                   Transaction Internet Protocol
                          Version 3.0

Status of this Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements. Please refer to the current edition of the "Internet
   Official Protocol Standards" (STD 1) for the standardization state
   and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

   In many applications where different nodes cooperate on some work,
   there is a need to guarantee that the work happens atomically. That
   is, each node must reach the same conclusion as to whether the work
   is to be completed, even in the face of failures. This document
   proposes a simple, easily-implemented protocol for achieving this
   end.

Table of Contents

1. Introduction

   The standard method for achieving atomic commitment is the two-phase
   commit protocol; see [1] for an introduction to atomic commitment and
   two-phase commit protocols.

   Numerous two-phase commit protocols have been implemented over the
   years. However, none of them has become widely used in the Internet,
   due mainly to their complexity. Most of that complexity comes from
   the fact that the two-phase commit protocol is bundled together with
   a specific program-to-program communication protocol, and that
   protocol lives on top of a very large infrastructure.

   This memo proposes a very simple two-phase commit protocol. It
   achieves its simplicity by specifying only how different nodes agree
   on the outcome of a transaction; it allows (even requires) that the
   subject matter on which the nodes are agreeing be communicated via
   other protocols. By doing so, we avoid all of the issues related to
   application communication semantics and data representation (to name
   just a few). Independent of the application communication protocol a
   transaction manager may use the Transport Layer Security protocol [3]
   to authenticate other transaction managers and encrypt messages.

   It is envisioned that this protocol will be used mainly for a
   transaction manager on one Internet node to communicate with a
   transaction manager on another node. While it is possible to use this
   protocol for application programs and/or resource managers to speak
   to transaction managers, this communication is usually intra-node,
   and most transaction managers already have more-than-adequate
   interfaces for the task.

   While we do not expect this protocol to replace existing ones, we do
   expect that it will be relatively easy for many existing
   heterogeneous transaction managers to implement this protocol for
   communication with each other.

   Further supplemental information regarding the TIP protocol can be
   found in [5].

2. Example Usage

   Today the electronic shopping basket is a common metaphor at many
   electronic store-fronts. Customers browse through an electronic
   catalog, select goods and place them into an electronic shopping
   basket. HTTP servers [2] provide various means ranging from URL
   encoding to context cookies to keep track of client context (e.g.
   the shopping basket of a customer) and resume it on subsequent
   customer requests.

   Once a customer has finished shopping they may decide to commit their
   selection and place the associated orders. Most orders may have no
   relationship with each other except being executed as part of the
   same shopping transaction; others may be dependent on each other (for
   example, if made as part of a special offering). Irrespective of
   these details a customer will expect that all orders have been
   successfully placed upon receipt of a positive acknowledgment.
   Today's electronic store-fronts must implement their own special
   protocols to coordinate such placement of all orders. This
   programming is especially complex when orders are placed through
   multiple electronic store-fronts. This complexity limits the
   potential utility of internet applications, and constrains growth.
   The protocol described in this document intends to provide a standard
   for Internet servers to achieve agreement on a unit of shared work
   (e.g. placement of orders in an electronic shopping basket). The
   server (e.g. a CGI program) placing the orders may want to start a
   transaction calling its local transaction manager, and ask other
   servers participating in the work to join the transaction. The
   server placing the orders passes a reference to the transaction as
   user data on HTTP requests to the other servers. The other servers
   call their transaction managers to start a local transaction and ask
   them to join the remote transaction using the protocol defined in
   this document. Once all orders have been placed, execution of the
   two-phase-commit protocol is delegated to the involved transaction
   managers. If the transaction commits, all orders have been
   successfully placed and the customer gets a positive acknowledgment.
   If the transaction aborts no orders will be placed and the customer
   will be informed of the problem.

   Transaction support greatly simplifies programming of these
   applications as exception handling and failure recovery are delegated
   to a special component. End users are also not left having to deal
   with the consequences of only partial success. While this example
   shows how the protocol can be used by HTTP servers, applications may
   use the protocol when accessing a remote database (e.g. via ODBC), or
   invoking remote services using other already existing protocols (e.g.

   RPC). The protocol makes it easy for applications in a heterogeneous
   network to participate in the same transaction, even if using
   different communication protocols.

3. Transactions

   "Transaction" is the term given to the programming model whereby
   computational work performed has atomic semantics. That is, either
   all work completes successfully and changes are made permanent (the
   transaction commits), or if any work is unsuccessful, changes are
   undone (the transaction aborts). The work comprising a transaction
   (unit of work), is defined by the application.

4. Connections

   The Transaction Internet Protocol (TIP) requires a reliable ordered
   stream transport with low connection setup costs. In an Internet (IP)
   environment, TIP operates over TCP, optionally using TLS to provide a
   secured and authenticated connection, and optionally using a protocol
   to multiplex light-weight connections over the same TCP or TLS
   connection.

   Transaction managers that share transactions establish a TCP (and
   optionally a TLS) connection. The protocol uses a different
   connection for each simultaneous transaction shared betwween two
   transaction managers. After a transaction has ended, the connection
   can be reused for a different transaction.

   Optionally, instead of associating a TCP or TLS connection with only
   a single transaction, two transaction managers may agree on a
   protocol to multiplex light-weight connections over the same TCP or
   TLS connection, and associate each simultaneous transaction with a
   separate light-weight connection. Using light-weight connections
   reduces latency and resource consumption associated with executing
   simultaneous transactions. Similar techniques as described here are
   widely used by existing transaction processing systems. See Appendix
   A for an example of one such protocol.

   Note that although the TIP protocol itself is described only in terms
   of TCP and TLS, there is nothing to preclude the use of TIP with
   other transport protocols. However, it is up to the implementor to
   ensure the chosen transport provides equivalent semantics to TCP, and
   to map the TIP protocol appropriately.

In this document the terms "connection" or "TCP connection" can refer
to a TIP TCP connection, a TIP TLS connection, or a TIP multiplexing
connection (over either TCP or TLS). It makes no difference which,
the behavior is the same in each case. Where there are differences in
behavior between the connection types, these are stated explicitly.

5. Transaction Identifiers

Unfortunately, there is no single globally-accepted standard for the
format of a transaction identifier; there are various standard and
proprietary formats. Allowed formats for a TIP transaction
identifier are described below in the section "TIP Uniform Resource
Locators". A transaction manager may map its internal transaction
identifiers into this TIP format in any manner it sees fit.
Furthermore, each party in a superior/subordinate relationship gets
to assign its own identifier to the transaction; these identifiers
are exchanged when the relationship is first established. Thus, a
transaction manager gets to use its own format of transaction
identifier internally, but it must remember a foreign transaction
identifier for each superior/subordinate relationship in which it is
involved.

6. Pushing vs. Pulling Transactions

Suppose that some program on node "A" has created a transaction, and
wants some program on node "B" to do some work as part of the
transaction. There are two classical ways that he does this,
referred to as the "push" model and the "pull" model.

In the "push" model, the program on A first asks his transaction
manager to export the transaction to node B. A's transaction manager
sends a message to B's TM asking it to instantiate the transaction as
a subordinate of A, and return its name for the transaction. The
program on A then sends a message to its counterpart on B on the
order of "Do some work, and make it part of the transaction that your
transaction manager already knows of by the name ...". Because A's
TM knows that it sent the transaction to B's TM, A's TM knows to
involve B's TM in the two-phase commit process.

In the "pull" model, the program on A merely sends a message to B on
the order of "Do some work, and make it part of the transaction that
my TM knows by the name ...". The program on B asks its TM to enlist
in the transaction. At that time, B's TM will "pull" the transaction
over from A. As a result of this pull, A's TM knows to involve B's
TM in the two-phase commit process.

The protocol described here supports both the "push" and "pull"
models.

7. TIP Transaction Manager Identification and Connection Establishment

   In order for TIP transaction managers to connect they must be able to
   identify and locate each other. The information necessary to do this
   is described by the TIP transaction manager address.

   [This specification does not prescribe how TIP transaction managers
   initially obtain the transaction manager address (which will probably
   be via some implementation-specific configuration mechanism).]

   TIP transaction manager addresses take the form:

     <hostport><path>

   The <hostport> component comprises:

     <host>[:<port>]

   where <host> is either a <dns name> or an <ip address>; and <port> is
   a decimal number specifying the port at which the transaction manager
   (or proxy) is listening for requests to establish TIP connections. If
   the port number is omitted, the standard TIP port number (3372) is
   used.

   A <dns name> is a standard name, acceptable to the domain name
   service. It must be sufficiently qualified to be useful to the
   receiver of the command.

   An <ip address> is an IP address, in the usual form: four decimal
   numbers separated by period characters.

   The <hostport> component defines the scope (locale) of the <path>
   component.

   The <path> component of the transaction manager address contains data
   identifying the specific TIP transaction manager, at the location
   defined by <hostport>.

   The <path> component takes the form:

     "/" [path_segments]

     path_segments = segment *("/" segment)
     segment = *pchar *(";" param)
     param = *pchar

     pchar = unreserved | escaped | ":" | "@" | "&" | "=" | "+"
     unreserved = ASCII character octets with values in the range

```
                (inclusive): 48-57, 65-90, 97-122 | "$" | "-" | "_" |
                ". " | "!" | "~" | "*" | "'" | "(" | ")" | ","
  escaped = "%" hex hex
  hex = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
        "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" |
        "e" | "f"
```

The <path> component may consist of a sequence of path segments
separated by a single slash "/" character. Within a path segment, the
characters "/", ";", "=", and "?" are reserved. Each path segment may
include a sequence of parameters, indicated by the semicolon ";"
character. The parameters are not significant to the parsing of
relative references.

[It is intended that the form of the transaction manager address
follow the proposed scheme for Uniform Resource Identifiers (URI)
[8].]

The TIP transaction manager address therefore provides to the
connection initiator (the primary) the endpoint identifier to be used
for the TCP connection (<hostport>), and to the connection receiver
(the secondary) the path to be used to locate the specific TIP
transaction manager (<path>). This is all the information required
for the connection between the primary and secondary TIP transaction
managers to be established.

After a connection has been established, the primary party issues an
IDENTIFY command. This command includes as parameters two transaction
manager addresses: the primary transaction manager address, and the
secondary transaction manager address.

The primary transaction manager address identifies the TIP
transaction manager that initiated the connection. This information
is required in certain cases after connection failures, when one of
the parties of the connection must re-establish a new connection to
the other party in order to complete the two-phase-commit protocol.
If the primary party needs to re-establish the connection, the job is
easy: a connection is established in the same way as was the original
connection. However, if the secondary party needs to re-establish the
connection, it must be known how to contact the initiator of the
original connection. This information is supplied to the secondary
via the primary transaction manager address on the IDENTIFY command.
If a primary transaction manager address is not supplied, the primary
party must not perform any action which would require a connection to
be re-established (e.g. to perform recovery actions).

The secondary transaction manager address identifies the receiving
TIP transaction manager. In the case of TIP communication via
intermediate proxy servers, this URL may be used by the proxy servers
to correctly identify and connect to the required TIP transaction
manager.

8. TIP Uniform Resource Locators

Transactions and transaction managers are resources associated with
the TIP protocol. Transaction managers and transactions are located
using the transaction manager address scheme. Once a connection has
been established, TIP commands may be sent to operate on transactions
associated with the respective transaction managers.

Applications which want to pull a transaction from a remote node must
supply a reference to the remote transaction which allows the local
transaction manager (i.e. the transaction manager pulling the
transaction) to connect to the remote transaction manager and
identify the particular transaction. Applications which want to push
a transaction to a remote node must supply a reference to the remote
transaction manager (i.e. the transaction manager to which the
transaction is to be pushed), which allows the local transaction
manager to locate the remote transaction manager. The TIP protocol
defines a URL scheme [4] which allows applications and transaction
managers to exchange references to transaction managers and
transactions.

A TIP URL takes the form:

   tip://<transaction manager address>?<transaction string>

where <transaction manager address> identifies the TIP transaction
manager (as defined in Section 7 above); and <transaction string>
specifies a transaction identifier, which may take one of two forms
(standard or non-standard):

i. "urn:" <NID> ":" <NSS>

   A standard transaction identifier, conforming to the proposed
   Internet Standard for Uniform Resource Names (URNs), as specified
   by RFC2141; where <NID> is the Namespace Identifier, and <NSS> is
   the Namespace Specific String. The Namespace ID determines the
   syntactic interpretation of the Namespace Specific String. The
   Namespace Specific String is a sequence of characters representing
   a transaction identifier (as defined by <NID>). The rules for the
   contents of these fields are specified by [6] (valid characters,
   encoding, etc.).

This format of <transaction string> may be used to express global
transaction identifiers in terms of standard representations.
Examples for <NID> might be <iso> or <xopen>. e.g.

   tip://123.123.123.123/?urn:xopen:xid

Note that Namespace Ids require registration. See [7] for details
on how to do this.

  ii. <transaction identifier>

A sequence of printable ASCII characters (octets with values in the
range 32 through 126 inclusive (excluding ":") representing a
transaction identifier. In this non-standard case, it is the
combination of <transaction manager address> and <transaction
identifier> which ensures global uniqueness. e.g.

   tip://123.123.123.123/?transid1

To create a non-standard TIP URL from a transaction identifier,
first replace any reserved characters in the transaction identifier
with their equivalent escape sequences, then insert the appropriate
transaction manager address. If the transaction identifier is one
that you created, insert your own transaction manager address. If
the transaction identifier is one that you received on a TIP
connection that you initiated, use the secondary transaction
manager address that was sent in the IDENTIFY command. If the
transaction identifier is one that you received on a TIP connection
that you did not initiate, use the primary transaction manager
address that was received in the IDENTIFY command.

TIP URLs must be guaranteed globally unique for all time. This
uniqueness constraint ensures TIP URLs are never duplicated, thereby
preventing possible non-deterministic behaviour. How uniqueness is
achieved is implementation specific. For example, the Universally
Unique Identifier (UUID, also known as a Globally Unique Identifier,
or GUID (see [9])) could be used as part of the <transaction string>.
Note also that some standard transaction identifiers may define their
own rules for ensuring global uniqueness (e.g. OSI CCR atomic action
identifiers).

Except as otherwise described above, the TIP URL scheme follows the
rules for reserved characters as defined in [4], and uses escape
sequences as defined in [4] Section 5.

Note that the TIP protocol itself does not use the TIP URL scheme (it
does use the transaction manager address scheme). The TIP URL scheme
is proposed as a standard way to pass transaction identification

information through other protocols. e.g. between cooperating
application processes. The TIP URL may then be used to communicate to
the local transaction manager the information necessary to associate
the application with a particular TIP transaction. e.g. to PULL the
transaction from a remote transaction manager. It is anticipated that
each TIP implementation will provide some set of APIs for this
purpose ([5] includes examples of such APIs).

9. States of a Connection

At any instant, only one party on a connection is allowed to send
commands, while the other party is only allowed to respond to
commands that he receives. Throughout this document, the party that
is allowed to send commands is called "primary"; the other party is
called "secondary". Initially, the party that initiated the
connection is primary; however, a few commands cause the roles to
switch. A connection returns to its original polarity whenever the
Idle state is reached.

When multiplexing is being used, these rules apply independently to
each "virtual" connection, regardless of the polarity of the
underlying connection (which will be in the Multiplexing state).

Note that commands may be sent "out of band" by the secondary via the
use of pipelining. This does not affect the polarity of the
connection (i.e. the roles of primary and secondary do not switch).
See section 12 for details.

In the normal case, TIP connections should only be closed by the
primary, when in Initial state. It is generally undesirable for a
connection to be closed by the secondary, although this may be
necessary in certain error cases.

At any instant, a connection is in one of the following states. From
the point of view of the secondary party, the state changes when he
sends a reply; from the point of view of the primary party, the state
changes when he receives a reply.

Initial: The initial connection starts out in the Initial state.
   Upon entry into this state, the party that initiated the connection
   becomes primary, and the other party becomes secondary. There is no
   transaction associated with the connection in this state. From this
   state, the primary can send an IDENTIFY or a TLS command.

Idle: In this state, the primary and the secondary have agreed on a
   protocol version, and the primary supplied an identifier to the
   secondary party to reconnect after a failure. There is no
   transaction associated with the connection in this state. Upon

entry to this state, the party that initiated the connection
becomes primary, and the other party becomes secondary. From this
state, the primary can send any of the following commands: BEGIN,
MULTIPLEX, PUSH, PULL, QUERY and RECONNECT.

Begun: In this state, a connection is associated with an active
transaction, which can only be completed by a one-phase protocol.
A BEGUN response to a BEGIN command places a connection into this
state. Failure of a connection in Begun state implies that the
transaction will be aborted. From this state, the primary can send
an ABORT, or COMMIT command.

Enlisted: In this state, the connection is associated with an active
transaction, which can be completed by a one-phase or, two-phase
protocol. A PUSHED response to a PUSH command, or a PULLED response
to a PULL command, places the connection into this state. Failure
of the connection in Enlisted state implies that the transaction
will be aborted. From this state, the primary can send an ABORT,
COMMIT, or PREPARE command.

Prepared: In this state, a connection is associated with a
transaction that has been prepared. A PREPARED response to a
PREPARE command, or a RECONNECTED response to a RECONNECT command
places a connection into this state. Unlike other states, failure
of a connection in this state does not cause the transaction to
automatically abort. From this state, the primary can send an
ABORT, or COMMIT command.

Multiplexing: In this state, the connection is being used by a
multiplexing protocol, which provides its own set of connections.
In this state, no TIP commands are possible on the connection. (Of
course, TIP commands are possible on the connections supplied by
the multiplexing protocol.) The connection can never leave this
state.

Tls: In this state, the connection is being used by the TLS
protocol, which provides its own secured connection. In this state,
no TIP commands are possible on the connection. (Of course, TIP
commands are possible on the connection supplied by the TLS
protocol.) The connection can never leave this state.

Error: In this state, a protocol error has occurred, and the
connection is no longer useful. The connection can never leave this
state.

10. Protocol Versioning

   This document describes version 3 of the protocol. In order to
   accommodate future versions, the primary party sends a message
   indicating the lowest and the highest version number it understands.
   The secondary responds with the highest version number it
   understands.

   After such an exchange, communication can occur using the smaller of
   the highest version numbers (i.e., the highest version number that
   both understand). This exchange is mandatory and occurs using the
   IDENTIFY command (and IDENTIFIED response).

   If the highest version supported by one party is considered obsolete
   and no longer supported by the other party, no useful communication
   can occur. In this case, the newer party should merely drop the
   connection.

11. Commands and Responses

   All commands and responses consist of one line of ASCII text, using
   only octets with values in the range 32 through 126 inclusive,
   followed by either a CR (an octet with value 13) or an LR (an octet
   with value 10). Each line can be split up into one or more "words",
   where successive words are separated by one or more space octets
   (value 32).

   Arbitrary numbers of spaces at the beginning and/or end of each line
   are allowed, and ignored.

   Lines that are empty, or consist entirely of spaces are ignored.
   (One implication of this is that you can terminate lines with both a
   CR and an LF if desired; the LF will be treated as terminating an
   empty line, and ignored.)

   In all cases, the first word of each line indicates the type of
   command or response; all defined commands and responses consist of
   upper-case letters only.

   For some commands and responses, subsequent words convey parameters
   for the command or response; each command and response takes a fixed
   number of parameters.

   All words on a command or response line after (and including) the
   first undefined word are totally ignored. These can be used to pass
   human-readable information for debugging or other purposes.

12. Command Pipelining

   In order to reduce communication latency and improve efficiency, it
   is possible for multiple TIP "lines" (commands or responses) to be
   pipelined (concatenated) together and sent as a single message.
   Lines may also be sent "ahead" (by the secondary, for later procesing
   by the primary). Examples are an ABORT command immediately followed
   by a BEGIN command, or a COMMITTED response immediately followed by a
   PULL command.

   The sending of pipelined lines is an implementation option. Likewise
   which lines are pipelined together. Generally, it must be certain
   that the pipelined line will be valid for the state of the connection
   at the time it is processed by the receiver. It is the responsibility
   of the sender to determine this.

   All implementations must support the receipt of pipelined lines - the
   rules for processing of which are described by the following
   paragraph:

      When the connection state is such that a line should be read
      (either command or response), then that line (when received) is
      processed. No more lines are read from the connection until
      processing again reaches such a state. If a line is received on a
      connection when it is not the turn of the other party to send, that
      line is _not_ rejected. Instead, the line is held and processed
      when the connection state again requires it. The receiving party
      must process lines and issue responses in the order of lines
      received. If a line causes an error the connection enters the Error
      state, and all subsequent lines on the connection are discarded.

13. TIP Commands

   Commands pertain either to connections or transactions. Commands
   which pertain to connections are: IDENTIFY, MULTIPLEX and TLS.
   Commands which pertain to transactions are: ABORT, BEGIN, COMMIT,
   PREPARE, PULL, PUSH, QUERY, and RECONNECT.

   Following is a list of all valid commands, and all possible responses
   to each:

   ABORT

      This command is valid in the Begun, Enlisted, and Prepared states.
      It informs the secondary that the current transaction of the
      connection will abort. Possible responses are:

ABORTED
  The transaction has aborted; the connection enters Idle state.

ERROR
  The command was issued in the wrong state, or was malformed. The
  connection enters the Error state.

BEGIN

  This command is valid only in the Idle state. It asks the secondary
  to create a new transaction and associate it with the connection.
  The newly created transaction will be completed with a one-phase
  protocol. Possible responses are:

  BEGUN <transaction identifier>
    A new transaction has been successfully begun, and that
    transaction is now the current transaction of the connection.
    The connection enters Begun state.

  NOTBEGUN
    A new transaction could not be begun; the connection remains in
    Idle state.

  ERROR
    The command was issued in the wrong state, or was malformed. The
    connection enters the Error state.

COMMIT

  This command is valid in the Begun, Enlisted or Prepared states.
  In the Begun or Enlisted state, it asks the secondary to attempt to
  commit the transaction; in the Prepared state, it informs the
  secondary that the transaction has committed. Note that in the
  Enlisted state this command represents a one-phase protocol, and
  should only be done when the sender has 1) no local recoverable
  resources involved in the transaction, and 2) only one subordinate
  (the sender will not be involved in any transaction recovery
  process). Possible responses are:

  ABORTED
    This response is possible only from the Begun and Enlisted
    states. It indicates that some party has vetoed the commitment of
    the transaction, so it has been aborted instead of committing.
    The connection enters the Idle state.

COMMITTED
  This response indicates that the transaction has been committed,
  and that the primary no longer has any responsibilities to the
  secondary with respect to the transaction. The connection enters
  the Idle state.

ERROR
  The command was issued in the wrong state, or was malformed. The
  connection enters the Error state.

ERROR

  This command is valid in any state; it informs the secondary that a
  previous response was not recognized or was badly formed. A
  secondary should not respond to this command. The connection enters
  Error state.

IDENTIFY <lowest protocol version>
          <highest protocol version>
          <primary transaction manager address> | "-"
          <secondary transaction manager address>

  This command is valid only in the Initial state. The primary party
  informs the secondary party of: 1) the lowest and highest protocol
  version supported (all versions between the lowest and highest must
  be supported; 2) optionally, an identifier for the primary party at
  which the secondary party can re-establish a connection if ever
  needed (the primary transaction manager address); and 3) an
  identifier which may be used by intermediate proxy servers to
  connect to the required TIP transaction manager (the secondary
  transaction manager address). If a primary transaction manager
  address is not supplied, the secondary party will respond with
  ABORTED or READONLY to any PREPARE commands. Possible responses
  are:

  IDENTIFIED <protocol version>
    The secondary party has been successfully contacted and has saved
    the primary transaction manager address. The response contains
    the highest protocol version supported by the secondary party.
    All future communication is assumed to take place using the
    smaller of the protocol versions in the IDENTIFY command and the
    IDENTIFIED response. The connection enters the Idle state.

  NEEDTLS
    The secondary party is only willing to communicate over TLS
    secured connections. The connection enters the Tls state, and all
    subsequent communication is as defined by the TLS protocol. This
    protocol will begin with the first octet after the line

terminator of the IDENTIFY command (for data sent by the primary
party), and the first byte after the line terminator of the
NEEDTLS response (for data sent by the secondary party). This
implies that an implementation must not send both a CR and a LF
octet after either the IDENTIFY command or the NEEDTLS response,
lest the LF octet be mistaken for the first byte of the TLS
protocol. The connection provided by the TLS protocol starts out
in the Initial state. After TLS has been negotiated, the primary
party must resend the IDENTIFY command. If the primary party
cannot support (or refuses to use) the TLS protocol, it closes
the connection.

   ERROR
      The command was issued in the wrong state, or was malformed.
      This response also occurs if the secondary party does not support
      any version of the protocol in the range supported by the primary
      party. The connection enters the Error state. The primary party
      should close the connection.

   MULTIPLEX <protocol-identifier>

      This command is only valid in the Idle state. The command seeks
      agreement to use the connection for a multiplexing protocol that
      will supply a large number of connections on the existing
      connection. The primary suggests a particular multiplexing
      protocol. The secondary party can either accept or reject use of
      this protocol.

      At the present, the only defined protocol identifier is "TMP2.0",
      which refers to the TIP Multiplexing Protocol, version 2.0. See
      Appendix A for details of this protocol. Other protocol identifiers
      may be defined in the future.

      If the MULTIPLEX command is accepted, the specified multiplexing
      protocol will totally control the underlying connection. This
      protocol will begin with the first octet after the line terminator
      of the MULTIPLEX command (for data sent by the initiator), and the
      first byte after the line terminator of the MULTIPLEXING response
      (for data received by the initiator). This implies that an
      implementation must not send both a CR and a LF octet after either
      the MULTIPLEX command or the MULTIPLEXING response, lest the LF
      octet be mistaken for the first byte of the multiplexing protocol.

      Note that when using TMP V2.0, a single TIP command (TMP
      application message) must be wholly contained within a single TMP
      packet (the TMP PUSH flag is not used by TIP). Possible responses
      to the MULTIPLEX command are:

MULTIPLEXING
  The secondary party agrees to use the specified multiplexing
  protocol. The connection enters the Multiplexing state, and all
  subsequent communication is as defined by that protocol. All
  connections created by the multiplexing protocol start out in the
  Idle state.

CANTMULTIPLEX
  The secondary party cannot support (or refuses to use) the
  specified multiplexing protocol. The connection remains in the
  Idle state.

ERROR
  The command was issued in the wrong state, or was malformed. The
  connection enters the Error state.

PREPARE

  This command is valid only in the Enlisted state; it requests the
  secondary to prepare the transaction for commitment (phase one of
  two-phase commit). Possible responses are:

  PREPARED
    The subordinate has prepared the transaction; the connection
    enters PREPARED state.

  ABORTED
    The subordinate has vetoed committing the transaction. The
    connection enters the Idle state. After this response, the
    superior has no responsibilities to the subordinate with respect
    to the transaction.

  READONLY
    The subordinate no longer cares whether the transaction commits
    or aborts. The connection enters the Idle state. After this
    response, the superior has no responsibilities to the subordinate
    with respect to the transaction.

  ERROR
    The command was issued in the wrong state, or was malformed. The
    connection enters the Error state.

PULL   <superior's transaction identifier>
       <subordinate's transaction identifier>

  This command is only valid in Idle state. This command seeks to
  establish a superior/subordinate relationship in a transaction,
  with the primary party of the connection as the subordinate (i.e.,

he is pulling a transaction from the secondary party). Note that
the entire value of <transaction string> (as defined in the section
"TIP Uniform Resource Locators") must be specified as the
transaction identifier. Possible responses are:

> PULLED
>   The relationship has been established. Upon receipt of this
>   response, the specified transaction becomes the current
>   transaction of the connection, and the connection enters Enlisted
>   state. Additionally, the roles of primary and secondary become
>   reversed. (That is, the superior becomes the primary for the
>   connection.)
>
> NOTPULLED
>   The relationship has not been established (possibly, because the
>   secondary party no longer has the requested transaction). The
>   connection remains in Idle state.
>
> ERROR
>   The command was issued in the wrong state, or was malformed. The
>   connection enters the Error state.

PUSH <superior's transaction identifier>

This command is valid only in the Idle state. It seeks to establish
a superior/subordinate relationship in a transaction with the
primary as the superior. Note that the entire value of <transaction
string> (as defined in the section "TIP Uniform Resource Locators")
must be specified as the transaction identifier. Possible responses
are:

> PUSHED <subordinate's transaction identifier>
>   The relationship has been established, and the identifier by
>   which the subordinate knows the transaction is returned. The
>   transaction becomes the current transaction for the connection,
>   and the connection enters Enlisted state.
>
> ALREADYPUSHED <subordinate's transaction identifier>
>   The relationship has been established, and the identifier by
>   which the subordinate knows the transaction is returned.
>   However, the subordinate already knows about the transaction, and
>   is expecting the two-phase commit protocol to arrive via a
>   different connection. In this case, the connection remains in the
>   Idle state.
>
> NOTPUSHED
>   The relationship could not be established. The connection remains
>   in the Idle state.

ERROR
  The command was issued in the wrong state, or was malformed. The
  connection enters Error state.

QUERY <superior's transaction identifier>

  This command is valid only in the Idle state. A subordinate uses
  this command to determine whether a specific transaction still
  exists at the superior. Possible responses are:

  QUERIEDEXISTS
    The transaction still exists. The connection remains in the Idle
    state.

  QUERIEDNOTFOUND
    The transaction no longer exists. The connection remains in the
    Idle state.

  ERROR
    The command was issued in the wrong state, or was malformed. The
    connection enters Error state.

RECONNECT <subordinate's transaction identifier>

  This command is valid only in the Idle state. A superior uses the
  command to re-establish a connection for a transaction, when the
  previous connection was lost during Prepared state. Possible
  responses are:

  RECONNECTED
    The subordinate accepts the reconnection. The connection enters
    Prepared state.

  NOTRECONNECTED
    The subordinate no longer knows about the transaction. The
    connection remains in Idle state.

  ERROR
    The command was issued in the wrong state, or was malformed. The
    connection enters Error state.

TLS

  This command is valid only in the Initial state. A primary uses
  this command to attempt to establish a secured connection using
  TLS.

If the TLS command is accepted, the TLS protocol will totally
control the underlying connection. This protocol will begin with
the first octet after the line terminator of the TLS command (for
data sent by the primary), and the first byte after the line
terminator of the TLSING response (for data received by the
primary). This implies that an implementation must not send both a
CR and a LF octet after either the TLS command or the TLSING
response, lest the LF octet be mistaken for the first byte of the
TLS protocol.

Possible responses to the TLS command are:

TLSING
  The secondary party agrees to use the TLS protocol [3]. The
  connection enters the Tls state, and all subsequent communication
  is as defined by the TLS protocol. The connection provided by the
  TLS protocol starts out in the Initial state.

CANTTLS
  The secondary party cannot support (or refuses to use) the TLS
  protocol. The connection remains in the Initial state.

ERROR
  The command was issued in the wrong state, or was malformed. The
  connection enters the Error state.

14. Error Handling

If either party receives a line that it cannot understand it closes
the connection. If either party (either a command or a response),
receives an ERROR indication or an ERROR response on a connection the
connection enters the Error state and no further communication is
possible on that connection. An implementation may decide to close
the connection. Closing of the connection is treated by the other
party as a communication failure.

Receipt of an ERROR indication or an ERROR response indicates that
the other party believes that you have not properly implemented the
protocol.

15. Connection Failure and Recovery

A connection failure may be caused by a communication failure, or by
any party closing the connection. It is assumed TIP implementations
will use some private mechanism to detect TIP connection failure
(e.g. socket keepalive, or a timeout scheme).

Depending on the state of a connection, transaction managers will
need to take various actions when a connection fails.

If the connection fails in Initial or Idle state, the connection does
not refer to a transaction. No action is necessary.

If the connection fails in the Multiplexing state, all connections
provided by the multiplexing protocol are assumed to have failed.
Each of them will be treated independently.

If the connection fails in Begun or Enlisted state and COMMIT has
been sent, then transaction completion has been delegated to the
subordinate (the superior is not involved); the outcome of the
transaction is unknown by the superior (it is known at the
subordinate). The superior uses application-specific means to
determine the outcome of the transaction (note that transaction
integrity is not compromised in this case since the superior has no
recoverable resources involved in the transaction). If the connection
fails in Begun or Enlisted state and COMMIT has not been sent, the
transaction will be aborted.

If the connection fails in Prepared state, then the appropriate
action is different for the superior and subordinate in the
transaction.

If the superior determines that the transaction commits, then it must
eventually establish a new connection to the subordinate, and send a
RECONNECT command for the transaction. If it receives a
NOTRECONNECTED response, it need do nothing else. However, if it
receives a RECONNECTED response, it must send a COMMIT request and
receive a COMMITTED response.

If the superior determines that the transaction aborts, it is allowed
to (but not required to) establish a new connection and send a
RECONNECT command for the transaction. If it receives a RECONNECTED
response, it should send an ABORT command.

The above definition allows the superior to reestablish the
connection before it knows the outcome of the transaction, if it
finds that convenient. Having succeeded in a RECONNECT command, the
connection is back in Prepared state, and the superior can send a
COMMIT or ABORT command as appropriate when it knows the transaction
outcome.

Note that it is possible for a RECONNECT command to be received by
the subordinate before it is aware that the previous connection has
failed. In this case the subordinate treats the RECONNECT command as

a failure indication and cleans-up any resources associated with the
connection, and associates the transaction state with the new
connection.

If a subordinate notices a connection failure in Prepared state, then
it should periodically attempt to create a new connection to the
superior and send a QUERY command for the transaction. It should
continue doing this until one of the following two events occurs:

1. It receives a QUERIEDNOTFOUND response from the superior. In this
   case, the subordinate should abort the transaction.

2. The superior, on some connection that it initiated, sends a
   RECONNECT command for the transaction to the subordinate. In this
   case, the subordinate can expect to learn the outcome of the
   transaction on this new connection. If this new connection should
   fail before the subordinate learns the outcome of the transaction,
   it should again start sending QUERY commands.

Note that if a TIP system receives either a QUERY or a RECONNECT
command, and for some reason is unable to satisfy the request (e.g.
the necessary recovery information is not currently available), then
the connection should be dropped.

16. Security Considerations

This section is meant to inform application developers, transaction
manager developers, and users of the security implications of TIP as
described by this document. The discussion does not include
definitive solutions to the issues described, though it does make
some suggestions for reducing security risks.

As with all two phase-commit protocols, any security mechanisms
applied to the application communication protocol are liable to be
subverted unless corresponding mechanisms are applied to the
commitment protocol. For example, any authentication between the
parties using the application protocol must be supported by security
of the TIP exchanges to at least the same level of certainty.

16.1. TLS, Mutual Authentication and Authorization

TLS provides optional client-side authentication, optional server-
side authentication, and optional packet encryption.

A TIP implementation may refuse to provide service unless TLS is
being used. It may refuse to provide service if packet encryption is
not being used. It may refuse to provide service unless the remote
party has been authenticated (via TLS).

A TIP implementation should be willing to be authenticated itself
(via TLS). This is true regardless of whether the implementation is
acting as a client or a server.

Once a remote party has been authenticated, a TIP transaction manager
may use that remote party's identity to decide what operations to
allow.

Whether TLS is to be used on a connection, and if so, how TLS is to
be used, and what operations are to subsequently be allowed, is
determined by the security policies of the connecting TIP transaction
managers towards each other. How these security policies are defined,
and how a TIP transaction manager learns of them is totally private
to the implementation and beyond the scope of this document.

16.2. PULL-Based Denial-of-Service Attack

Assume that a malicious user knows the identity of a transaction that
is currently active in some transaction manager. If the malefactor
opens a TIP connection to the transaction manager, sends a PULL
command, then closes the connection, he can cause that transaction to
be aborted. This results in a denial of service to the legitimate
owner of the transaction.

An implementation may avoid this attack by refusing PULL commands
unless TLS is being used, the remote party has been authenticated,
and the remote party is trusted.

16.3. PUSH-Based Denial-of-Service Attack

When the connection between two transaction managers is closed while
a transaction is in the Prepared state, each transaction manager
needs to remember information about the transaction until a
connection can be re-established.

If a malicious user exploits this fact to repeatedly create
transactions, get them into Prepared state and drop the connection,
he may cause a transaction manager to suffer resource exhaustion,
thus denying service to all legitimate users of that transaction
manager.

An implementation may avoid this attack by refusing PUSH commands
unless TLS is being used, the remote party has been authenticated,
and the remote party is trusted.

16.4. Transaction Corruption Attack

   If a subordinate transaction manager has lost its connection for a
   particular prepared transaction, a malicious user can initiate a TIP
   connection to the transaction manager, and send it a RECONNECT
   command followed by either a COMMIT or an ABORT command for the
   transaction. The malicious user could thus cause part of a
   transaction to be committed when it should have been aborted, or vice
   versa.

   An implementation may avoid this attack by recording the
   authenticated identity of its superior in a transaction, and by
   refusing RECONNECT commands unless TLS is being used and the
   authenticated identity of the remote party is the same as the
   identity of the original superior.

16.5. Packet-Sniffing Attacks

   If a malicious user can intercept traffic on a TIP connection, he may
   be able to deduce information useful in planning other attacks. For
   example, if comment fields include the product name and version
   number of a transaction manager, a malicious user might be able to
   use this information to determine what security bugs exist in the
   implementation.

   An implementation may avoid this attack by always using TLS to
   provide session encryption, and by not putting any personalizing
   information on the TLS/TLSING command/response pair.

16.6. Man-in-the-Middle Attack

   If a malicious user can intercept and alter traffic on a TIP
   connection, he can wreak havoc in a number of ways. For example, he
   could replace a COMMIT command with an ABORT command.

   An implementation may avoid this attack by always using TLS to
   provide session encryption and authentication of the remote party.

17. References

   [1] Gray, J. and A. Reuter (1993), Transaction Processing: Concepts
        and Techniques. San Francisco, CA: Morgan Kaufmann Publishers.
        (ISBN 1-55860-190-2).

   [2] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and T.
        Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC
        2068, January 1997.

   [3] Dierks, T., et. al., "The TLS Protocol Version 1.0", Work in
        Progress.

   [4] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform
        Resource Locators (URL)", RFC 1738, December 1994.

   [5] Evans, K., Klein, J., and J. Lyon, "Transaction Internet
        Protocol - Requirements and Supplemental Information", RFC 2372,
        July 1998.

   [6] Moats, R., "URN Syntax", RFC 2141, May 1997.

   [7] Faltstrom, P., et. al., "Namespace Identifier Requirements for
        URN Services", Work in Progress.

   [8] Berners-Lee, T., et. at., "Uniform Resource Identifiers (URI):
        Generic Syntax and Semantics", Work in Progress.

   [9] Leach, P., and R. Salz, "UUIDs and GUIDs", Work in Progress.

18. Authors' Addresses

   Jim Lyon
   Microsoft Corporation
   One Microsoft Way
   Redmond, WA 98052-6399, USA

   Phone: +1 (206) 936 0867
   Fax: +1 (206) 936 7329
   EMail: JimLyon@Microsoft.Com

   Keith Evans
   Tandem Computers, Inc.
   5425 Stevens Creek Blvd
   Santa Clara, CA 95051-7200, USA

   Phone: +1 (408) 285 5314
   Fax: +1 (408) 285 5245
   EMail: Keith.Evans@Tandem.Com

   Johannes Klein
   Tandem Computers Inc.
   10555 Ridgeview Court
   Cupertino, CA 95014-0789, USA

   Phone: +1 (408) 285 0453
   Fax: +1 (408) 285 9818
   EMail: Johannes.Klein@Tandem.Com

19. Comments

   Please send comments on this document to the authors at
   <JimLyon@Microsoft.Com>, <Keith.Evans@Tandem.Com>,
   <Johannes.Klein@Tandem.Com>, or to the TIP mailing list at
   <Tip@Lists.Tandem.Com>. You can subscribe to the TIP mailing list by
   sending mail to <Listserv@Lists.Tandem.Com> with the line "subscribe
   tip <full name>" somewhere in the body of the message.

Appendix A. The TIP Multiplexing Protocol Version 2.0.

   This appendix describes version 2.0 of the TIP Multiplexing Protocol
   (TMP). TMP is intended solely for use with the TIP protocol, and
   forms part of the TIP protocol specification (although its
   implementation is optional). TMP V2.0 is the only multiplexing
   protocol supported by TIP V3.0.

Abstract

   TMP provides a simple mechanism for creating multiple lightweight
   connections over a single TCP connection. Several such lightweight
   connections can be active simultaneously. TMP provides a byte
   oriented service, but allows message boundaries to be marked.

A.1. Introduction

   There are several protocols in widespread use on the Internet which
   create a single TCP connection for each transaction. Unfortunately,
   because these transactions are short lived, the cost of setting up
   and tearing down these TCP connections becomes significant, both in
   terms of resources used and in the delays associated with TCP's
   congestion control mechanisms.

   The TIP Multiplexing Protocol (TMP) is a simple protocol running on
   top of TCP that can be used to create multiple lightweight
   connections over a single transport connection. TMP therefore
   provides for more efficient use of TCP connections. Data from several
   different TMP connections can be interleaved, and both message
   boundaries and end of stream markers can be provided.

   Because TMP runs on top of a reliable byte ordered transport service
   it can avoid most of the extra work TCP must go through in order to
   ensure reliability. For example, TMP connections do not need to be
   confirmed, so there is no need to wait for handshaking to complete
   before data can be sent.

   Note: TMP is not intended as a generalized multiplexing protocol. If
   you are designing a different protocol that needs multiplexing, TMP
   may or may not be appropriate. Protocols with large messages can
   exceed the buffering capabilities of the receiver, and under certain
   conditions this can cause deadlock. TMP when used with TIP does not
   suffer from this problem since TIP is a request-response protocol,
   and all messages are short.

A.2. Protocol Model

   The basic protocol model is that of multiple lightweight connections
   operating over a reliable stream of bytes. The party which initiated
   the connection is referred to as the primary, and the party which
   accepted the connection is referred to as the secondary.

   Connections may be unidirectional or bi-directional; each end of a
   bi-directional connection may be closed separately. Connections may
   be closed normally, or reset to indicate an abortive release.
   Aborting a connection closes both data streams.

   Once a connection has been opened, applications can send messages
   over it, and signal the end of application level messages.
   Application messages are encapsulated in TMP packets and transferred
   over the byte stream. A single TIP command (TMP application message)
   must be wholly contained within a single TMP packet.

A.3. TMP Packet Format

   A TMP packet consists of a 64 bit header followed by zero or more
   octets of data. The header contains three fields; a flag byte, the
   connection identifier, and the packet length. Both integers, the
   connection identifier and the packet length must be sent in network
   byte order.

```
    FLAGS
   +--------+--------+--------+--------+
   |SFPR0000| Connection ID            |
   +--------+--------+--------+--------+
   |        | Length                   |
   +--------+--------+--------+--------+
```

A.3.1. Flag Details

```
   +-------+----------+---------------------------------------+
   | Name  | Mask     | Description                           |
   +-------+----------+---------------------------------------+
   | SYN   | 1xxx|0000 | Open a new connection                 |
   | FIN   | x1xx|0000 | Close an existing connection          |
   | PUSH  | xx1x|0000 | Mark application level message boundary |
   | RESET | xxx1|0000 | Abort the connection                  |
   +-------+----------+---------------------------------------+
```

A.4. Connection Identifiers

   Each TMP connection is identified by a 24 bit integer. TMP
   connections created by the party which initiated the underlying TCP
   connection must have even identifiers; those created by the other
   party must have odd identifiers.

A.5. TMP Connection States

   TMP connections can exist in several different states; Closed,
   OpenWrite, OpenSynRead, OpenSynReset, OpenReadWrite, CloseWrite, and
   CloseRead. A connection can change its state in response to receiving
   a packet with the SYN, FIN, or RESET bits set, or in response to an
   API call by the application. The available API calls are open, close,
   and abort.

   The meaning of most states is obvious (e.g. OpenWrite means that a
   connection has been opened for writing). The meaning of the states
   OpenSynRead and OpenResetRead need more explanation.

   In the OpenSynRead state a primary opened and immediately closed the
   output data stream of a connection, and is now waiting for a SYN
   response from the secondary to open the input data stream for
   reading.

   In the OpenResetRead state a primary opened and immediately aborted a
   connection, and is now waiting for a SYN response from the secondary
   to finally close the connection.

A.6. Event Priorities and State Transitions

   The state table shown below describes the actions and state
   transitions that occur in response to a given event. The events
   accepted by each state are listed in priority order with highest
   priority first. If multiple events are present in a message, those
   events matching the list are processed. If multiple events match, the
   event with the highest priority is accepted and processed first. Any
   remaining events are processed in the resultant successor state.

   For example, if a TMP connection at the secondary is in the Closed
   state, and the secondary receives a packet containing a SYN event, a
   FIN event and an input data event (i.e. DATA-IN), the secondary first
   accepts the SYN event (because it is the only match in Closed state).
   The secondary accepts the connection, sends a SYN event and enters
   the ReadWrite state. The SYN event is removed from the list of
   pending events. The remaining events are FIN and DATA-IN. In the
   ReadWrite state the secondary reads the input data (i.e. the DATA-IN
   event is processed first because it has higher priority than the FIN

event). Once the data has been read and the DATA-IN event has been
removed from the list of pending events, the FIN event is processed
and the secondary enters the CloseWrite state.

If the secondary receives a packet containing a SYN event, and is for
some reason unable to accept the connection (e.g. insufficient
resources), it should reject the request by sending a SYN event
followed by a RESET event. Note that both events can be sent as part
of the same TMP packet.

If either party receives a TMP packet that it does not understand, or
an event in an incorrect state, it closes the TCP connection.

```
+==============+=========+==========+==============+
| Entry State  | Event   | Action   | Exit State   |
+==============+=========+==========+==============+
| Closed       | SYN     | SYN      | ReadWrite    |
|              | OPEN    | SYN      | OpenWrite    |
+--------------+---------+----------+--------------+
| OpenWrite    | SYN     | Accept   | ReadWrite    |
|              | WRITE   | DATA-OUT | OpenWrite    |
|              | CLOSE   | FIN      | OpenSynRead  |
|              | ABORT   | RESET    | OpenSynReset |
+--------------+---------+----------+--------------+
| OpenSynRead  | SYN     | Accept   | CloseRead    |
+--------------+---------+----------+--------------+
| OpenSynReset | SYN     | Accept   | Closed       |
+--------------+---------+----------+--------------+
| ReadWrite    | DATA-IN | Accept   | ReadWrite    |
|              | FIN     | Accept   | CloseWrite   |
|              | RESET   | Accept   | Closed       |
|              | WRITE   | DATA-OUT | ReadWrite    |
|              | CLOSE   | FIN      | CloseRead    |
|              | ABORT   | RESET    | Closed       |
+--------------+---------+----------+--------------+
| CloseWrite   | RESET   | Accept   | Closed       |
|              | WRITE   | DATA-OUT | CloseWrite   |
|              | CLOSE   | FIN      | Closed       |
|              | ABORT   | RESET    | Closed       |
+--------------+---------+----------+--------------+
| CloseRead    | DATA-IN | Accept   | CloseRead    |
|              | FIN     | Accept   | Closed       |
|              | RESET   | Accept   | Closed       |
|              | ABORT   | RESET    | Closed       |
+--------------+---------+----------+--------------+
```

           TMP Event Priorities and State Transitions

    Copyright (C) The Internet Society (1998). All Rights Reserved.

    This document and translations of it may be copied and furnished to
    others, and derivative works that comment on or otherwise explain it
    or assist in its implementation may be prepared, copied, published
    and distributed, in whole or in part, without restriction of any
    kind, provided that the above copyright notice and this paragraph are
    included on all such copies and derivative works. However, this
    document itself may not be modified in any way, such as by removing
    the copyright notice or references to the Internet Society or other
    Internet organizations, except as needed for the purpose of
    developing Internet standards in which case the procedures for
    copyrights defined in the Internet Standards process must be
    followed, or as required to translate it into languages other than
    English.

    The limited permissions granted above are perpetual and will not be
    revoked by the Internet Society or its successors or assigns.

    This document and the information contained herein is provided on an
    "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING
    TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING
    BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION
    HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF
    MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.