection, as a Java component. The layer enables endpoint migration and pluggable communication channel. We describe design and current implementation of our component. Furthermore, by measuring communication performance of our component, we show our component can draw full performance from an underlying communication channel.**

I. INTRODUCTION

Fundamental functionality of communication service is management of connectivity between endpoints which are associated with application programs respectively. The TCP/IP technology intrinsically enables symmetric communication model. However, today's communication services adopt asymmetric model. Connectivity management between communication services is an issue for which higher layer over transport layer should take on the responsibility. Since each application program tackles the issue individually, role specialization of a server/client is "hard-coded" onto application program and is managed on particular node for simplifying.

Distributed object technology CORBA[5] and Java RMI[7] can make communication service independent from node by means of concealing communication channel with object request broker(ORB) layer. However application cannot be aware of latency or bandwidth of communication channel. Data communication and multimedia communication require different quality of service(QoS) each other. Network transparency rather complicates construction of reliable network services because it makes QoS control harder[9].

MobileSocket[6] and Medlar[4] offer continuous connection for migrating node across subnets. The former is realized as substitution of `java.net.Socket` class which manages implicit/explicit reconnection of an underlying TCP-socket connection. The latter provides dedicated proxy which conceals communication channel to server peer and manages QoS. They support migration of client node itself across subnets. They don't support migration of connection across nodes.

We introduce a new layer enabling transport independent connection, as a Java[2] component. The layer enables endpoint migration (shown at Fig.1) and pluggable communication channel. In "digital convergence" era, very diverse services should be provided through computer devices which exist ubiquitously. Our component is a base technology for node independent communication services which will be demanded in that era.

We introduce design and initial implementation of our component in this paper. Furthermore, we measure performance of the initial implementation and evaluate reasonability of the design.

II. DESIGN

A. *Elements and Roles*

Fig.2 illustrates design overview of our component.

We introduce a layer providing transport layer independent connection. We design our component as two-layered structure consists of *ForwardableSocket* object and *Connector* object.

ForwardableSocket object provides application program interface (API). And, it conceals detail of procedures of transport layer level connection. Socket reconnection is processed internally responding to peer endpoint migration. And buffer synchronization is performed if needed.

Connector object is an adapter which encapsulates transport layer level socket into uniform interface for *ForwardableSocket*.

ContractManager objects which manage connectivity based on agreements between services or nodes. Our component provides generic interface of *ContractManager*. Each application must implement dedicated *ContractManager*. Strictly speaking, there are two types of *ContractManager* objects: *ServiceContractManager* and *TerminalContractManager*. The former handles *ServiceContract* object and the latter handles *TerminalContract* object respectively.

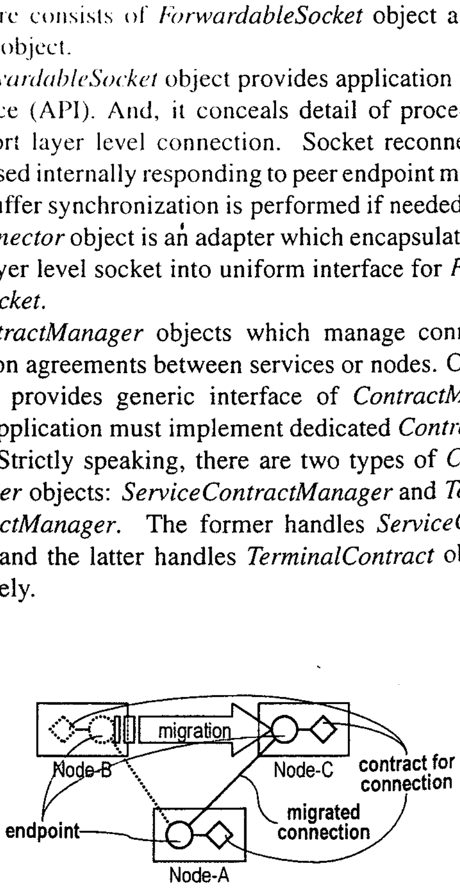


Fig. 1 . Endpoint migration

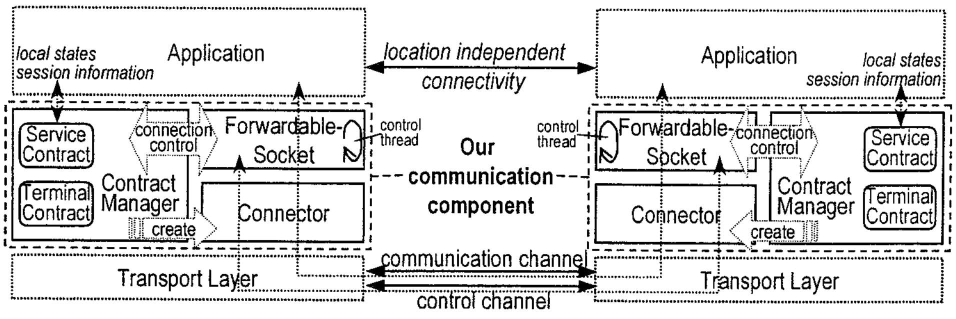


Fig. 2 . A layer enabling transport independent connection

Each *Contract* object is placeholder for runtime information against the agreements. Endpoint migration is accompanied by *ServiceContract* object migration. Application program can also use *ServiceContract* to suspend local states before migration and to resume local states after migration.

B. Application Program Interface

ForwardableSocket acts as a communication endpoint for applications. *ForwardableSocket* provides API for connection oriented connectivity control and datagram-passing style communication operation. Implementation of *Connector* assures reliability and sequential delivery of datagram.

Fig.3 illustrates state diagram of *ForwardableSocket* as client socket. Like TCP-socket, *connect* operation attempts to make a connection to another *ForwardableSocket*. However, unlike TCP-socket, this attempt is performed under the control of *ContractManager* which is implemented with application. This design enables application to control connection by own policy.

ForwardableSocket is also server socket. Fig.4 illustrates state diagram of *ForwardableSocket* as server socket. Like TCP-socket, *accept* method returns a new socket corresponding to incoming connection.

In the same way as the above discussion, connectivity control is managed by *ContractManager*.

We provide three types operations for *ForwardableSocket*: *forward*, *suspend*, *resume*. These operations make connection node independent.

ForwardableSocket can be forwarded to other ones. If the forwarding succeeds, the *ForwardableSocket* itself is closed on forwarder node side, and a new *ForwardableSocket* is created on forwardee node side. Since *ServiceContract* and internal buffer of *Connector* is migrated during this operation, *ForwardableSocket* reconstructs connection properly. Consequently, the newly created *ForwardableSocket* and the peer *ForwardableSocket* against it can keep operating continuously. The application on the peer node against the forwardee node don't have to be aware of the detail of this reconnection procedure in transport layer level, because our components conceals it.

The *suspend* and the *resume* operations perform suspending/resuming connection between services respectively. *Suspend* operation attempts to disconnect *Connector*, but keeps *ServiceContract*. *Resume* operation attempts to reconnect *Connector* based on the kept *ServiceContract*.

Fig.5 illustrates example code of connection migration. Each service has daemon thread and service thread. Daemon thread accepts newly connection or forwarded connection. Then the daemon thread starts service on the connection.

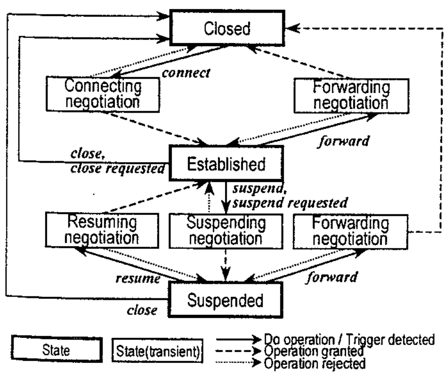


Fig. 3 . State diagram as client socket

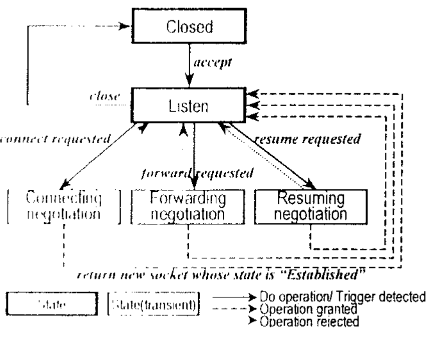
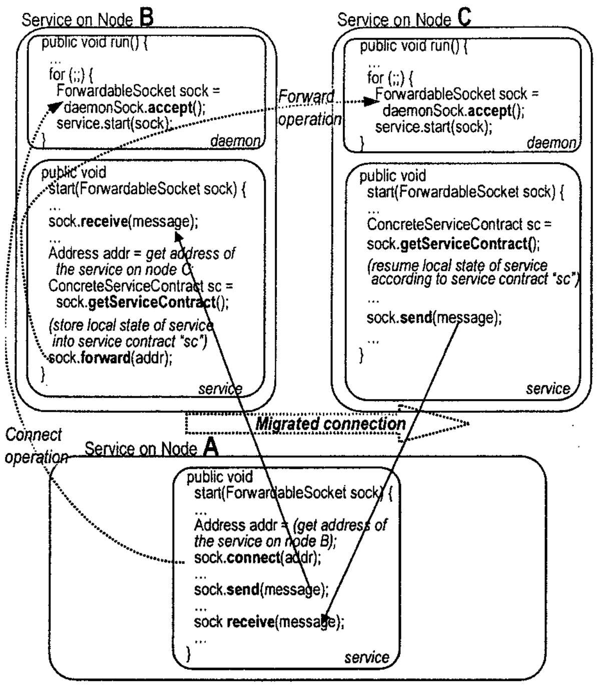


Fig. 4 . State diagram as server socket



```
public void run() {
  ...
  for (;;) {
    ForwardableSocket sock =
      daemonSock.accept();
    service.start(sock);
}

public void
  start(ForwardableSocket sock) {
  ...
  sock.receive(message);
  ...
  Address addr = get address of
    the service on node C
  ConcreteServiceContract sc =
    sock.getServiceContract();
  (store local state of service
   into service contract "sc")
  sock.forward(addr);
}
```

```
public void run() {
  ...
  for (;;) {
    ForwardableSocket sock =
      daemonSock.accept();
    service.start(sock);
}

public void
  start(ForwardableSocket sock) {
  ...
  ConcreteServiceContract sc =
  sock.getServiceContract();
  (resume local state of service
   according to service contract "sc")
  ...
  sock.send(message);
  ...
}
```

```
public void
  start(ForwardableSocket sock) {
  ...
  Address addr = (get address of
  the service on node B);
  sock.connect(addr);
  ...
  sock.send(message);
  ...
  sock receive(message);
  ...
}
```

Fig. 5 . Migrating service on B to C

In this example, connection is migrated from service on node B to service on node C. Service on node A can handle message continuously through same socket.

C. *Requirements for socket encapsulation*

Connector must provides a communication channel which enables datagram-passing style communication and primitive connection/disconnection controlling operations. *ForwardableSocket* directly delegates *send* operation and *receive* operation to *Connector*. Any of concrete socket operation is allowed for sending/receiving datagram. *ForwardableSocket* is not aware of reliability or sequential delivery of datagram. Application program must select proper *Connector* corresponding to requirements of QoS. Our design allows development of various kinds of *Connector*.

Furthermore, *Connector* must provide a control channel which has communication semantics as sequenced, reliable, two-way, connection-based byte streams. This is used internally by the *ForwardableSocket* to control communication channel. Any concrete implementation of control channel is allowed if it satisfy the above communication semantics.

D. *Connection Control with ContractManager*

As we described in section II.B., *ContractManager* is responsible for management of control of service level connectivity between communication services. *ContractManager* must manage life-cycle of *ServiceContract* and *TerminalContract* depending operations on *ForwardableSocket*.

Our component provides generic interfaces for *ContractManager* which responsible for *ServiceContract* and *TerminalContract* respectively. The former is *ServiceContractManager* interface. The latter is *TerminalContractManager* interface. Each application must implement concrete ones.

Fig.6 describes *ServiceContractManager* interface. This interface defines basic property of communication service. *getServiceAddress* represents listening an address of daemon thread's socket on service. *createConnector* is a factory method to create concrete *Connector* instance suitable for QoS demand of service. Connection control must be programmed in *createServiceContract* and *updateServiceContract* methods. Based on both sides *ServiceContract*, communication service can decide to accept or deny incoming connection.

Fig.7 illustrates the sequence of connect-operation. On the figure, Host1 attempts to connect to Host2. This operation is performed through a control channel of *Connector*. Both sockets exchange contract each other, then they decide to accept or deny incoming connection by checking the contract.

```
public interface ServiceContractManager {
  public Address getServiceAddress();
  public Connector createConnector();
  public ServiceContract
    createServiceContract(String action,
                          TerminalContract terminalContract)
    throws ServiceContractException;
  public void
    updateServiceContract(ServiceContract localContract,
                          String action,
                          TerminalContract terminalContract,
                          ServiceContract remoteContract)
    throws ServiceContractException;
  public byte [ ] marshalContract(ServiceContract contract);
  public ServiceContract unmarshalContract(byte [ ] contract);
}
```

Fig. 6 . Interface for controlling service level connection

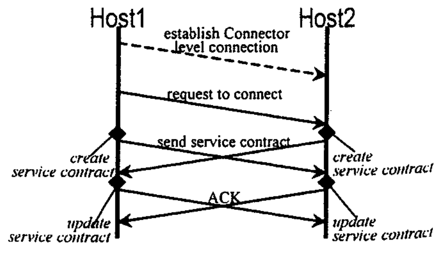


Fig. 7 . Connect operation with exchanging contracts

III. IMPLEMENTATION FOR JAVA 2 PLATFORM, STANDARD EDITION

A. *UDP-socket Adapter*

We have developed a UDP-socket adapter class called *SimpleDatagramConnector* (SDC). Fig.8 illustrates internal structure of SDC. SDC uses `java.net.DatagramSocket` class (UDP-socket support in Java) as a communication channel. Thus, this connector provides unreliable communication channel.

On compile time, application uses uniform interface defined for *ForwardableSocket* API. On the other hand, on run time, application can handle directly `java.net.DatagramSocket` by means of object oriented technique. As we mentioned in the section II.C., *ForwardableSocket* just delegates *send/receive* operation to *Connector* object's ones. This *Connector* class, furthermore, just delegates them to `java.net.DatagramSocket`. Our component rarely imposes overhead upon application if you use this *Connector*.

B. *TCP-socket Adapter*

We have also developed a TCP-socket adapter class called *SimpleStreamConnector* (SSC). Fig.9 illustrates internal structure of SSC. This connector provides reliable communication channel.

To assure this even if *forward* operation is performed, *SimpleStreamConnector* handles a synchronized internal buffer and performs sending the buffered data after connection migration. On this design, there is overhead to copy and synchronize internal buffer for application if you use this *Connector*. Big issue for implementation on the control is "I/O blocking". If a running thread enters into "I/O blocking" status, *ForwardableSocket* can not control the thread. Synchronized internal buffer control is also a safety mechanism to avoid "I/O blocking".

C. *Adopting Object Oriented Design Using Javadoc Tool*

On the design process, we have used the Javadoc[8] tool. We discussed and refined our design on API documents

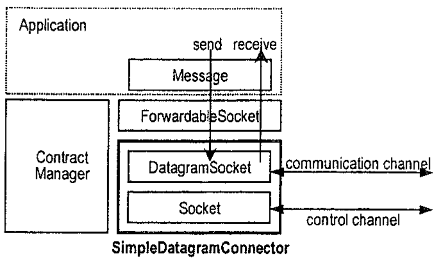


Fig. 8 . UDP-Socket direct manipulation

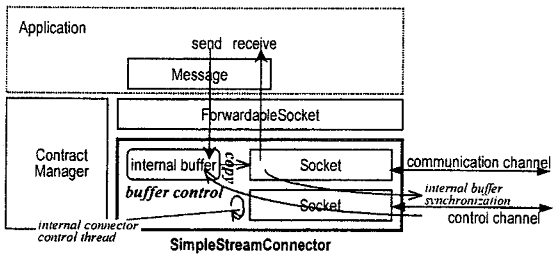


Fig. 9 . TCP-Socket with buffer synchronization

generated by the Javadoc tool. We achieved designing our component beforehand with completed implementation.

D. *Current Implementation*

We have implemented our component on Java 2 Platform, Standard Edition[1] version 1.3.1 which runs on Linux version 2.2.19 with glibc version 2.1.3. We have implemented full functions except *suspend* and *resume* operations.

IV. EVALUATION

A. *Performance with UDP-socket Adapter*

This experiment measures communication performance of *ForwardableSocket* combining the UDP-socket adapter SDC. The result is on TABLE I.

TABLE I
COMMUNICATION PERFORMANCE WITH UDP-SOCKET ADAPTER

packet size [bytes]		1K	2K	4K	8K	16K	32K
Elapsed time [msec.] (send)	ForwardableSocket + SDC	834	1705	3398	6812	13654	27326
	DatagramSocket	866	1733	3418	6841	13681	27341
	Efficiency(%)	100+	100+	100+	100	100	100
Elapsed time [msec.] (receive)	ForwardableSocket + SDC	878	1740	3427	6847	13689	27336
	DatagramSocket	874	1742	3428	6852	13694	27392
	Efficiency(%)	100	100	100	100	100	100

We measured the total elapsed time in sending/receiving a fixed length byte array to/from peer in 1000 times after connection established. We used isolated 10Mbps Ethernet for this experiment. Moreover, as a reference experiment, we measured the total elapsed time of same data-passing operations on normal socket. Then, we compared both elapsed time. We compared SDC class to `java.net.DatagramSocket` class. On TABLE I, each elapsed time is the mean of 10 times measurement.

TABLE I shows that overhead hardly appears about this UDP-socket adapter as we intend in the design phase. This results show that design of our component can draw full performance from an underlying communication mechanism provided by Java runtime environment. Thus, we will be able to do porting and run UDP-communication based Java programs very effectively on our component.

B. *Performance with TCP-socket Adapter*

This experiment measures communication performance of *ForwardableSocket* combining the TCP-socket adapter SSC.

We measured performance with the same condition to the above experiment. On this experiment, reference component is `java.net.Socket` class. The result is on TABLE II.

As we can see in TABLE II, large overhead appears about TCP socket family. SSC handles a dedicated internal flow control mechanism and a buffer management to realize reliable communication through forward operation. The overhead is compensation for this property.

However, our design can handle both `java.net.Socket` class and `java.net.DatagramSocket` class. Since they are primitive communication API to build other communication mechanism, we may easily improve the efficiency of the implementation by handling a dedicated lightweight transmission control protocol.

C. *Connection management performance*

This experiment measures performance of connection management.

We measured each elapsed time of performing *connect&close* or *forward* operation. TABLE III shows the result of this experiment. On this experiment, each elapsed time is the mean of 100 times measurement, size of *ServiceContract* object is zero.

TABLE III
CONNECTION MANAGEMENT PERFORMANCE

Operation†	(0)	(1)	(2)	(3)	(4)
Elapsed time [msec.]	0.737	60.7	143	132	303

† (0)connect: Socket,
(1)connect: ForwardableSocket + SDC,
(2)connect: ForwardableSocket + SSC,
(3)forward: ForwardableSocket + SDC,
(4)forward: ForwardableSocket + SSC

This results shows that *forward* operation requires about double elapsed time compared to *connect* operation. Since each our component handles two temporary connections on forwarding negotiation phase, this result reflects such transitional procedure and it is a necessary cost.

On the other hand, we should indicate an improvement point against performance difference between SSC and SDC. SSC requires over double elapsed time compared to SDC. This result is not caused by the numbers of TCP-sockets *Connector* uses internally, because TCP-socket level connection is very low-cost operation. This result is caused by synchronized buffer initialization with hand-shake. As we discussed in the section IV.B., we may easily improve by handling a dedicated lightweight transmission control protocol.

V. CONCLUSION

We have developed a new layer enabling transport independent connection, as a Java component. The layer enables endpoint migration and pluggable communication channel. Our current implementation can handle both UDP-socket and TCP-socket as an underlying communication channel. Furthermore, we have showed by experiments that our component can draw full performance from an underlying communication mechanism provided by Java runtime environment. Our component will be a base technology for transport independent communication services and will be able to adapt to diverse services in the digital convergence era.

TABLE II
COMMUNICATION PERFORMANCE WITH TCP-SOCKET ADAPTER

packet size [bytes]		1K	2K	4K	8K	16K	32K
Elapsed time [msec.] (send)	ForwardableSocket + SSC	1504	2725	5471	11037	22915	55791
	Socket	1135	2220	4665	9226	18976	37025
	Efficiency(%)	75	81	85	84	83	66
Elapsed time [msec.] (receive)	ForwardableSocket + SSC	1665	2602	4742	8885	17034	50201
	Socket	996	1844	3659	7552	14908	29840
	Efficiency(%)	60	71	77	85	88	59

It is future work for us to design a framework which en-
les dynamical deployment and initialization of applica-
n program code for using services ubiquitously. *Con-
ctManager* can use "policy objects" in runtime environ-
ent to acquire basic authorization information and some
ormation related to services which end-user has con-
cted to. We may extend the current design to become
mpliant to Java Security Architecture[3].

VI. ACKNOWLEDGMENTS

We have created this technology through collaboration
ith members of WISP research group. We gratefully ac-
owledge contributions of Kazuhiro Kazama. His great
ntributions are by no means inferior to be listed as author.
e would like to thank Takao Maekawa, Yasuaki Nakan-
i, Hiroshi Sasaki and Shuichi Fujieda for very creative
scussion. We would also like to thank Koichi Takiguchi,
akoto Hirose and Yoshihiro Masuda for adequate and
oughtful support for our activities.

VII. REFERENCES

] Aberdeen Group, Inc. *Deliverling Real-World Benefits with Client-Side Java Technology*, 2000. URL: "`http://java.sun.com/j2se/1.3/whitepaper1.pdf`".

] Ken Arnold, James Gosling, and David Holmes. *The Java(tm) Language Specification, Third Edition.* Addison-Wesley, 2000.

] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java(TM) Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[4] Minoru Katayama, Koichi Takasugi, Minoru Kubota, and Ichizou Kogiku. A method of achieving service continuity between different networks. *Transactions of the institute of electronics, information and communication engineers*, J84-B(3):452–460, 2001.

[5] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1999. Revision 2.3.

[6] Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hideyuki Tokuda. MobileSocket: Session layer continuous operation support for Java applications. *Transactions of Information Processing Society of Japan*, 41(2):222–234, 2000.

[7] Sun Microsystems, Inc. *Java(TM) Remote Method Invocation Specification*, 1999. URL: "`ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf`".

[8] Sun Microsystems, Inc. *How to Write Doc Comments for the Javadoc(TM) Tool*, 2000. URL: "`http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`".

[9] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.