An Extension Scheme of the Berkeley Socket Interface for Multipeer Multimedia Communications

Yong-Woon Kim

Protocol Engineering Center, ETRI

E-mail : qkim@pec.etri.re.kr

Abstract

This paper proposes an extension scheme of the well-known Berkeley Socket Interface for supporting QoS-provided multipeer multimedia communications. The current BSD Socket Interface has been TCP/IP-centric historically and the protocol suite was developed for the peer-to-peer best effort data communications. In order to support multiparty multimedia applications, such key functions as group management, QoS control, and reliable multicast transport should be exploited in the current simple unicast communication architecture and the well-known interface should be extended to cover the functions. This paper proposes a few new API functions and some changes in the current APIs.

I. Introduction

Interactive/distributed multimedia applications require a reliable concurrent multicast service based on transmitting user data from a single or multiple sources to all members of a multicast group.[1] The concurrent transmission of data to multiple recipients has been receiving increased attention since the late 1980s.[2] In particular, many applications in the Internet can benefit from multicasting their data to peer applications.[3,4] For many newer Internet protocol developments, multicasting is rapidly becoming the general case of datagram transmission, with unicasting just a special case.[5,6]

Current IP multicast technologies are based on RFC 1112 which has been updated by RFC 2236. The IGMP is used by IP hosts to report their multicast group memberships to any immediately-neighboring multicast routers. Multicast routers use IGMP to learn which groups have members on each of their attached physical networks. A multicast router keeps a list of multicast group memberships for each attached network, and a timer for each membership. "Multicast group memberships" means the presence of at least one member of a multicast group on a given attached network, not a list of all of the members.[7]

IP multicast allows sources to send to a multicast group without being a receiver of that group. However, for many conferencing purposes it is useful to know who is listening to the conference, and whether the media flows are reaching receivers properly. Accurately performing both these tasks restricts the scaling of the conference. IP multicast means that no-one knows the precise membership of a conference at a specific time, and this information cannot be discovered, as to try to do so would cause an implosion of messages, many of which would be lost. Instead, RTCP gives approximate membership information through periodic multicast of session messages which, in addition to information about the recipient.[8] So it has been said that the MBone is loosely coordinated for membership.

In networks that are able to carry a range of different types of traffic from different applications, this contract is usually expressed in terms of a set of performance measures commonly known as Quality of Service parameters.[9] An integrated-services network is a network designed to carry various types of traffic, and to offer various services in an attempt to satisfy the different requirements of those types. Different types of traffic have different QoS requirements; for example, an interactive voice packet is to be delivered with a relatively short delay and a small delay

variation; a file data needs reliable transmission and a reasonable transfer time; and so on. Thus, an integrated-services network must be governed by protocols that are sensitive to the QoS requirements of the various types of traffic and try to satisfy each one of them as much as possible.[10] The Internet has no basic, widely implemented way of expressing transmission rate and delay parameters, qualitatively or quantitatively.[9]

Up to now, UNIX applications that were to exploit the advantages of multicasting only had UDP available as a transport protocol.[11] UDP essentially relays the unreliable datagram semantics of IP to the transport level, only adding error detection (discard on error) and an additional layer of multiplexing. On top of UDP, specialized protocols such as RTP/RTCP have been built that converted these semantics into those actually needed by the application. [12,13]

For those requirements, a ubiquitous transport protocol is not enough: to enable the emergence of portable multicast and QoS based applications, the multicast transport service interface must be as widely available. For UDP and TCP, the Berkeley socket interface is the most widely available API. The current socket interface that was developed initially and has been used in the BSD UNIX operating system and its derivations. It supports only the reliable unicast transmission and IP multicast capability which contains join of a network multicast group and transmission of a data multicast. It cannot provide QoS and transport group functions. Section II describes new transport services specified in ECTS in order to support the requirements. Section III explains extensions to the socket API for ECTP qos-based multicast transport protocol. Then conclusions are offered in section IV.

II. Enhanced Transport Services

The existing transport services based on the OSI reference model or the TCP/IP protocol stack are designed for peer-to-peer unicast communications. So their protocol functions are very simple and poor. ISO/IEC JTC1/SC6 and ITU-T SG7 have worked for a common project of developing new standards for the multipeer multimedia communications of which key service functions are QoS provision, multicast transmission, and group management.

ECTS (Enhanced Communications Transport Service[14]) provides for those services and defines a wide range of services ranging from unreliable unicast with best-effort QoS to reliable multicast with guaranteed QoS. In this way, ECTS is meant to provide for a universal and uniform service interface between transport protocols and applications of the present and the future information age, especially for those applications requiring versatile and powerful multimedia group communication capabilities. ECTP is a protocol to provide the transport services as follows defined in ECTS.

- Create: can be used by the connection owner to establish a homogeneous TC (Transport Connection), provided the enrolled TS-users exist and are known to the TS-provider. It is assumed that there exists one and only one TC-owner who possesses the right to create and terminate a transport connection of a given enrolled group.
- Invite: can be used by the owner to invite the TS-users to collectively establishing a heterogeneous connection, provided the enrolled TS-users exist and are known to the TS-provider. A heterogeneous TC is established by individual establishment of multiple 1xN simplex channels, each by every focal TS-user through join primitives.
- Data Transfer: provides for two types of transfer of TSDUs from a sending TS-user to the other receiving TS-user(s). In one type, data transfer takes place over a successfully established TC using T-DATA primitives. In the other type, it takes place at any phase of a TC using T-UNITDATA primitives; it may take place even when no TC is available between the sending and the receiving TS-users.
- Pause: provides for the TS-provider to indicate with the T-PAUSE indication to the active group TS-user(s) that the TC has entered the state where the data transfer is not allowed. The reason parameter within the primitive should deliver the reason, e.g., violation of the QoS or the AGI.
- Resume: is used to resume the data transfer recovering from the temporarily violated TC-characteristics. After the receipt of the RESUME indication, the active group TS-user may restart issuing T-DATA request

primitives or receiving T-DATA indication ones.

- Report: is used to notify the change or selection of TC-characteristics to the active TS-users during data transfer or in TC establishments.
- Join: can be used by the focal TS-user to establish a heterogeneous TC, provided the enrolled TS-users exist and are known to the TS-provider, and by a TS-user to join an already existing homogeneous TC as a send and/or receive TS-user or a heterogeneous TC as a receive-only user.
- Leave: is used to remove a TS-user from the TC.
- Terminate: is used to terminate a TC. The termination may be initiated by the TC-owner or the TS-provider due to fatal failure of some TC-characteristics. It is permitted at any time regardless of the state of the TC.
- Token Manage: can be used by the TC-owner and other sending TS-users to pass around the token(s) for the right to transmit data.
- Ownership Transfer: can be used by the TC-owner and TS-users to pass the TC ownership.

Corresponding transport service primitives and their parameters can be summarized in Table 1.

Service	Primitives	Parameters
create	T-CREATE request	Called address, Calling address, TC-characteristics, TS-user data
	T-CREATE indication	Called address, Calling address, TC-characteristics, TS-user data
	T-CREATE response	Responding address, TC-characteristics, TS-user data
	T-CREATE confirm	Responding address, TC-characteristics, TS-user data
invite	T-INVITE request	Called address, Calling address, TC-characteristics, TS-user data
	T-INVITE indication	Called address, Calling address, TC-characteristics, TS-user data
data transfer	T-DATA request	TS-user data
	T-DATA indication	Calling address, Status, TS-user data
	T-UNITDATA request	Called address, Calling address, TC-characteristics, TS-user data
	T-UNITDATA indication	Called address, Calling address, TC-characteristics, Status, TS-user data
pause	T-PAUSE indication	Reason
resume	T-RESUME indication	Reason
report	T-REPORT indication	Reason
join	T-JOIN request	Called address, Calling address, TC-characteristics, TS-user data
	T-JOIN indication	Called address, Calling address, TC-characteristics, TS-user data
	T-JOIN response	Responding address, TC-characteristics, TS-user data
	T-JOIN confirm	Responding address, TC-characteristics, TS-user data
leave	T-LEAVE request	Called address, Calling address, TS-user data
	T-LEAVE indication	Called address, Reason
terminate	T-TERM. request	TS-user data
	T-TERM. indication	Rereason, TS-usdata
ownership	T-OWNER request	Called address, Calling address, TS-user data
	T-OWNER indication	Called address, Calling address, TS-user data
	T-OWNER response	Responding address, TS-user data
	T-OWNER confirm	Responding address, TS-user data
token give	T-GIVE request	Called address, Calling address, TS-user data
	T-GIVE indication	Called address, Calling address, TS-user data
	T-GIVE response	Responding address, TS-user data
	T-GIVE confirm	Responding address, TS-user data
token get	T-GET request	Called address, Calling address, TS-user data
	T-GET indication	Called address, Calling address, TS-user data
	T-GET response	Responding address, TS-user data
	T-GIVE confirm	Responding address, TS-user data

Table 1 - Transport Services and Their Primitives

III. API Extensions

Group communication presents by necessity a more complex service interface than point-to-point communication. It is important that the ECTP API does not further confuse application programmers, that is, the interface should be compatible both syntactically and semantically to well-known existing interfaces as far as possible and be the same for both kernel driver and user-mode daemon implementation of ECTP.

The BSD socket interface is the widely accepted UNIX interface for interprocess communication today and the majority of application programmers are expected to be comfortable with it. Therefore it has been chosen as the basis for the ECTP API.

ECTP[15] has been designed for providing the enhanced transport services. A transport service user calls an API function that is a means to interact with the transport service provider through an access point called port.

The service provider is a collection of every participating transport entity which may exchange control packets for cooperation. API functions are summarized below. In the function names, "m" means to support the reliable multicast transport based on QoS.

Some of the functions such as msocket, mlisten, mrecv, and so on, have been extended in their semantic definitions and parameters and the other ones such as mrespond, mtokenget, mownertr, and so on have been specified newly.

(1) msocket()

It creates a new unnamed clientt/server or multipeer socket within the ECTP domain. To perform ECTP network I/O, the first thing a process must do is calls the msocket function, specifying the type of communication protocol desired (ECTP using IPv4, ECTP using IPv6, ECTP using OSI protocols, etc.).

```
int msocket(int family, int type, int
  protocol, int role);
```

Returns: non-negative descriptor if OK, or -1 on error

- *family*: specifies the protocol family such as AF_INET, AF_INET6, AF_ISO, etc.;
- *type*: specifies the type of socket such as SOCK_STREAM, SOCK_DGRAM, or SOCK_TSDU.;
- *protocol*: is set to 0 except for raw sockets; and
- *role*: specifies the role of this calling initiator such as TC_OWNER, TC_USER, TC_CLIENT, or TC_SERVER.

(2) mbind()

With the Internet protocols the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit ECTP port number. It associates a pair of local and group addresses along with each port to a socket.

```
int mbind(int msockfd, const struct
  sockaddr *myaddr, socklen_t myaddrlen,
  const struct sockaddr *grpaddr,
  socklen_t grpaddrlen);
```

Returns : 0 if OK, or -1 on error.

- *msockfd*: is a socket descriptor that was returned by the msocket function;
- *myaddr*: is a pointer to a protocol-specific address to bind a local address to the above socket;
- *myaddrlen*: is the size of the above address structure;
- *grpaddr*: is a pointer to a protocol-specific address to bind a target group address to the socket; and
- *grpaddrlen*: is the size of the group address structure.

In the ECTP multipeer model, the two ports for the local and group addresses must be identical each other for distinguishing multiple connection end points. A process can mbind a specific IP address or network address and a target group network address to its socket. The source and group addresses must belong to an interface on the host. For an ECTP user, these assign the source IP address and the group IP address that will be used for IP datagrams sent on the socket and the group address restricts the socket to receive an incoming connection destined only to that IP addresses.

Otherwise, in the client/server model, servers bind their well-known port when they start. Operations are the same as in the existing client/server model.

(3) mconnect()

It initiates a connection creation to a specified foreign address and contains the initiator's QoS proposal. It may be mapped to a T-CREATE request and its return to either T- CREATE confirm, if OK, or T-TERMINATE indication, if failed.

Or it initiates a late join process. It may correspond to a T-JOIN request and its return to either T-JOIN confirm, if OK, or T-LEAVE indication, if failed.

```
int mconnect(int msockfd, const struct
  sockaddr *rmtaddr, socklen_t
  rmtaddrlen);
```

Returns: a code number specifying what kind of protocol control information is generated to user if OK, or -1 on error.

- *rmtaddr*: is a pointer to a protocol-specific address to which a connection initiator opens a connection or a late joining user asks his join request. Thus this address may be a group address or a peer address; and
- *rmtaddrlen*: is the size of the above remote address structure.

(4) mlisten()

It converts an unconnected socket into a passive socket, indicating that the kernel should accept an incoming connection request, and waits for and accept a single connection. Its return message corresponds to a T-CREATE

indication.

The semantics of the function is somewhat different in the multipeer model and the client/server model.

```
int mlisten(int msockfd, int backlog);
```

Returns: 0 if OK, or -1 on error.

- *backlog*: specifies the maximum number of connections or data channels that the kernel should queue for this socket.

When a multipeer socket is created by the msocket function, it is assumed to be an active socket, that is, the owner socket that will issue an mconnect. The mlisten function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming channel requests directed to this socket in the heterogeneous connection type. The second argument, *backlog*, to this function specifies the maximum number of channels that the kernel should queue for this multipeer socket. But it should be 1 in the homogeneous connection type because there should be only one single connection in a multipeer group. This function is normally called after both the msocket and mbind functions and must be called before calling the mrespond function.

(5) maccept()

It returns a next completed connection from the front of the completed connection queue and is called by an ECTP server. This function can be used only for the existing peer-to-peer client/server model. If the connection queue is empty, the process is put to sleep.

```
int maccept(int msockfd, struct sockaddr
   *cliaddr, socklen_t *cliaddrlen);
```

Returns: non-negative descriptor if OK, or -1 on error.

- *cliaddr*: returns the protocol address of the connected peer process (the client); and
- *cliaddrlen*: is a pointer to the size of the socket address structure pointed by *cliaddr*.

If maccept is successful, its return value is a brand new descriptor that was automatically created by the kernel.

(6) mrespond()

It responds in order to take part in a creating connection, i.e., T-CREATE response, accept the ownership transferred from the owner, i.e., T-OWNER response, accept a token given from the owner, i.e., T-GIVE response of a TS-user, or receive a coming-back token, i.e., T-GIVE response of the owner, or does to a join request initiated by a late joining TS-user, i.e., T-JOIN response of the owner, to a token retrieval of the owner, i.e., T-GET response of a TS-user, or to a token request initiated by a TS-user, i.e., T-GET response of the owner.

This function is used by an ECTP user including the owner. It provides multiple protocol functions as above. The returning *flags* are indicated locally by a transport entity. Other indication *flags* are returned by the mrecv function after the exchanges.

```
int mrespond (int msockfd, const struct
   sockaddr *rmtaddr, socklen_t rmtaddrlen,
   int flags);
```

Returns: zero if OK, or -1 on error.

- *rmtaddr*: is a pointer to a protocol-specific address to which a connection initiator opens a connection or a late joining user asks his join request. Thus this address may be a group address or a peer address; and
- *flags*: specifies what kind of protocol control information is generated to the kernel.

(7) msend()

It sends data from a single buffer into a connected socket. It may correspond to a T-DATA request.

```
ssize_t msend (int msockfd, const void
     *userdata, size_t nbytes, int *flags);
```

Returns: number of bytes written if OK, or -1 on error.

- *userdata*: is a pointer to buffer to write from; and
- *nbytes*: is the number of bytes to write.

(8) msendto()

It sends a unit of data to a specified address. It may correspond to a T-UNITDATA request. The "to" argument for "msendto" is a socket address structure containing the protocol address of where the data is to be sent.

```
ssize_t msendto (int msockfd, const void
     *userdata, size_t nbytes, int *flags,
     const struct sockaddr *toaddr,
     socklen_t toaddrlen, tc_chars *chars);
```

Returns: number of bytes written if OK, or -1 on error.

- *toaddr*: is a pointer to a protocol-specific address to which user data is sent. It may be a group address or a peer address;

- *toaddrlen*: is the size of the above *toaddr* address structure; and
- *chars*: is a pointer to a TC characteristics structure containing the AGI and QoS parameters.

(9) mownertr()

The TC-owner only can use this function when he wants to transfer the ownership to a specified user or a group of members for contention. It can be mapped to a T-OWNER request.

```
int mownertr (int msockfd, const struct
   sockaddr *myaddr, socklen_t myaddrlen,
   const struct sockaddr *rmtaddr,
   socklen_t rmtaddrlen) ;
```

Returns: zero if OK, or -1 on error.

- *myaddr*: is a pointer to a protocol-specific address, specifying the TC-owner; and
- *rmtaddr*: is a pointer to a protocol-specific address to which the ownership transfer packet is transmitted in unicast or multicast.

(10) mtokenget()

The TC-owner or a TS-user calls this mtokenget function when the owner wants to withdraw a token from a TS-user or the TS-user wants to get the floor. The former case is called the token retrieval, i.e., the owner's T-GET request, and the latter one is called the token request, i.e., the TS-user's T-GET request.

```
int mtokenget (int msockfd, const struct
   sockaddr *myaddr, socklen_t myaddrlen,
   const struct sockaddr *rmtaddr,
   socklen_t rmtaddrlen) ;
```

Returns: zero if OK, or -1 on error.

(11) mtokengive()

The TC-owner or a TS-user calls this mtokengive function when the owner allocates a token to a TS-user or a TS-user wants to return back the floor to the owner. The former case is called the token allocation, i.e., the owner's T-GIVE request and the latter one is called the token return, i.e., a TS-user's T-GIVE request.

```
int mtokengive (int msockfd, const
   struct sockaddr *myaddr, socklen_t
   myaddrlen, const struct sockaddr
   *rmtaddr, socklen_t rmtaddrlen) ;
```

Returns: zero if OK, or -1 on error.

(12) mrecv()

This function is used for a lot of actions by an ECTP user including the owner. It provides multiple protocol functions as below and delivers some information to a TS-user. It can be mapped appropriately to a T-DATA ind., T-REPORT ind., T-PAUSE ind., T-RESUME ind., T-LEAVE ind., T-JOIN ind., T-OWNER ind., T-OWNER con., T-GIVE ind., T-GIVE con., T-GET ind., T-GET con., or T-TERMINATE ind..

```
ssize_t mrecv (int msockfd, void
     *userdata, size_t nbytes, int *flags,
     struct sockaddr *fromaddr,
     socklen_t *fromaddrlen) ;
```

Returns: number of bytes read if OK, or -1 on error.

- *userdata*: is a pointer to buffer to read into;
- *nbytes*: is the number of bytes to read into; and
- *fromaddr*: is a pointer to a protocol-specific address to specify the sender.

When a TS-user receives user data from the buffer, he must evaluate *fromaddr* to distinguish senders and deliver each user data into a buffer for a specific sender. Thus an application may handle multiple receiving buffers for every concurrent sender.

(13) mrecvfrom()

It delivers a unit of data and address of sender, corresponding to a T-UNITDATA ind. The function fills in the socket address structure pointed to by *fromaddr* with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *fromaddrlen*.

```
ssize_t mrecvfrom (int msockfd, void
     *userdata, size_t nbytes, int *flags,
     struct sockaddr *fromaddr,
     socklen_t *fromaddrlen, tc_chars
     *chars) ;
```

Returns: number of bytes read if OK, or -1 on error.

- *userdata*: is a pointer to buffer to read into;
- *fromaddr*: is a pointer to a protocol-specific address to specify the sender;
- *fromaddrlen*: is the size of the above *fromaddr* address structure; and
- *chars*: is a pointer to a TC characteristics structure containing the AGI and QoS parts.

When a TS-user receives user data from the buffer, he must evaluate *fromaddr* to distinguish senders and deliver each user data into a buffer for a specific sender. Thus an application

may handle multiple receiving buffers for every concurrent sender.

(14) mclose()

It terminates an existing connection and releases its associated socket. It may correspond to a T-TERMINATE request, if the initiator is the owner, or a T-LEAVE request, if it is a non-owner TS-user. The default action with an ECTP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: it cannot be used as an argument to msend or mrecv.

```
int mclose (int msockfd);
```

Returns: 0 if OK, or -1 on error.

IV. Conclusion

Making multicasting accessible to application developers would be greatly facilitated by ubiquitous deployment of a suitable reliable multicast transport protocol. This paper proposes to use ECTP as this protocol, as it is scalable, reasonably efficient for most applications, reliable, and provides globally ordered delivery of messages, thus simplifying application development.

Modified and extended API functions which has been designed to use the ECTP have been described in this paper. Since their structures are similar to those of the existing socket API, developers will be comfortable with the proposed functions.

ECTP is being implemented on FreeBSD as a user-mode daemon now, but will be done as a kernel module at the next version. These results will be contributed to ISO/IEC JTC1/SC6 WG7 ECTP project having a goal of designing an enhanced transport protocol.

References

[1] Brian Neil Levine, David B. Lavo and J.J. Garcia- Luna-Aceves, "The Case For Reliable Concurrent Multicasting Using Shared Ack Trees", Proc. ACM Multimedia 1996 Boston, MA, November 18--22, 1996. http://www.cse.ucsc.edu/research/ccrg/publications/brian.mm96.ps.gz

[2] S. Deering, "Host Extensions for IP Multicasting", IETF RFC 1112, August 1989

[3] Kurt Lidl, Josh Osborne, and Joseph Malcolm, "Drinking From the Firehose: Multicast Usenet News", in Proc. of the 1994 Winter USENIX, pp. 33-45, January 1994

[4] Stephen Casner and Stephen Deering, "First IETF Internet Audiocast", ACM Computer Communications Review 22(3), July 1992

[5] Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala, "RSVP: A New Resource ReSerVation Protocol", IEEE Network 7(5):8-18, September 1993

[6] Frank Kastenholz and Craig Partridge, "Technical Criteria for Choosing IP: The Next Gerneration (IPng)", Internet Draft, March 1994

[7] Sridhar Pingali, Don Towsley, and James F. Kurose, "A comparison of sender-initiated and receiver-initiated reliable multicast protocols", Proceedings Of ACM SIGMETRICS 94, Vol. 14, pp. 221-230, 1994

[8] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", IEEE/ACM Transactions on Networking, Nov. 1996

[9] Jon Crowcroft, Ian Wakeman, Mark Handley and Stuart Clayman, "Internetworking Multimedia", UCL Press, January 1997

[10] Demenico Ferrari, "Multimedia network protocols: where are we?", Technical Report of the Tenet Group

[11] J. Postel, "User Datagram Protocol", IETF RFC 768, August 1980

[12] H. Schulzrinne and S. Casner, "RTP: A Transport Protocol for Real-Time Applications", IETF RFC 1889

[13] Walid Dabbous and Blaise Kiss, "A reliable multicast protocol for a white board application", INRIA Centre de Sophia Antipolis, November 1993

[14] ISO/IEC JTC1/SC6 ECTS FDIS 13252, Sep. 1998

[15] ISO/IEC JTC1/SC6 ECTP CD 14476, Feb. 1998