

Network Working Group
Request for Comments: 2372
Category: Informational

K. Evans
J. Klein
Tandem Computers
J. Lyon
Microsoft
July 1998

Transaction Internet Protocol - Requirements and Supplemental Information

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

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

Abstract

This document describes the purpose (usage scenarios), and requirements for the Transaction Internet Protocol [1]. It is intended to help qualify the necessary features and functions of the protocol. It also provides supplemental information to aid understanding and facilitate implementation of the TIP protocol.

Table of Contents

1. Introduction	2
2. The Transaction Internet Protocol	3
3. Scope	4
4. Anticipated Usage of TIP	4
5. TIP Compliant Systems	4
6. Relationship to the X/Open DTP Model	5
7. Example TIP Usage Scenario	5
8. TIP Transaction Recovery	9
9. TIP Transaction and Application Message Serialisation	10
10. TIP Protocol and Local Actions	10
11. Security Considerations	11
12. TIP Requirements	11
References	14
Authors' Addresses	15
Comments	15
A. An Example TIP Transaction Manager API	16
Full Copyright Statement	24

1. Introduction

Transactions are a very useful programming paradigm, greatly simplifying the writing of distributed applications. When transactions are employed, no matter how many distributed application components participate in a particular unit-of-work, the number of possible outcomes is reduced to only two; that is, either all of the work completed successfully, or none of it did (this characteristic is known as atomicity). Applications programming is therefore much less complex since the programmer does not have to deal with a multitude of possible failure scenarios. Typically, transaction semantics are provided by some underlying system infrastructure (usually in the form of products such as Transaction Processing Monitors, and/or Databases). This infrastructure deals with failures, and performs the necessary recovery actions to guarantee the property of atomicity. The use of transactions enables the development of reliable distributed applications which would otherwise be difficult, if not impossible.

A key technology required to support distributed transactions is the two-phase commit protocol (2-pc). 2-pc protocols have been used in commercial Transaction Processing (TP) systems for many years, and are well understood (e.g. the LU6.2 2-pc (syncpoint) protocol was first implemented more than 12 years ago). Today a number of different 2-pc protocols are supported by a variety of TP monitor and database products. 2-pc is used between the components participating in a distributed unit-of-work (transaction) to ensure agreement by all parties regarding the outcome of that work (regardless of any failure).

Today both standard and proprietary 2-pc protocols exist. These protocols typically employ a "one-pipe" model. That is, the transaction and application protocols are tightly-integrated, executing over the same communications channel. An application may use only the particular communications mechanism associated with the transaction protocol. The standard protocols (OSI TP, LU6.2) are complex, with a large footprint and extensive configuration and administration requirements. For these reasons they are not very widely deployed. The net of all this is restricted application flexibility and interoperability if transactions are to be used. Applications may wish to use a number of communications protocols for which there are no transactional variants (e.g. HTTP), and be deployed in very heterogeneous application environments.

In summary, transactions greatly simplify the programming of distributed applications, and the 2-pc protocol is a key transactional technology. Current 2-pc protocols only offer transaction semantics to a limited set of applications, operating

within a special-purpose (complex, homogeneous) infrastructure, using a particular set of intercommunication protocols. The restrictions thus imposed by current 2-pc protocols limits the widespread use of the transaction paradigm, thereby inhibiting the development of new distributed business applications.

(See [2] for more information re transactions, atomicity, and two-phase commit protocols in general.)

2. The Transaction Internet Protocol (TIP)

TIP is a 2-pc protocol which is intended to provide ubiquitous distributed transaction support, in a heterogeneous (networked) environment. TIP removes the restrictions of current 2-pc protocols and enables the development of new distributed business applications.

This goal is achieved primarily by satisfying two key requirements:

- 1) Keep the protocol simple (yet functionally sufficient). If the protocol is complex it will not be widely deployed or quickly adopted. Simplicity also means suitability to a wide range of application environments.
- 2) Enable the protocol to be used with any applications communications protocol (e.g. HTTP). This ensures heterogeneous environments can participate in distributed work.

TIP does not reinvent the 2-pc protocol itself, the well-known presumed-abort 2-pc protocol is used as a basis. Rather the novelty and utility of TIP is in its separation from the application communications protocol (the two-pipe model).

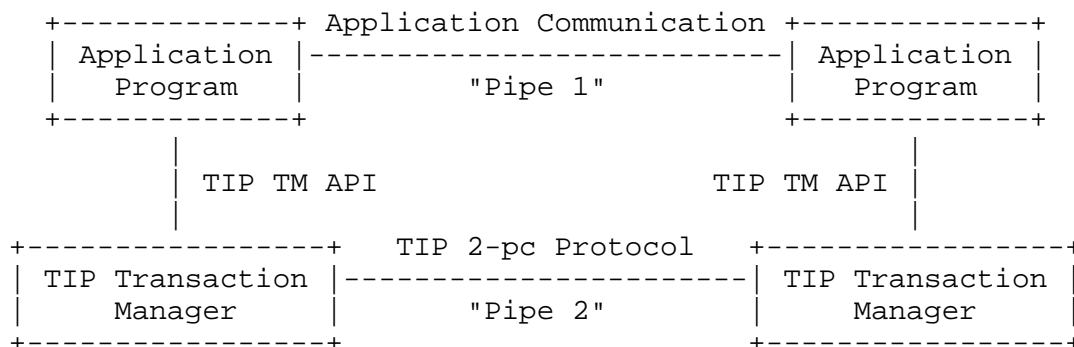


Fig 1: The two-pipe nature of TIP

3. Scope

TIP does not describe how business transactions or electronic commerce are to be conducted on the internet, it specifies only the 2-pc transaction protocol (which is an aid in the development of such applications). e.g. TIP does not provide a mechanism for non-repudiation. Such protocols might be a subject for subsequent IETF activity, once the requirements for general electronic commerce are better understood. TIP does not preclude the later definition of these protocols.

TIP does not specify Application Programming Interfaces (note that an example TIP TM API is included in this document (Appendix A), as an aid to understanding).

4. Anticipated Usage of TIP

As described above, transactions are a very useful tool in simplifying the programming of distributed applications. TIP is therefore targeted at any application that involves distributed work. Such applications may comprise components executing within a single system, across a corporate intranet, across the internet, or any other distributed system configuration. The application may be of "enterprise" class (requiring high-levels of performance and availability), or be less demanding. TIP is intended to be generally applicable, meeting the requirements of any application type which would benefit from the provision of transaction semantics.

5. TIP Compliant Systems

There are two classes of TIP compliant Transaction Manager system:

- 1) Client-only systems. Those which provide an application interface to demarcate TIP transactions, but which do not offer access to local recoverable resources. Such a lightweight implementation is useful for systems which host client applications only (e.g. desktop machines). Such client systems may be unreliable, and are not appropriate as transaction coordinators (their unavailability might cause resources on other transaction participant systems to remain locked and unavailable). These so-called "volatile client" systems therefore delegate the responsibility to coordinate the transaction (and recover from failures), to other "full" (server) TIP system implementations. For these lightweight systems, only the TIP IDENTIFY, BEGIN, COMMIT, and ABORT commands are needed; no transaction log is required.

- 2) Server systems. Those which offer the above support, plus TIP transaction coordination and recovery services. These systems may also provide access to recoverable resources (e.g. relational databases). Server systems support all TIP commands, and provide a recoverable transaction log.

A TIP compliant Transaction Manager (TM), will also supply application programming interfaces to demarcate transactions (e.g. the X/Open TX interface [3]), plus commands to generate TIP URLs, to PUSH/PULL TIP transactions, and to set the current TIP transaction context. TIP support can be added to TMs with existing APIs and 2-pc protocols, and transactions may comprise both proprietary and TIP transaction branches (it is assumed existing TM implementations will provide "TIP gateway" facilities which will coordinate between TIP and other transaction protocols).

6. Relationship to the X/Open DTP Model

The X/Open Distributed Transaction Processing (DTP) Model [4] defines four components: 1) Application Program (AP), 2) Transaction Manager (TM), 3) Resource Manager (RM), and 4) Communications Resource Manager (CRM). In this model, TIP defines a TM to TM interoperability protocol, which is independent of application communications (there is no such equivalent protocol specified by X/Open, where all transaction and application communication occurs between CRMs (the one-pipe model)). Programmatic interfaces between the AP and TM/RM are unaffected by, and may be used with TIP. The TM to RM interaction is defined via the X/Open XA interface specification [5]. TIP is compatible with XA, and a TIP transaction may comprise applications accessing multiple RMs where the XA interface is being used to coordinate the RM transaction branches.

7. Example TIP Usage Scenario

It is expected that a typical internet usage of TIP will involve applications using the agency model. In this model, the client node itself is not directly involved in the TIP protocol at all, and does not need the services of a local TIP TM. Instead, an agency (server) application handles the dialogue with the client, and is responsible for the coordination of the TIP transaction. The agency works with other service providers to deliver the service to the client. e.g. as a Travel Agency acts as an intermediate between airlines/hotels/etc and the customer. A big benefit of this model is that the agency is trusted by the service providers, and there are fewer such agencies (compared to user clients), so issues of security and performance are reduced.

Consider a Travel Agency example. A client running a web browser on a network PC accesses the Travel Agency web page. Via pages served up by the agency (which may in turn be constructed from pages provided by the airline and hotel servers), the client creates an itinerary involving flights and hotel choices. Finally, the client clicks the "make reservation" button. At this point the following sequence of events occurs (user-written application code is invoked by the various web servers, via any of the standard or proprietary techniques available (e.g. CGI)):

- 1) The travel agency begins a local transaction, and gets a TIP URL for this transaction (both of these functions are performed using the API of the local TM. e.g. "tip_xid_to_url()" would return the TIP URL for the local transaction). The TIP URL contains the listening endpoint IP address of the local TM and the transaction identifier of the local transaction.
- 2) The travel agency application sends a request to the airline server (via some protocol (e.g. HTTP)), requesting the "book_flight" service, passing the flights selected by the client, and the TIP URL (obtained in 1. above).
- 3) The request is received by the airline server which invokes the book_flight application. This application retrieves the TIP URL from the input data, and passes this on a "tip_pull()" API request to its local TM. The tip_pull() function causes the following to occur:
 - a. the local TM creates a local transaction (under which the work will be performed),
 - b. if a TIP connection does not already exist to the superior (travel agency) TM (as identified via the IP address passed in the TIP URL), one is created and an IDENTIFY exchange occurs (if multiplexing is to be used on the connection, this is followed by a MULTIPLEX exchange),
 - c. a PULL command is sent to the superior TM,
 - d. in response to the PULL, the superior TM associates the subordinate (airline) TM with the transaction (by associating the connection with the transaction), and sends a PULLED response to the subordinate TM,
 - e. the subordinate TM returns control to the book_flight application, which is now executing in the context of the newly created local transaction.

- 4) The `book_flight` application does its work (which may involve access to a recoverable resource manager (e.g. an RDBMS), in which case the local TM will associate the RM with the local transaction (via the XA interface or whatever)).
- 5) The `book_flight` application returns to the travel agency application indicating success.
- 6) Steps 2-5 are then repeated with the hotel server "`book_room`" application. At the conclusion of this, the superior TM has registered two subordinate TMs as participants in the transaction, there are TIP connections between the agency TM and the airline and hotel TMs, and there are inflight transactions at the airline and hotel servers. [Note that steps 2-5 and 6 could be performed in parallel.]
- 7) The travel agency application issues a "commit transaction" request (using the API of the local TM). The local TM sends a PREPARE command on the TIP connections to the airline and hotel TMs (as these are registered as subordinate transaction participants).
- 8) The TMs at the airline and hotel servers perform the necessary steps to prepare their local recoverable resources (e.g. by issuing `xa_prepare()` requests). If successful, the subordinate TMs change their TIP transaction state to Prepared, and log recovery information (e.g. local and superior transaction branch identifiers, and the IP address of the superior TM). The subordinate TMs then send PREPARED commands to the superior TM.
- 9) If both subordinates respond PREPARED, the superior TM logs that the transaction is Committed, with recovery information (e.g. local and subordinate transaction identifiers, and subordinate TM IP addresses). The superior TM then sends COMMIT commands on the two subordinate TIP connections.
- 10) The TMs at the airline and hotel servers perform the necessary steps to commit their local recoverable resources (e.g. by issuing `xa_commit()` requests). The subordinate TMs forget the transaction. The subordinate TMs then send COMMITTED commands to the superior TM.
- 11) The superior TM forgets the transaction. The TIP connections between the superior and subordinate TMs return to Idle state (not associated with any transaction). The superior TM returns success to the travel agency application "commit transaction" request.

- 12) The travel agency application returns "reservation made" to the client.

This example illustrates the use of PULL. If PUSH were to be used instead, events 2) and 3) above would change as follows:

- 2) The travel agency application:

- a. passes the TIP URL obtained in 1. above, together with the listening endpoint address of the TM at the airline server, to its local TM via a "tip_push()" API request. The tip_push() function causes the following to occur:
 - i. if a TIP connection does not already exist to the subordinate (airline server) TM (as identified via the IP address passed on the tip_push), one is created and an IDENTIFY exchange occurs (if multiplexing is to be used on the connection, this is followed by a MULTIPLEX exchange),
 - ii. a PUSH command is sent to the subordinate TM,
 - iii. in response to the PUSH, the subordinate TM creates a local transaction, associates this transaction with the connection, and sends a PUSHED response to the superior TM,
 - iv. in response to the PUSHED response, the superior TM associates the subordinate TM with the transaction,
 - v. the superior TM returns control to the travel agency application.
- b. the travel agency application sends a request to the airline server (via some protocol (e.g. HTTP)), requesting the "book_flight" service, passing the flights selected by the client, and the TIP URL (obtained in 1 above).

- 3) The request is received by the airline server which invokes the book_flight application. This application retrieves the TIP URL from the input data, and passes this on a "tip_pull()" API request to its local TM. Since the local TM has already "seen" this URL (it was already pushed), it simply returns to the book_flight application, which is now executing in the context of the previously created local transaction.

[Note that although in this example the transaction coordinator role is performed by a node which is also a participant in the transaction (the Travel Agency), other configurations are possible (e.g. where the transaction coordinator role is performed by a non-participant 3rd-party node).]

8. TIP Transaction Recovery

Until the transaction reaches the Prepared state, any failure results in the transaction being aborted. If an error occurs once the transaction has reached the Prepared state, then transaction recovery must be performed. Recovery behaviour is different for superior and subordinate; the details depend upon the outcome of the transaction (committed or aborted), and the precise point at which failure occurs.

In the travel agency application for example, if the connection to the hotel server fails before the COMMIT command has been received by the hotel TM, then (once the connection is restored):

- 1) The superior (travel agency) TM sends a RECONNECT command (passing the subordinate transaction identifier (recovered from the transaction log if necessary)).
- 2) The subordinate (hotel) TM responds RECONNECTED (since it never received the COMMIT command, and still has the transaction in Prepared state (if the failure had occurred after the subordinate had responded COMMITTED, then the subordinate would have forgotten the transaction, and responded NOTRECONNECTED to the RECONNECT command)).
- 3) The superior TM sends a COMMIT command. The subordinate TM commits the transaction and responds COMMITTED. The transaction is now resolved.
- 4) If the subordinate TM restores the connection to the superior TM before receiving a RECONNECT command, then it may send a QUERY command. In this case, the superior TM will respond QUERIEDEXISTS, and the subordinate TM should wait for the superior to send a RECONNECT command. If the transaction had been aborted, then the superior may respond QUERIEDNOTFOUND, in which case the subordinate should abort the transaction (note that the superior is not obliged to send a RECONNECT command for an aborted transaction (i.e. it could just forget the transaction after sending ABORT and before receiving an ABORTED response)).

There are failure circumstances in which the client application (the one calling "commit") may not receive a response indicating the final outcome of the transaction (even though the transaction itself is successfully completed). This is a common problem, and one not unique to TIP. In such circumstances, it is up to the application to ascertain the final outcome of the transaction (a TIP TM may facilitate this by providing some implementation specific mechanism. e.g. writing the outcome to a user-log).

9. TIP Transaction and Application Message Serialisation

A relationship exists between TIP commands and application messages: a TIP transaction must not be committed until it is certain that all participants have properly registered, and have finished work on the transaction. Because of the two-pipe nature of TIP, this behaviour cannot necessarily be enforced by the TIP system itself (although it may be possible in some implementations). It is therefore incumbent upon the application to behave properly. Generally, an application must not:

- 1) call it's local TMs "commit" function when it has any requests associated with the transaction still outstanding.
- 2) positively respond to a transactional request from a partner application prior to having registered it's local TM with the transaction.

10. TIP Protocol and Local Actions

In order to ensure that transaction atomicity is properly guaranteed, a system implementing TIP must perform other local actions at certain points in the protocol exchange. These actions pertain to the creation and deletion of transaction "log-records" (the necessary information which survives failures and ensures that transaction recovery is correctly executed). The following information regarding the relationship between the TIP protocol and logging events is advisory, and is not intended to be definitive (see [2] for more discussion on this subject):

- 1) before sending a PREPARED response, the system should create a prepared-recovery-record for the transaction.
- 2) having created a prepared-recovery-record, this record should not be deleted until after:
 - a. an ABORT message is received; or
 - b. a COMMIT message is received; or
 - c. a QUERIEDNOTFOUND response is received.

- 3) the system should not send a COMMITTED or NOTRECONNECTED message if a prepared-recovery-record exists.
- 4) before creating a commit-recovery-record for the transaction, the system should have received a PREPARED response.
- 5) before sending a COMMIT message in Prepared state, the system should have created a commit-recovery-record for the transaction.
- 6) having created a commit-recovery-record, this record should not be deleted until after:
 - a. a COMMITTED message is received; or
 - b. a NOTRECONNECTED message is received.

11. Security Considerations

The means by which applications communicate and perform distributed work are outside the scope of the TIP protocol. The mechanisms used for authentication and authorisation of clients to access programs and information on a particular system are part of the application communications protocol and the application execution infrastructure. Use of the TIP protocol does not affect these considerations.

Security relates to the TIP protocol itself inasmuch that systems require to protect themselves from the receipt of unauthorised TIP commands, or the impersonation of a trusted partner TIP TM. Probably the worst consequence of this is the possibility of undetected data inconsistency resulting from violations of the TIP commitment protocol (e.g. a COMMIT command is injected on a TIP connection in place of an ABORT command). TIP uses the Transport Layer Security protocol [6] to restrict access to only trusted partners (i.e. to control from which remote endpoints TIP transactions will be accepted, and to verify that an end-point is genuine), and to encrypt TIP commands. Usage of TLS (or not) is negotiated between partner TIP TMs. See [1] for details of how TLS is used with TIP.

TIP TM implementations will also likely provide local means to time-out and abort transactions which have not completed within some time period (thereby preventing unavailability of resources due to malicious intent). Transaction time-out also serves as a means of deadlock resolution.

12. TIP Requirements

Most of these requirements stem from the primary objective of making transactions a ubiquitous system service, available to all application classes (much as TCP may be assumed to be available everywhere). In general this requires imposing as few restrictions

regarding the use of TIP as possible (applications should not be required to execute in some "special" environment in order to use transactions), and keeping the protocol simple and efficient. This enables the widespread implementation of TIP (it's cheap to do), on a wide range of systems (it's cheap to run).

1) Application Communications Protocol Independence

The TIP protocol must be defined independently of the communications protocol used for transferring application data, to allow TIP usage in conjunction with any application protocol. It must be possible for applications using arbitrary communications protocols to begin, end, and propagate TIP transactions.

This implies that the TIP protocol employ a 2-pipe model of operation. This model requires the separation of application communications and transaction coordination, into two discrete communication channels (pipes). This separation enables the use of the transaction coordination protocol (TIP), with any application communications protocol (e.g. HTTP, ODBC, plain TCP/UDP, etc).

2) Support for Transaction Semantics

The TIP protocol must provide the functionality of the de-facto standard presumed-abort 2-pc protocol, to guarantee transactional atomicity even in the event of failure. It should provide a means to construct the transaction tree, as well as provide commitment and recovery functions.

3) Application Transaction Propagation and Interoperability

In order to facilitate protocol independence, application interoperability, and provide a means for TIP transaction context propagation, a standard representation of the TIP transaction context information is required (in the form of a URL). This information must include the listening endpoint address of the partner TIP TM, and transaction identifier information.

4) Ease of Implementation

The TIP protocol must be simple to implement. It should support only those features necessary to provide a useful, performant 2-pc protocol service. The protocol should not add complexity in the form of extraneous optimisations.

5) Suitability for All Application Classes

The TIP protocol should be complete and robust enough not only for electronic commerce on the web, but also for intranet applications and for traditional TP applications spanning heterogeneous transaction manager environments. The protocol should be performant and scalable enough to meet the needs of low to very high throughput applications.

- a. the TIP protocol should support the concept of client-only transaction participants (useful for ultra-lightweight implementations on low-end platforms).
- b. since some clients may be unreliable, TIP must provide support for delegation of transaction coordination (to a more reliable (trusted) node).
- c. the TIP protocol must scale between 1 and n (> 1) concurrent transactions per TCP connection.
- d. TIP commands should be able to be concatenated (pipelined).
- e. TIP should be compatible with the X/Open XA interface.

6) Security

The TIP protocol must be compatible with existing security mechanisms, potentially including encryption, firewalls, and authorization mechanisms (e.g. TLS may be used to authenticate the sender of a TIP command, and for encryption of TIP commands). Nothing in the protocol definition should prevent TIP working within any security environment.

7) TIP Protocol Transport Independence

It would be beneficial to some applications to allow the TIP protocol to flow over different transport protocols. The benefit is when using different transport protocols for the application data, the same transport can be used for the TIP 2PC protocol. TIP must therefore not preclude use with other transport protocols.

8) Recovery

Recovery semantics need to be defined sufficiently to avoid ambiguous results in the event of any type of communications transport failure.

9) Extensibility

The TIP protocol should be able to be extended, whilst maintaining compatibility with previous versions.

References

- [1] Lyon, J., Evans, K., and J. Klein, "The Transaction Internet Protocol Version 3.0", RFC 2371, July 1998.
- [2] Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers. (ISBN 1-55860-190-2). J. Gray, A. Reuter.
- [3] X/Open CAE Specification, April 1995, Distributed Transaction Processing: The TX Specification. (ISBN 1-85912-094-6).
- [4] X/Open Guide, November 1993, Distributed Transaction Processing: Reference Model Version 2. (ISBN 1-85912-019-9).
- [5] X/Open CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification. (ISBN 1-872630-24-3).
- [6] Dierks, T., et. al., "The TLS Protocol Version 1.0", Work in Progress.

Authors' Addresses

Keith Evans
Tandem Computers Inc, LOC 252-30
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

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

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@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. An Example TIP Transaction Manager Application Programming Interface.

Note that this API is included solely for informational purposes, and is not part of the formal TIP specification (TIP conformant implementations are free to define alternative APIs).

1) tip_open() - establish a connection to a TIP TM.

Synopsis

```
int tip_open ([out] tip_handle_t *ptiptm)
```

Parameters

```
ptiptm [out]
```

Pointer to the TIP TM handle.

Description

tip_open() establishes a connection to a TIP TM. The call returns a handle which identifies the TIP TM. This function must be called before any work can be performed on a TIP transaction.

Return Values

[TIPOK]

Connection has been successfully established.

[TIPNOTCONNECTED]

User has been disconnected from the TIP TM.

[TIPNOTCONFIGURED]

TIP TM has not been configured.

[TIPTRANSIENT]

Too many openers; re-try the open.

[TIPERROR]

An unexpected error occurred.

2) tip_close() - close a connection to a TIP TM.

Synopsis

```
int tip_close([in] tip_handle_t handle)
```

Parameters

```
handle [in]
```

The TIP TM handle.

Description

tip_close() closes a connection to a TIP TM. All outstanding requests associated with that connection will be cancelled.

Return Values

[TIPOK]

Connection has been successfully closed.

[TIPINVALIDPARM]

Invalid connection handle specified.

[TIPERROR]

An unexpected error occurred.

- 3) `tip_push()` - export a local transaction to a remote node and return a TIP transaction identifier for the associated remote transaction.

Synopsis

```
int tip_push ([in] tip_handle_t TM,
              [in] char *tm_url,
              [in] void *plocal_xid,
              [out] char *pxid_url,
              [in] unsigned int url_length)
```

Parameters

TM [in]

The TIP TM handle.

tm_url [in]

Pointer to the TIP URL of the remote transaction manager. A TIP URL for a transaction manager takes the form:
TIP://<host>[:<port>]

plocal_xid [in]

Pointer to the local transaction identifier. The structure of the transaction identifier is defined by the local transaction manager.

pxid_url [out]

Pointer to the TIP URL of the associated remote transaction. A TIP URL for a transaction takes the form:
TIP://<host>[:<port>]/<transaction identifier>

url_length [in]

The size in bytes of the buffer for the remote transaction URL.

Description

`tip_push()` exports (pushes) a local transaction to a remote node. If a local transaction identifier is not supplied, the caller's current transaction context is used. The call returns a TIP URL for the associated remote transaction. The TIP transaction identifier may be passed on application requests to the remote node (as part of a TIP URL). The receiving process uses this information in order to do work on behalf of the transaction.

Return Values

[TIPOK]

Transaction has been successfully pushed to the remote node.

[TIPINVALIDXID]

An invalid transaction identifier has been provided.

[TIPNOCURRENTTX]

Process is currently not associated with a transaction (and none was supplied).

[TIPINVALIDHANDLE]

Invalid connection handle specified.

[TIPNOTPUSHED]

Transaction could not be pushed to the remote node.
[TIPNOTCONNECTED]
Caller has been disconnected from the TIP TM.
[TIPINVALIDURL]
Invalid endpoint URL is provided.
[TIPTRANSIENT]
Transient error occurred; re-try the operation.
[TIPTRUNCATED]
Insufficient buffer size is specified for the TIP
transaction identifier.
[TIPERROR]
An unexpected error occurred.

- 4) `tip_pull()` - create a local transaction and join it with the TIP transaction.

Synopsis

```
int tip_pull([in] tip_handle_t TM,  
             [in] char *pxid_url,  
             [out] void *plocal_xid,  
             [in] unsigned int xid_length)
```

Parameters

TM [in]
The TIP TM handle.

pxid_url [in]
Pointer to the TIP URL of the associated remote transaction. A TIP URL for a transaction takes the form:
TIP://<host>[:<port>]/<transaction identifier>

plocal_xid [out]
Pointer to the local transaction identifier. The structure of the transaction identifier is defined by the local transaction manager.

xid_length [in]
The size in bytes of the buffer for the local transaction identifier.

Description

`tip_pull()` creates a local transaction and joins the local transaction with the TIP transaction (the caller becomes a subordinate participant in the TIP transaction). The remote TIP TM is identified via the URL (`*pxid_url`). The local transaction identifier is returned. If a local transaction has already been created for the TIP transaction identifier supplied, then [TIPOK] is returned (with the local transaction identifier), and no other action is taken.

Return Values

[TIPOK]
The local transaction has been successfully created and joined with the TIP transaction.

[TIPINVALIDHANDLE]

Invalid connection handle specified.

[TIPTRUNCATED]
Insufficient buffer size is specified for the local transaction identifier.

[TIPNOTPULLED]
Joining of the local transaction with the TIP transaction has failed.

[TIPNOTCONNECTED]
Caller has been disconnected from the TIP TM.

[TIPINVALIDURL]
Invalid URL has been supplied.

[TIPTRANSIENT]
Transient error occurred; retry the operation.

[TIPERROR]
An unexpected error occurred.

- 5) `tip_pull_async()` - create a local transaction and join it with the TIP transaction. Control is returned to the caller as soon as a local transaction is created.

Synopsis

```
int tip_pull_async ([in] tip_handle_t TM
                   [in] char *pxid_url,
                   [out] void *plocal_xid,
                   [in] unsigned int xid_length)
```

Parameters

`TM` [in]
The TIP gateway handle.

`pxid_url` [in]
Pointer to the TIP URL of the associated remote transaction. A TIP URL for a transaction takes the form:
TIP://<host>[:<port>]/<transaction identifier>

`plocal_xid` [out]
Pointer to the local transaction identifier. The structure of the transaction identifier is defined by the local transaction manager.

`xid_length` [in]
The size in bytes of the buffer for the local transaction identifier.

Description

`tip_pull_async()` creates a local transaction and joins the local transaction with the TIP transaction (the caller becomes a subordinate participant in the TIP transaction). The remote TIP TM is identified via the URL (`*pxid_url`). The local transaction identifier is returned. A call to `tip_pull_async()` returns immediately after the local transaction has been created (before the TIP PULL protocol command is sent). A subsequent call to `tip_pull_complete()` must be issued to check

for successful completion of the pull request.

Return Values

[TIPOK]

The local transaction has been successfully created.

[TIPINVALIDHANDLE]

Invalid connection handle specified.

[TIPNOTCONNECTED]

User has been disconnected from the TIP TM.

[TIPINVALIDURL]

Invalid URL has been supplied.

[TIPTRANSIENT]

Transient error has occurred; retry the operation.

[TIPTRUNCATED]

Insufficient buffer size is specified for the local transaction identifier.

[TIPERROR]

An unexpected error occurred.

- 6) `tip_pull_complete()` - check whether a previous `tip_pull_async()` request has been successfully completed.

Synopsis

```
int tip_pull_complete ([in] tip_handle_t TM,  
                      [in] void *plocal_xid)
```

Parameters

TM [in]

The TIP TM handle.

plocal_xid [in]

Pointer to the local transaction identifier. The structure of the transaction identifier is defined by the local transaction manager.

Description

`tip_pull_complete()` checks whether a previous call to `tip_pull_async()` has been successfully completed. i.e. whether the local transaction has been successfully joined with the TIP transaction. The caller supplies the local transaction identifier returned by the previous call to `tip_pull_async()`. Repeated calls to `tip_pull_complete()` for the same local transaction identifier are idempotent.

Return Values

[TIPOK]

The local transaction has been successfully joined with the TIP transaction.

[TIPINVALIDHANDLE]

Invalid connection handle specified.

[TIPINVALIDXID]

An invalid transaction identifier has been provided.

[TIPNOTPULLED]

Joining of the local transaction with the TIP transaction

has failed. The local transaction has been aborted.
[TIPNOTCONNECTED]
Caller has been disconnected from the TIP TM.
[TIPERROR]
An unexpected error occurred.

- 7) `tip_xid_to_url()` - return a TIP transaction identifier for a local transaction identifier.

Synopsis

```
int tip_xid_to_url ([in] tip_handle_t TM,  
                   [in] void *plocal_xid,  
                   [out] char *pxid_url,  
                   [in] unsigned int url_length)
```

Parameters

TM [in]
The TIP TM handle.

plocal_xid [in]
Pointer to the local transaction identifier. The structure of the transaction identifier is defined by the local transaction manager.

pxid_url [out]
Pointer to the TIP URL of the local transaction. A TIP URL for a transaction takes the form:
TIP://<host>[:<port>]/<transaction identifier>

url_length [in]
The size in bytes of the buffer for the TIP URL.

Description

`tip_xid_to_url()` returns a TIP transaction identifier for a local transaction identifier. The TIP transaction identifier can be passed to remote applications to enable them to do work on the transaction. e.g. to pull the local transaction to the remote node. If a local transaction identifier is not supplied, the caller's current transaction context is used. The constant `TIPURLSIZE` defines the size of a TIP transaction identifier in bytes. This value is implementation specific.

Return Values

[TIPOK]
TIP transaction identifier has been returned.

[TIPNOTCONNECTED]
Caller has been disconnected from the TIP TM.

[TIPNOCURRENTTX]
Process is currently not associated with a transaction (and none was supplied).

[TIPINVALIDXID]
An invalid local transaction identifier has been supplied.

[TIPTRUNCATED]
Insufficient buffer size is specified for the TIP

transaction identifier.
[TIPERROR]
An unexpected error occurred.

- 8) tip_url_to_xid() - return a local transaction identifier for a TIP transaction identifier.

Synopsis

```
int tip_url_to_xid ([in] tip_handle_t TM,  
                   [in] char *pxid_url,  
                   [out] void *plocal_xid,  
                   [in] unsigned int xid_length)
```

Parameters

TM [in]

The TIP TM handle.

pxid_url [in]

Pointer to the TIP URL of the local transaction. A TIP URL for a transaction takes the form:
TIP://<host>[:<port>]/<transaction identifier>

plocal_xid [out]

Pointer to the local transaction identifier. The structure of the transaction identifier is defined by the local transaction manager.

xid_length [in]

The size in bytes of the buffer for the local transaction identifier.

Description

tip_url_to_xid() returns a local transaction identifier for a TIP transaction identifier (note that the local transaction must have previously been created via a tip_push(), or tip_pull (or tip_pull_async()). The constant TIPXIDSIZE defines the size of a local transaction identifier in bytes. This value is implementation specific.

Return Values

[TIPOK]

Local transaction identifier is returned.

[TIPINVALIDURL]

An invalid TIP transaction identifier has been provided.

[TIPTRUNCATED]

Insufficient buffer size is specified for the local transaction identifier.

[TIPERROR]

An unexpected error occurred.

- 9) `tip_get_tm_url()` - get the name of the local TIP transaction manager in TIP URL form.

Synopsis

```
int tip_get_tm_url ([in] tip_handle_t TM,  
                   [out] char *tm_url,  
                   [in] int tm_len);
```

Parameters

`TM[in]`

The TIP TM handle.

`tm_url [in]`

Pointer to the TIP URL of the local transaction manager. A TIP URL for a transaction manager takes the form:

TIP://<host>[:<port>]

`tm_len [out]`

The size in bytes of the buffer for the TIP URL of the local transaction manager.

Description

`tip_get_tm_url()` gets the name of the local transaction manager in TIP URL form (i.e. TIP://<host>[:<port>])

Return Values

[TIPOK]

The name of the local transaction manager has been successfully returned.

[TIPTRUNCATED]

The name of the local transaction manager has been truncated due to insufficient buffer size. Retry the operation with larger buffer size.

Full Copyright Statement

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.

