XTP Application Programming Interface[1]

Roland Mechler and Gerald W. Neufeld
Department of Computer Science
University of British Columbia
Vancouver, B.C., V6T 1Z4
Canada

Abstract

The Xpress Transport Protocol (XTP) is a lightweight transport protocol intended for high-speed networks. High-speed networks provide bandwidths of 100 Mbps and beyond, enabling a new class of applications (e.g., multimedia). So as not to be a bottleneck in the delivery of data, a transport protocol must provide high performance. Features of XTP which enhance performance include implicit connection setup, sender driven acknowledgement, selective retransmission, fixed format word aligned packet structures and suitability for parallel implementation. Since the new generation of applications may require a variety of services from the transport layer, a transport protocol designed for high-speed networks should also be flexible enough to provide these services. XTP provides the mechanisms to allow applications to tailor the functionality of the protocol to their individual needs. In particular, XTP provides flow control, rate control and error control, the use of each being optional and orthogonal to the others. This report describes an Application Programming Interface designed for a multi-threaded implementation of XTP. The API allows all XTP parameters to be set from the application level, and addresses the issue of performance by providing a mechanism for zero copy transmission and reception of data.

1. This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Canadian Institute for Telecommunications Research.

Table Of Contents

1.0 Introduction 3

2.0 XTP Background 3

2.1 Packet Types 6
2.2 Services 6
2.3 Command Options 7
2.4 Modes 7

3.0 API Reference Guide 7

3.1 API Routines 8
3.2 Service Structure 11
3.3 Configuration Structure 13

4.0 Programming Guide 14

4.1 XTP Initialization 15
4.2 Contexts 15
4.3 Connection Establishment 16
4.4 Data Transmission 17
4.5 Closing Procedures 18
4.6 XTP Modes 19

5.0 Addressing 21

6.0 Packet Encapsulation 21

7.0 Example 22

7.1 Sender Program 22
7.2 Receiver Program 23

References 24

1 Introduction

This is a brief guide to the Application Programming Interface (API) for a multithreaded implementation of XTP (Xpress Transport Protocol), Revision 4.0 [4]. The API is loosely based on the one outlined in the XTP Definition, Revision 3.6 [3]. Henceforth in this document, XTP will refer to the XTP Definition, Revision 4.0.

This implementation of XTP was originally designed to run in the *r-kernel*[1] multiprocessor operating system environment, but now runs in the RT Threads [6] environment as a multithreaded user level UNIX process. The multicast features of XTP are not included in this implementation.

Some background on XTP is given in section 2 to provide a better understanding of how to use the interface[2]. Section 3 is a reference guide for the API, describing the interface features. Section 4 is a programming guide, giving a better idea of how to use the interface routines. Sections 5 and 6 give some extra information on addressing and encapsulation. Section 7 shows an example program.

2 XTP Background

The Xpress Transport Protocol (XTP) is a lightweight transport protocol intended for high-speed networks.

XTP is a connection oriented protocol which allows bi-directional data transfer. Because implicit connection setup is possible with XTP, it can also provide connectionless semantics. At each endpoint of an XTP connection, a *context* maintains all state information for the connection. Each context is identified by a *key*, which is unique at a given endpoint.

XTP seeks to provide both better performance and greater functionality than traditional transport protocols like TCP/IP: better performance because protocols can become a major bottleneck as network bandwidth increases; greater functionality because new applications making use of high-speed networks (e.g., multimedia) require services which protocols like TCP do not provide.

The features of XTP which are intended to improve performance include:

- implicit connection setup
- efficient indexed context lookup (key exchange)
- sender driven acknowledgement scheme which can reduce timer overhead and acknowledgment traffic

1. The r-kernel is a lightweight multiprocessor kernel with user-level threads. It currently runs in the Motorola MVME188 Hypermodule hardware environment.
2. Because much of the functionality of the XTP protocol can be controlled by the application, knowledge of how the protocol works is an asset when using the interface. Reading the XTP 4.0 specification [4] is recommended.

- selective retransmission (or no retransmissions)
- fixed format, word-aligned packet structures for efficient processing
- suitability for parallel implementation.

Each byte of data in a stream is represented by a sequence number. XTP uses a 64-bit sequence space in anticipation of sequence number aliasing problems which could occur with 32-bit sequence numbers in networks with a high bandwidth-delay product.

XTP allows protocol parameters to be set at the application level so the functionality of the protocol can be tailored to individual application needs. Central to this idea of parameterized functionality is the orthogonality of the following functions: *rate control*, *flow control* and *error control*.

The two streams of a bi-directional XTP connection are independent and may have different parameter values. For example, the stream in one direction might be flow controlled and error controlled, but not rate controlled, while the stream in the opposite direction might be rate controlled but not flow controlled or error controlled.

A short description of the parameterized functions follows:

rate control - Rate control is used to set the maximum rate at which the sender will transmit data. There is no guaranteed minimum rate. The current policy for rate control negotiation is to use the lower one of each parameter value (i.e., whichever one will lead to a lower rate). The parameters for rate control are

rate - This is the maximum rate, in bytes/sec, at which the sender will transmit data. A *rate* value of zero will halt transmission.

burst - This is the maximum number of bytes that will be transmitted in a single burst. The interval between bursts, known as the RTIMER value, is calculated as *rate/burst*. A *burst* value of zero will allow unrestrained transmission (i.e., no rate control). A *burst* value of zero takes precedence over a *rate* value of zero.

flow control - Flow control specifies the maximum amount of outstanding unacknowledged data which may be transmitted by the sender. A sliding window mechanism is used, whereby the receiver transmits *alloc* values to the sender (at the XTP level), where *alloc* is one greater than the highest sequence number which the sender is allowed to transmit. Flow control may turned off (i.e., infinite window size) by using NOFLOW mode. Flow control has the following parameter:

rwindow - This is the size of the window, in bytes, as set by the receiver. If the sender wants to transmit data before receiving any acknowledgements (i.e., control packets) from the receiver, it needs to set its own initial window size, using the *swindow* parameter.

error control - Checksums[1] are used for both the header and the data payload of XTP packets. The use of checksums for data can be turned off by using NOCHECK mode.

XTP allows selective retransmission, meaning that only lost or corrupted packets are retransmitted. This implementation uses selective retransmission. Retransmission can be turned off by using NOERR mode.

Other features of XTP which are parameterized include:

addressing - Rather than defining a new addressing scheme, or using a single existing one, XTP allows the use of a variety of addressing formats.

timers - Timers are specified by their timeout values, currently measured in microseconds. The following timers are used by XTP, some of whose values can be set by the user:

WTIMER, *retry_count* - WTIMER is the wait timer for a response to a status request. Its value is calculated by XTP based on measured round trip times. An initial value can be specified by the user. *retry_count* (which can be set by the user) specifies the number of retries which will be attempted before the connection aborts. The WTIMER value is doubled for each retry. A successful request/acknowledgement is called a *synchronizing handshake*.

CTIMEOUT - This timer puts a limit on the overall time for the WTIMER retry sequence (synchronizing handshake). Its value can be set by the user.

CTIMER - This is the connection keepalive timer. Its value can be set by the user. If no packets at all are received on a connection within this interval, a synchronizing handshake is initiated. CTIMER is intended to be a long term timer, typically an hour or more in duration. At the very minimum it should be longer than CTIMEOUT.

RTIMER - This is the rate control burst interval. Its value is determined by the *rate* and *burst* parameters (which can be set by the user).

packet size - The *maxdata* parameter can be used to set the maximum packet data size for transmission.

1. The IP checksumming algorithm is used.

2.1 Packet Types

The XTP packet types which are of concern are:

FIRST - These packets are sent by the initiating context to establish a connection. The FIRST packet contains an Address Segment (used to locate a listening context at the receiving endpoint) and a Traffic Segment (used to establish rate control parameters and a maximum data size). The FIRST packet may carry data as well.

DATA - These are data bearing packets and may travel in either direction since XTP connections are bi-directional.

CNTL - These contain control information. A receiver (i.e., for a given direction of data transfer) will usually only send CNTL packets in response to requests from the sender. CNTL packets are used mostly for flow control. ECNTL and TCNTL packets contain extra information for error control and rate control.

ECNTL - These error control packets contain the same information as CNTL packets, plus a list of sequence number spans for data to be retransmitted.

TCNTL - These traffic control packets contain the same information as CNTL packets, plus a Traffic Segment to indicate changes in rate control parameters.

DIAG - Diagnostic packet, returns error information to a sending context (e.g., non-existent context at receiver).

2.2 Services

Implicit connection setup together with parameterization allows XTP to provide a variety of connection-oriented and connectionless service semantics. Standard service types are listed below.

`XTPUNSPECIFIED`	- unspecified
`XTPUNACKDG`	- traditional unacknowledged datagram service
`XTPACKDG`	- acknowledged datagram service
`XTPTRANS`	- transaction service
`XTPSTREAM`	- traditional reliable unicast stream service
`XTPUMULTISTREAM`	- unacknowledged multicast stream service
`XTPRMULTISTREAM`	- reliable multicast stream service

Service types are currently not interpreted by the XTP implementation (except for the purpose of matching incoming requests). It is up to the application to provide these semantics using the parameterized features of XTP.

2.3 Command Options

There is an *options* bitfield present in every XTP packet. These option bits are used internally by XTP, but the following options may also be set at the application level using the *options* parameter of *XtpWrite()* (see section 4.1).

`XTPBTAG`	Packet contains out-of-band data (first 8 bytes of data).
`XTPEND`	Cease further communication.
`XTPWCLOSE`	Local writer is closed. No new data will be sent.
`XTPDREQ`	Request CNTL packet response from receiver *after* all queued data has been delivered to user.
`XTPSREQ`	Request CNTL packet response from receiver immediately upon receiving packet.

Note that an application is not required to request CNTL packet responses using `XTPDREQ` and `XTPSREQ`, as the necessary status requests will be handled internally by XTP. These features are included in the interface to provide flexibilty, but it is not anticipated that they will be used often in practice.

2.4 Modes

There are a number of modes under which XTP can operate (on a per connection basis). These may be used in combination, although when FASTNAK mode is used with NOERR mode, FASTNAK is ignored. The modes are described as follows:

FASTNAK	The receiver should send an acknowledgement (i.e., `ECNTL` packet) immediately when a misordered packet is encountered.
NOFLOW	Flow control is turned off.
RES	Reservation mode. The receiver will only accept data for which buffer space has been reserved at the application level (using `XtpReserve`).
NOERR	Error control is turned off (i.e., no retransmissions).
NOCHECK	Checksum calculations are not performed on data.

The two streams of a bi-directional XTP connection may operate under different modes. The mode for a particular stream is determined by the sending endpoint.

3 API Reference Guide

The API is defined by routines for context creation, connection establishment and reading and writing data. Service parameters are passed to XTP using the service structure (type `XtpService`), which can be passed to the protocol at context creation (thus they apply per context).

3.1 API Routines

The following routines are available. All routines return either `XTPOK` or `XTPFAILED` unless otherwise noted. Contexts are referenced by context identifiers (of type `XtpCtxtId`):

(1) *int XtpInit(u_int ipAddr, u_int portNo)* initializes XTP and should be called only once per threads environment (UNIX process). The argument *ipAddr* is used to specify the local IP address, as an unsigned integer value. If 0 is specified for *ipAddr*, XTP will determine the IP address to use[1]. The argument *portNo* is used to specify the local port number for the XTP engine, which must be unique among active XTP engines using the same IP address. If 0 is specified for *portNo*, XTP will choose a port number. The IP address and port number chosen by XTP can be determined after the call to *XtpInit()* with the routines *XtpMyIP()* and *XtpMyPort()*.

(2) *int XtpMyIP(u_int *ipAddr)* and *int XtpMyPort(u_int *portNo)* return, by reference, the IP address and port number, respectively, being used by the local XTP engine. Both routines return `XTPFAILED` if called before either *XtpInit()* or *RttNetInit()* have been called.

(3) *int XtpConfiguration(int command, XtpConfig *config, XtpService *svc)* allows the default configuration for the local XTP engine to be set at the application level. The *command* argument indicates whether the parameters are to be set or retrieved, by specifying either `XTPSET` or `XTPGET`. The *config* argument is used to access the configuration parameters (see section 3.3) for the XTP engine. The *svc* argument is used to access the default context parameters (see section 3.2), used to initialize contexts (unless overridden at context creation). If `XTPNULL` is used to specify either *config* or *svc*, that argument will be ignored. *XtpConfiguration()* will fail if called using the `XTPSET` command after calling *XtpInit()* (the configuration may not be changed once the XTP engine is started).

(4) *int XtpCreateContext(XtpCtxtId *ctxtId, XtpService *svc)* creates a new context, returning a context identifier *ctxtId* by reference. It takes pointer to service structure *svc* as argument, allowing default values of context parameters to be overridden. If `XTPNULL` is used to specify *svc*, the argument will be ignored and the context will be initialized using the default values.

(5) *int XtpCloseContext(XtpCtxtId ctxtId)* closes the given context, freeing all of its resources[2]. If the context is an endpoint of an active connection, *XtpCloseContext()*

---

1. If the local machine has more than one IP address, then specifying 0 for *ipAddr* may result in any one of these being used.

will block until the connection has closed gracefully (i.e., all data has been transmitted reliably, if not in NOERR mode). After the call, *ctxtId* becomes an invalid context identifier.

(6) *int XtpAbortContext(XtpCtxtId ctxtId)* is similar to *XtpCloseContext()*, except it forces an abortive close (untransmitted data will be discarded). *XtpAbortContext()* will not block.

(7) *int XtpEntry(XtpCtxtId ctxtId, int command, XtpService *svc)* is used to access the context parameters. The following commands are available:

`XTPGET` - Returns context parameters using *svc*, as requested using the field flags.

`XTPSET` - Sets context parameters using *svc*, as requested using the field flags.

(8) *int XtpListen(XtpCtxtId ctxtId, int mode, u_int localXtpPort,)* allows context *ctxtId* to listen for a connection establishment request from a remote endpoint. Parameter *localXtpPort* specifies the XTP port number (32-bit unsigned integer) which acts as a filter for incoming FIRST packets. Parameter *mode* is used to indicate the desired blocking semantics, and can take on the following values:

`XTPBLOCK` - Block until connection is established (FIRST packet received).

`XTPNOBLOCK` - Don't block.

(9) *int XtpConnect(XtpCtxtId ctxtId, int mode, u_int remoteAddr, u_int remotePort, u_int remoteXtpPort)* establishes a connection. Parameter *mode* specifies whether it is an `XTPIMPLICIT` or an `XTPEXPLICIT` connection establishment. If `XTPEXPLICIT` is used, a FIRST packet is sent and the call will block until a response is received. If `XTPIMPLICIT` is used, the call will not block, and the FIRST packet will be sent as a result of the first *XtpWrite()* call. Parameter *remoteAddr* specifies the remote IP address, which can be determined using the *XtpHostnameToIP()* routine, see section 4. Parameter *remotePort* specifies the remote port number that the destination XTP engine was initialized with. Parameter *remoteXtpPort* specifies the XTP port number at which the remote context should be listening.

(10) *int XtpWrite(XtpCtxtId ctxtId, char *data, int len, int options, void (*func)(void *), void *arg)* writes a message of length *len* bytes from buffer *data* to the connection, without copying the data. Option flags specified in *options* (e.g., `XTPWCLOSE`) will

---

2. If *XtpCloseContext()* or *XtpAbortContext()* are called by the initiating endpoint of communication, the resources for the context may not actually be freed until a period of `CTIMEOUT` elapses. See [4].

be set in the XTP packet used to transmit the message. If *len* is greater than the maximum data size for an XTP packet (*maxdata*), the message will be segmented into multiple XTP packets and *options* will only cause bits to be set in the last packet of the message[1] (with the exception of `XTPBTAG`, which is set in the first packet of the message). More than one option can be specified using a logical OR (e.g., `XTPWCLOSE | XTPSREQ`). `XTPBTAG` is used to tag the first 8 bytes of a message as *out of band* data[2]. XTP does not interpret the `XTPBTAG`, it merely passes it to the receiving application.

Function *func* is called with argument *arg* when the data is no longer needed by the XTP implementation, i.e., when it has been successfully transmitted (and acknowledged when not in NOERR mode). The most common uses are for passing in a freeing routine to free the data, or for passing in a semaphore and signal routine so the calling thread can be blocked until the write is complete. If `XTPNULL` is used to specify *func*, this feature will be ignored.

(11) *int XtpReadBuf(XtpCtxtId ctxtId, char **data, int *bytes, int *flags, int *options)* returns the data buffer from one XTP packet in *data*, and returns the number of bytes read via reference parameter *bytes*. This data buffer must be freed by the application using *XtpFreeBuf()*. Parameter *options* contains the option bits from the packet. One or more of the following flags may be returned in *flags*:

`XTPEOC` - End-of-context, meaning there will be no more data to read.

`XTPNULLDATA` - There is no data. Parameter *bytes* represents the gap of missing data (for `NOERR` mode).

(12) *int XtpFreeBuf(char *buf)* frees a buffer which was obtained from *XtpReadBuf()*.

(13) *int XtpBufAvailable(XtpCtxtId ctxtId)* returns `XTPOK` if there is a buffer available for reading with *XtpReadBuf()*, and `XTPFAILED` otherwise.

(14) *int XtpRead(XtpCtxtId ctxtId, char *data, int len, int *bytes, int *flags, int *options)* reads (copies) *len* bytes into buffer *data* from the connection and returns number of bytes read via reference parameter *bytes*. Parameters *flags* and *options* are as above. `XTPNULLDATA` indicates only that at least some of the data is missing, but no indication is given as to which data is missing.

---

1. Option bits may also be set by the XTP implementation itself, irrespective of those set by the user.
2. Out of band simply means that the data is tagged for the application to use as it wishes. This data is sequenced with the rest of the data, XTP does not provide a facility for expedited data within a given connection.

(15) *int XtpDiagnostic(XtpCtxtId ctxtId, int *code, int *val)* returns `XTPOK` if a DIAG packet was received by the context, and `XTPFAILED` otherwise. If `XTPOK` is returned, reference parameters *code* and *val* will be set to the code and value of the DIAG packet.

(16) *int XtpPrintDiagnostic(char *message, int code, int val)* is a convenience routine which prints a message giving the meaning of the *code* and *val* parameters passed in. This routine will commonly be used in conjunction with the *XtpDiagnostic()* routine.

(17) *int XtpError(XtpCtxtId ctxtId, char *message)* when called upon failure of an XTP interface service routine, prints given message, followed by a message describing the error.

3.2 Service Structure

The service structure (type `XtpService`) defines the parameter interface to XTP. A reference to such a structure can passed as an argument to *XtpConfiguration(), XtpCreateContext()* and *XtpEntry()*. The structure is defined as follows:

```
typedef struct {
   unsigned long fieldFlags;   /* bitwise OR of selected fields */
   unsigned long modes;        /* XTPMODES                      */
   unsigned int reserveSize;   /* XTPRESERVESIZE                */
   char wservice;              /* XTPWSERVICE                   */
   char rservice;              /* XTPRSERVICE                   */
   unsigned int maxdata;       /* XTPMAXDATA                    */
   unsigned int rwindow;       /* XTPRWINDOW                    */
   unsigned int swindow;       /* XTPSWINDOW                    */
   long inrate;                /* XTPINRATE                     */
   long inburst;               /* XTPINBURST                    */
   long outrate;               /* XTPOUTRATE                    */
   long outburst;              /* XTPOUTBURST                   */
   unsigned int ctimer;        /* XTPCTIMER                     */
   unsigned int ctimeout;      /* XTPCTIMEOUT                   */
   unsigned int wtimer;        /* XTPWTIMER                     */
   unsigned int retryCount;    /* XTPRETRYCOUINT                */
   XtpAddress address;         /* XTPADDRESS                    */
   XtpTraffic traffic;         /* XTPTRAFFIC                    */
   unsigned long diagCode;     /* XTPDIAGCODE                   */
   unsigned long diagValue;    /* XTPDIAGVALUE                  */
} XtpService;
```

The *fieldFlags* field is a bitmap of the parameters which the user wishes to select at runtime. The user should set this field as a bitwise OR of the appropriate constants (as listed in comments beside the fields), and set the actual fields to their desired values (each constant corresponds to one of the fields of the service structure). Any parameters (fields) not specified by *fieldFlags* will take on default values.

The *modes* field is a bitmap used to select the mode(s) (see section 3.4) under which XTP is to operate. The modes can be selected independently for the stream in either direction, and are set by the sender in each case. The user should set this field as a bitwise OR of the appropriate constants from those listed below.

`XTPFASTNAK` - select FASTNAK mode
`XTPNOFLOW` - select NOFLOW mode
`XTPRES` - select RES mode
`XTPNOERR` - select NOERR mode
`XTPNOCHECK` - select NOCHECK mode
`XTPRCLOSE` - select simplex (unidirectional) connection from initiator to listener

The following table gives a description of each field of the `XtpService` structure, as well as its default value:

Table 1: Service Fields

Field Name	Description	Default Value
`modes`	Bitwise OR of modes selected for context. Will only be set at context creation times.	0
`reserveSize`	Size of reserved buffer space at receiver, in bytes, used for RES mode.	448400
`wservice`	Service type requested by the sender (connection initiator).	`XTPUNSPECIFIED`
`rservice`	Service type provided by receiver. `XTPUNSPECIFIED` indicates that any type will be accepted.	`XTPUNSPECIFIED`
`maxdata`	Maximum data (in bytes) per XTP packet.	8968
`rwindow`	Window size enforced by receiver for flow control.	15000
`swindow`	Window size used by sender for flow control, prior to any CNTL packets from receiver.	15000
`inrate`	Maximum data rate (in bytes/second) enforced by receiver for rate control.	0
`inburst`	Maximum burst size (in bytes) enforced by receiver for rate control.	0

Table 1: Service Fields

Field Name	Description	Default Value
`outrate`	Maximum data rate (in bytes/second) used by sender for rate control, prior to any response from receiver (for implicit connection establishment).	0
`outburst`	Maximum burst size (in bytes) used by sender for rate control, prior to any response from receiver (for implicit connection establishment).	0
`ctimer`	Keepalive timer (in microseconds).	3600000000
`ctimeout`	Handshake timeout (in microseconds), puts an overall limit on the time for a synchronizing handshake.	20000000
`wtimer`	Initial value for wait timer (in microseconds). Time to wait for response to SREQ. This value will change dynamically according to round trip time estimate calculations.	1000000
`retryCount`	Number of retries for a synchronizing handshake.	4
`address`	Address Segment	N/A
`traffic`	Traffic Segment	N/A
`diagCode`	DIAG packet code.	`XTPCONTEXTREFUSED`
`diagValue`	DIAG packet value.	`XTPUNSPECIFIEDVAL`

3.3 Configuration Structure

The configuration structure (type `XtpConfig`) provides the means to set the default configuration of the XTP engine from the application level. A reference to such a structure is passed as an argument to *XtpConfiguration()*. The structure is defined as follows:

```
typedef struct {
   int fieldFlags;            /* bitwise OR of selected fields */
   int numSysBufs;            /* XTPNUMSYSBUFS                 */
   int numRecvBufs;           /* XTPNUMRECVBUFS                */
   int numXbufs;              /* XTPNUMXBUFS                   */
   int recvBufSize;           /* XTPRECVBUFSIZE                */
   int timerInterval;         /* XTPTIMERINTERVAL              */
   int timerPriority;         /* XTPTIMERPRIORITY              */
   int senderPriority;        /* XTPSENDERPRIORITY             */
```

```
    int receiverPriority;       /* XTPRECEIVERPRIORITY          */
    int maxConnections;         /* XTPMAXCONNECTIONS            */
} XtpConfig;
```

The *fieldFlags* field is a bitmap of the parameters which the user wishes to select at runtime. The user should set this field as a bitwise OR of the appropriate constants (as listed in comments beside the fields), and set the actual fields to their desired values (each constant corresponds to one of the fields of the service structure). Any parameters (fields) not specified by *fieldFlags* will take on default values.

The priority parameters (*timerPriority*, *senderPriority* and *receiverPriority*) are intended for XTP debugging purposes and it is highly recommended that they not be modified by applications.

The following table gives a description of each field of the `XtpConfig` structure, as well as its default value:

Table 2: Configuration Fields

Field Name	Description	Default Value
`numSysBufs`	Number of receive buffers reserved for the system (control packets).	10
`numRecvBufs`	Number of receive buffers allocated for application data.	500
`numXbufs`	Number XTP internal buffer descriptors allocated (used for both sending and receiving).	2000
`recvBufSize`	Size of XTP receive buffers (this value limits the useful value of *maxdata*).	8968
`timerInterval`	Resolution of XTP timer, in microseconds.	50000
`timerPriority`	RT Threads priority of XTP timer thread.	0
`senderPriority`	RT Threads priority of XTP sender thread.	5
`receiverPriority`	RT Threads priority of XTP receiver thread.	4
`maxConnections`	Maximum number of simultaneous XTP connections	1024

4 Programming Guide

This section gives a brief description how to use the XTP Application Programming Interface in writing a program which uses XTP to transmit data between address spaces on the same or on different machines. Refer to section 3 for details (parameters, return values, etc.) on the routines mentioned in this section.

4.1 XTP Initialization

Using this implementation of XTP requires starting up an *XTP engine*, which is done by calling *XtpInit()*. One XTP engine runs per address space (i.e., RT Threads environment), thus *XtpInit()* should be called only once by an application. Alternately, *RttNetInit()* (see [6] for details) can be used to initialize XTP, since it makes a call to *XtpInit()*. *XtpInit()* allocates the resources and creates the threads used by XTP. Its other important function is to establish the engine as an endpoint for communication by allowing the user to specify the IP address and port number by which other address spaces can locate the local XTP engine. If 0 is specifed for the IP address, XTP will determine the IP address to use, although if the local machine has more than one IP address, it is not determined which will be chosen. The user can find out the IP address chosen using the *XtpMyIP()* routine. The port number specified must be unique among XTP engines using a given IP address. If 0 is specified, XTP will choose a unique port number. The chosen port number can be determined using the *XtpMyPort()* routine.

A number of default values are used by XTP for resource allocation and for initializing the parameters used when creating contexts. These defaults can be read and/or modified by the application using the *XtpConfiguration()* routine. The default values can only be modified *before* the call to *XtpInit()* (or *RttNetInit()*).

The following example shows how the above routines might be used to initialize XTP:

```
#define SERVER_PORT 6789
{
   XtpConfig config;
   u_int ipAddr, portNo;

   config.recvBufSize = 8000;
   config.fieldFlags = XTPRECVBUFSIZE;
   if (XtpConfiguration(XTPSET, &config, XTPNULL) == XTPFAILED) {
      printf("XtpConfigure() failed.\n");
   }

   if (XtpInit(0, SERVER_PORT) == XTPFAILED) {
      printf("RttNetInit() failed.\n");
      exit(0);
   }

   XtpMyIP(&ipAddr);
   XtpMyPort(&portNo);
   printf("IP address = %u portNo = %u\n, ipAddr, portNo);
}
```

4.2 Contexts

A context maintains the state for an endpoint of communication by maintaining various *context variables*. Before any XTP communication can take place at an endpoint, a context must be created. When a context is created, its variables are initialized according to default values. The values to which some of these variables, the *context parameters*, are initialized can also be set by the user. This can be done in two ways. The default values can be modified before XTP initialization using *XtpConfiguration()*. The default values can also be overridden by specifying different values using the `XtpService` structure at context creation time (i.e., the *svc* parameter to *XtpCreateContext()*).

See the example program in section 7 for an example of context creation.

4.3 Connection Establishment

There are a number of prerequisites for the establishment of a connection between two XTP endpoints. First, each endpoint application needs to create a context to maintain connection state for that endpoint. Then, one endpoint must listen for an incoming connection request, which it does by calling *XtpListen()*. Once one endpoint is listening, the other endpoint can establish a connection with it by calling *XtpConnect()*.

4.3.1 Listening For a Connection

The *localXtpPort* parameter for *XtpListen()* is used to specify a local XTP port number used to match incoming connection requests. More than one context may simultaneously listen on the same XTP port number. When a FIRST packet arrives at the local XTP engine, the port number specified in the packet will be compared to the port numbers for all listeners, until a match is found, at which point a connection will be established.

There are two possible modes for listening, one of which is to be specified using the *mode* parameter of *XtpListen()*. These modes are `XTPBLOCK` and `XTPNOBLOCK`, and are described as follows.

Using `XTPBLOCK` mode, the *XtpListen()* call will block until an incoming connection request has been accepted by the XTP engine. When *XtpListen()* returns, data may be read from and written to the connection.

Using `XTPNOBLOCK` mode, *XtpListen()* will return immediately, regardless of whether a connection has been established (it is very unlikely that a connection will have been established at the point when *XtpListen()* returns). A subsequent call to *XtpReadBuf()* or *XtpRead()* will block until a connection is established and there is data ready to be read. If *XtpWrite()* is called before a connection is established, it will return `XTPFAILED`.

See the example program in section 7 for an example of context creation.

4.3.2 Initiating a Connection

An application wishing to initiate a connection with another endpoint (which is presumed to be listening for the request) uses the *XtpConnect()* routine. The parameters *remoteIp*, *remotePort* and *remoteXtpPort* allow the request to locate the listening context at the other endpoint: *remoteIp* and *remotePort* specify the XTP engine at the other endpoint and *remoteXtpPort* specifies the port at which the remote context is listening. The request is transmitted to the other endpoint by XTP using a FIRST packet. There are two possible modes for *XtpConnect()*, as specified by the *mode* parameter. These modes are `XTPEXPLICIT` and `XTPIMPLICIT`, and are described as follows.

Using `XTPEXPLICIT` mode, *XtpConnect()* establishes a connection explicitly. In this case *XtpConnect*() will block until the connection is successfully established, in which case it returns `XTPOK`, or until the request fails (because either the remote XTP engine could not be located, there was no context listening on the given XTP port, or the request was refused by the listening context) in which cast *XtpConnect()* returns `XTPFAILED`.

Using `XTPIMPLICIT` mode, *XtpConnect()* establishes a connection implicitly. In this case *XtpConnect()* returns immediately, and the connection request (FIRST packet) is not sent by XTP until the first call to *XtpWrite()*. This allows more than one packet's worth of data to be sent before the connection is established. Since at this point (i.e., before the connection is established) the sender does not know the size of the receiver's flow control window (*alloc* value), the *swindow* service parameter (`XtpService` field) can be set at context creation time with an initial value to use for flow controlling *XtpWrite()*s until the connection is established. Should the connection request eventually fail, subsequent *XtpWrite()* calls will return `XTPFAILED` (previous *XtpWrite()* calls will have returned `XTPOK`, but the data from these calls will not have been successfully received).

If the connection request finds the target XTP engine, but fails because the request was refused, the target XTP engine will send a DIAG packet back to the local context. The initiating application can determine whether a DIAG packet was received, and if so determine the reason for refusal, using the *XtpDiagnostic()* routine. *XtpDiagnostic()* returns `XTPOK` if a DIAG packet was received, in which case its *code* and *value* parameters are set to the code and value of the DIAG packet. *XtpPrintDiagnostic()* is a convenience routine which will print the meaning of given *code* and *value* pairs. It is possible for a context to receive a DIAG packet at any time during the life of a connection, generally causing the connection to abort. *XtpDiagnostic()* can be called at any time to determine whether a DIAG packet was received (usually this would be done when an XTP API routine fails).

See the example program in section 7 for an example of context creation.

4.4 Data Transmission

Data transmission is accomplished using the *XtpWrite()* routine. Transmitted data can be read at the destination context using either *XtpReadBuf()* or *XtpRead()*.

4.4.1 Writing Data

The *XtpWrite()* routine provides a zero copy data transmission service. Since *XtpWrite()* does not block (except when flow controlled), and does not copy data, care must be taken not to free or corrupt the data before it has been successfully transmitted. It is for this reason that the *func* and *arg* parameters are provided, allowing a function to be passed in which will be called once the data has been successfully transmitted. The most common use for this function would be to pass in a routine to free the data, but other uses, such as synchronization, are possible.

If the data length (*len*) specified is greater than will fit in a single XTP packet, *XtpWrite()* will segment the data over several packets, and set the EOM option bit in the final packet, signifying end of message. Data from two successive calls to *XtpWrite()* will never be sent in the same packet. Some XTP option bits may be set by the application using the *options* parameter to *XtpWrite()*, in which case the specified option bits will be set in the last packet of the message (with the exception of BTAG, which will be set in the first packet of the message).

The most common option bits to set are WCLOSE (using `XTPWCLOSE`), END (using `XTPEND`) and BTAG (using `XTPBTAG`). `XTPWCLOSE` and `XTPEND` are used to indicate the final message is being sent by the context (depending on how the connection is to be closed). Subsequent calls to *XtpWrite()* will fail. `XTPBTAG` is used to indicate that the first eight bytes of data in the message are to be interpreted (by the receiving application) as out of band data. A common use for `XTPBTAG` is for encapsulation in higher level protocols.

4.4.2 Reading Data

XtpReadBuf() is the primary routine used for reading data from an XTP connection. *XtpReadBuf()* returns (by reference) the data from exactly one XTP packet. *XtpReadBuf()* blocks until at least one buffer has been received and is ready to be read. The buffer returned is allocated by the XTP engine and no copies are done. Data read with *XtpReadBuf()* must be freed using *XtpFreeBuf()*. Additional information about the data is obtained via the reference parameters *options* and *flags*. The *options* parameter returns the XTP option bits which were set in the received packet, and is most useful for determining whether the BTAG bit was set (in which case the `XTPBTAG` bit of *options* will be set). The *flags* parameter is used to determine the following information. The `XTPNULLDATA` bit is meaningful only in NOERR mode, and when set indicates that data was lost. When `XTPNULLDATA` is set, the value returned via the *bytes* parameter indicates the amount of missing data in the sequence space. Note that this may be more than one packet's worth of data. When `XTPNULLDATA` is set, *data* does not specify a valid buffer, and should not be freed. When the `XTPEOC` bit of *flags* is set, this indicates the end of context, meaning that there will be no more data to be read from the connection, and subsequent calls to *XtpReadBuf()* (and *XtpRead()*) will fail. Note that when `XTPEOC` is set, *bytes* may return a value of 0, indicating that the last data has already been read by a previous call, and in this case *data* will not be a valid buffer.

XtpRead() is provided as a convenience to give more traditional read semantics. XtpRead() is a blocking call which may read more than one packet's worth of data, up to *len* bytes, or up to the end of a packet with the EOM bit set (i.e., up to the end of a message). The data buffer into which the data is read must be provided by the application, and *XtpRead()* copies the received data into this buffer (*XtpRead()* frees the XTP buffers for the received packets). The `XTPNULL` flag is less useful than for *XtpReadBuf()*, since it is impossible to determine which data it is associated with.

4.5 Closing Procedures

The two data streams of a bi-directional XTP connection may be closed independently. When both streams are closed, the connection is closed. Data streams may be closed gracefully or abortively.

When a stream is closed gracefully, all data sent using *XtpWrite()* will be transmitted (reliably if not in NOERR mode) before the stream is closed. A graceful close in one direction can be accomplished in two ways: by setting the `XTPWCLOSE` bit in the final *XtpWrite()*, or by calling *XtpCloseContext()* after the final *XtpWrite()*. Using *XtpWrite()* with `XTPWCLOSE` can save the sending of an extra control packet by the XTP engine. *XtpCloseContext()* will block until the stream has gracefully closed. When XtpCloseContext() returns, the context will no longer be available and the associated context identifier will be invalid. Any received but unread data will be discarded.

A stream may be closed abortively in two ways: by setting the `XTPEND` bit in the final *XtpWrite()*, or by calling *XtpAbortContext()*. When *XtpWrite()* is used with `XTPEND`, the data for the *XtpWrite()* call and previous *XtpWrite()* calls will be transmitted, but after the last byte has been sent, no data will be retransmitted (so there is no guarantee of reliable transmission). When *XtpAbortContext()* is used to close a connection, the connection is aborted immediately and there is no guarantee that data from previous *XtpWrite()*s will have been transmitted. *XtpAbortContext()* will not block, and the context identifier is invalid upon return.

Unless explicitly closed with *XtpCloseContext()* or *XtpAbortContext()*, a context remains valid after a connection is closed (i.e., if *XtpWrite() options* bits are used to close the connection, or if the connection aborts spontaneously at the XTP level). This allows received data to be read using *XtpReadBuf()* or *XtpRead()* after the connection is closed, until `XTPEOC` is returned by one of these routines (via the *flags* parameter). At this point the context should be explicitly closed so as to free its resources.

When a connection is closed abortively, any functions passed in by the *func* parameter to *XtpWrite()* will be called regardless of whether the associated data was transmitted successfully.

4.6 XTP Modes

XTP modes (i.e., FASTNAK, NOFLOW, RES, NOERR, NOCHECK) for a given stream are selected by the sending context. The set of modes used by the two streams of a bi-directional XTP connection may be different. Modes are set only at context creation, and cannot change during the life of the context (attempts to change them using *XtpEntry()* will be ignored). Modes are selected by setting the *modes* field of the `XtpService` structure to a bitwise OR of mode flags chosen from `XTPFASTNAK`, `XTPNOLFLOW`, `XTPRES`, `XTPNOERR` and `XTPNOCHECK`. In addition, a simplex stream can be selected using `XTPRCLOSE` as a mode flag.

By default, XTP streams are flow controlled, error controlled and perform data checksums. If any of these features are not desired, they must be explicitly turned off using `XTPNOFLOW`, `XTPNOERR or XTPNOCHECK`.

The following brief code sample shows how modes are set for a context. The context created will be flow controlled, but will not be error controlled and will not perform checksums on data.

```
{
   XtpCtxtId ctxtId;
   XtpService svc;

   svc.modes = XTPNOERR | XTPNOCHECK;
   svc.fieldFlags = XTPMODES;
   XtpCreateContext(&ctxtId, &svc);
}
```

The following sections give some more details on using XTP modes.

4.6.1 FASTNAK Mode

FASTNAK mode is selected using the `XTPFASTNAK` flag. When a sending context has FASTNAK mode set, the receiving context will send an ECNTL packet immediately when missing data is detected, so that the data will be retransmitted as soon as possible. FASTNAK mode should only be used when the underlying network does not reorder packets excessively. FASTNAK mode has no effect when NOERR mode is set.

4.6.2 NOFLOW Mode

NOFLOW mode is selected using the `XTPNOFLOW` flag. In NOFLOW mode, flow control is turned off. Thus, *XtpWrite()* will never block when in NOFLOW mode.

4.6.3 RES Mode

RES (reservation) mode is selected by a sending context using the `XTPRES` flag. Reservation mode allows a receiving application to reserve buffer space for receiving data. When a sender specifies reserved mode, *alloc* values returned by the receiver will represent the amount of reserved buffer space available, thus the sender is prevented from overflowing the reserved buffer space. The receiving context specifies the size of the reserved buffer space at context creation using the *reserveSize* field of the `XtpService` structure. Reserved mode is intended for bulk data transfer, to avoid exhausting internal XTP buffer resources.

4.6.4 NOERR Mode

NOERR mode is selected using the `XTPNOERR` flag. When NOERR mode is used, lost or corrupted data is not retransmitted. When data is received out of order (i.e., later than data with higher sequence numbers), it is discarded (and considered lost). Gaps in the data are indicated to the receiving application via `the XTPNULLDATA` flag returned in the *flags* parameter of *XtpRead()*. The function *func* passed into *XtpWrite()* is called as soon as data is transmitted in NOERR mode.

4.6.5 NOCHECK Mode

NOCHECK mode is selected using the `XTPNOCHECK` flag. In NOCHECK mode, checksums are calculated only on the header fields of XTP packets, so user data is not protected from corruption.

4.6.6 Simplex Connections

A unidirectional connection originating at the connection initiator (i.e., the context calling XtpConnect()) can be selected using the `XTPRCLOSE` flag. Selecting `XTPRCLOSE` at the listening end of a connection will have no effect.

5 Addressing

XTP provides parameterized addressing, so it can use a number of different addressing schemes. This implementation currently uses only IP addressing (although it is open to the addition of more schemes). Thus, all addresses should be specified as IP addresses. The following convenience routines are provided to assist in determining IP addresses:

(1) *u_int XtpGetLocalIP()* this routine will return the IP address of the local host. If the local host has more than one IP address, it is not defined which one will be returned.

(2) *u_int XtpHostnameToIP(char *hostname)* This routine will return the IP address of the given host. If the specified host has more than one IP address, it is not defined which one will be returned.

6 Packet Encapsulation

XTP packets are encapsulated in lower level packets, depending on the underlying communication mechanism. In the UNIX/RT Threads environment, XTP packets are encapsulated in UDP packets (and transmitted using UDP/IP).

Details of encapsulation are hidden from the user, so encapsulation in other lower level protocols (e.g., AAL5/ATM) can be supported without affecting applications.

7 Example

Two short example programs (a sender and a receiver) follow to demonstrate how the service interface to XTP is used. Some error checking is omitted for simplicity.

7.1 Sender Program

```
#include "rtthreads.h"
#include "xtp.h"

#define RCVR_UDP_PORT 5000
#define XTP_PORT 25

static void sender(void *);

void mainp(int argc, char *argv[])
{
   int remoteIp;
   RttSchAttr attr;
   RttThreadId sndr;

   if (argc != 2) {
      printf("usage: %s remote_hostname\n", argv[0]);
      exit(1);
   }

   if ((remoteIp = XtpHostnameToIP(argv[1])) == XTPFAILED){
      printf("host '%s' not found.\n", argv[1]);
      exit(0);
   }

   XtpInit(0, 0);

   /* create the sender thread, schedule it to be ready immediately */
   attr.startingtime = RTTZEROTIME;
   attr.priority = RTTNORM;
   attr.deadline = RTTNODEADLINE;
   RttCreate(&sndr, sender, 8192, "sender", (void *) remoteIp, attr, RTTUSR);
}

static void sender(void *remoteIp) {
   XtpCtxtId ctxtId;
   int flags, options;
   int bytes = 0;
   static char buf[100];
   u_int route;

   if (XtpCreateContext(&ctxtId, XTPNULL) == XTPFAILED) {
      printf("unable to create context\n");
      exit(1);
   }

   /* explicitly open a connection (block until established) */
   XtpConnect(ctxtId, XTPEXPLICIT, (u_int)remoteIp, RCVR_UDP_PORT, XTP_PORT);

   /* send a message to the receiver */
   XtpWrite(ctxtId, "How are you?", 13, XTPWCLOSE, XTPNULL, 0);

   /* wait for reply from receiver */
   XtpRead(ctxtId, buf, 100, &bytes, &flags, &options);
```

```
   printf("Reply: %s\n", buf);

   XtpCloseContext(ctxtId);
}
```

7.2 Receiver Program

```
#include "rtthreads.h"
#include "xtp.h"

#define RCVR_UDP_PORT 5000
#define XTP_PORT 25

static void receiver(void *);

void mainp( int argc, char *argv[] )
{
   RttSchAttr attr;
   RttThreadId rcvr;

   XtpInit(0, RCVR_UDP_PORT);

   /* create the receiver thread, schedule it to be ready immediately */
   attr.startingtime = RTTZEROTIME;
   attr.priority = RTTNORM;
   attr.deadline = RTTNODEADLINE;
   RttCreate(&rcvr, receiver, 8192, "receiver", (void *) 0, attr, RTTUSR);
}

static void receiver(void *arg) {
   XtpCtxtId ctxtId;
   XtpService svc;
   int flags, options;
   int bytes = 0;
   static char buf[100];

   /* assign service structure field values */
   svc.rwindow = 20000;
   svc.inrate = 300000;
   svc.inburst = 100000;
   svc.fieldFlags = XTPRWINDOW | XTPINRATE | XTPINBURST;

   if (XtpCreateContext(&ctxtId, &svc) == XTPFAILED) {
      printf("unable to create context\n");
      exit(1);
   }

   /* listen for a connection request (block until established) */
   XtpListen(ctxtId, XTPBLOCK, XTP_PORT);

   /* read up to 100 bytes */
   XtpRead(ctxtId, buf, 100, &bytes, &flags, &options);

   printf("Message: %s\n", buf);

   /* let sender know I'm done */
   XtpWrite(ctxtId, "Just fine.", 11, XTPWCLOSE, XTPNULL, 0);

   XtpCloseContext(ctxtId);
}
```

References

[1] Greg Chesson. XTP/PE Design Considerations. In *IFIP WG6.1/6.4 Workshop on Protocols for High-Speed Networks*, May 1989

[2] Greg Chesson. The Evolution of XTP. In *Proceedings of the Third International Conference on High Speed networking*. North-Holland, 1991.

[3] Protocol Engines Incorporated. XTP Protocol Definition, Revision 3.6, January, 1992.

[4] XTP Forum. Xpress Transport Protocol Specification, Revision 4.0, March, 1995.

[5] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. XTP: The Xpress Transfer Protocol. Addison-Wesley, Reading, Massachusetts, 1992.

[6] David Finkelstein, Norman C. Hutchinson, Dwight J. Makaroff, Roland Mechler and Gerald W. Neufeld. Real Time Threads Interface. UBC Technical Report 95-07, May 1995.