Linux TIPC 1.7 Programmer's Guide

15 April 2010 [software version: TIPC 1.7.7, tipc-config 1.1.8]

Table of Contents

Introduction

This document is designed to assist software developers who are writing applications that use TIPC.

> *Note:* Many topics discussed in this document are presented in a very concise manner, and presumes the reader is already familiar with many principles of computer networking and socket programming. A basic understanding of IP networking is helpful, but not required. Readers will gain the most benefit from this document by carefully reading (and understanding) how TIPC is used, and by observing the differences between programming using TIPC and programming using IP-based protocols such as TCP and UDP.

For more information about the TIPC protocol, including information about setting up and operating a network that supports TIPC, please consult the open source TIPC project website at http://tipc.sf.net. This site contains TIPC project software, documentation, news, and support instructions.

The TIPC development team welcomes input from the TIPC user community! Feel free to provide feedback on TIPC using the normal TIPC support procedures outlined at http://tipc.sf.net/support.html.

1. TIPC Fundamentals

A brief summary of the major concepts used in TIPC is provided in the following sections.

1.1. TIPC Network Structure

A TIPC network consists of individual processing elements or *nodes*. The network nodes are arranged in a strict hierarchy, according to the following rules:

- A set of related nodes can be grouped into a *cluster* if each node in the cluster has at least one direct (i.e. one-hop) path to every other node in the cluster. Each cluster can have from 1 to 4095 nodes.
- A set of related clusters can be grouped into a *zone* if each cluster in the zone has at least one direct (i.e. one-hop) path to every other cluster in the zone. Each zone can have from 1 to 4095 clusters, which need not all be of the same size.
- A set of related zones can be grouped into a *network* if each zone in the network has at least one direct (i.e. one-hop) path to every other zone in the network. A network can have from 1 to 255 zones, which need not all be of the same size.

Typically the grouping of nodes into clusters and zones is done based on proximity. For example, all nodes in the same shelf or the same room may be assigned to the same cluster, while all clusters in the same building may be assigned to the same zone. In addition, TIPC works best in networks where the expected amount of communication between nodes falls off extrememly rapidly with increasing distance. Thus, in a typical TIPC network, each node will communicate most frequently with other nodes in its own cluster, relatively infrequently with nodes in other clusters in its own zone, and almost never with nodes in other zones.

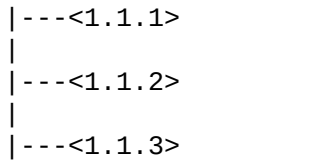
Each node in a TIPC network is assigned a unique *network address* consisting of a zone, cluster, and node identifier, usually denoted <Z.C.N>. Zone identifiers can range from 1 to 255, while cluster and node identifiers can both range from 1 to 4095.

> *Exception:* A TIPC node that has not yet been configured for network access uses network address <0.0.0> until its real network address is assigned.

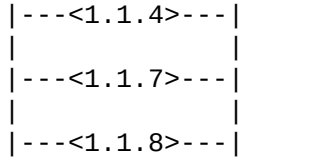
A TIPC network is also assigned a *network identifier*. This allows multiple logical networks to use the same physical medium (for example, the same Ethernet LAN cables) without interfering with one another, since each node in the only recognizes traffic originating from nodes having the same network identifier it has been assigned to use.

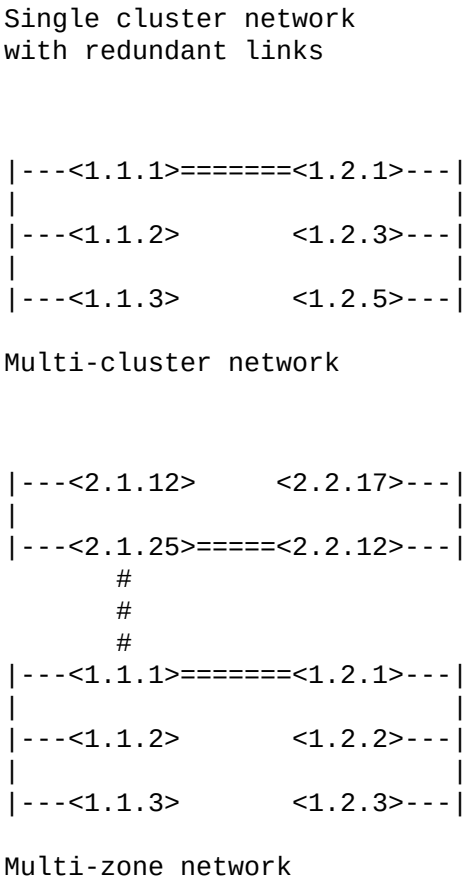
Nodes in a TIPC network communicate with each other by using one or more network interfaces to send and receive messages. Each network interface must be connected to a physical medium that is supported by TIPC (such as Ethernet). When properly configured, TIPC automatically establishes *links* to enable communication with the other nodes in the network, and takes care of routing traffic over the appropriate link, retransmitting messages in the event of errors, etc.

Examples of typical TIPC networks are shown below. Each node is represented by its network address, or <Z.C.N>. Intra-cluster links are denoted using '-', inter-cluster links are denoted by '=', inter-zone links are denoted using '#'.



Single cluster network





THINGS TO REMEMBER:

- TIPC network addressing is *not* like IP network addressing!!! There is only *one* network address per node in TIPC, even if the node has multiple network interfaces. A node's network interfaces are *not* assigned network addresses at all.
- The network administrator takes care of assigning the network address and the network identifier for each node in the network, so programmers don't have to worry about this.
- The network administrator also takes care of configuring each node's network interfaces to enable communication with all other nodes in the network, so programmers don't have to worry about this either.

CURRENT LIMITATIONS:

- Certain capabilities of TIPC cannot be utilized outside a node's cluster or zone. For example, TIPC does not support multicast messaging between clusters, nor does it support any communication using names between zones.
- TIPC does not support the *secondary nodes* concept mentioned in the draft TIPC specification document.

1.2. Messaging Overview

TIPC applications typically communicate with one another by exchanging data units, known as*messages*, between communication endpoints, known as*ports*.

From an application's perspective, a *message* is a byte string from 1 to 66000 bytes long, whose internal structure is determined by the application. A *port* is an entity that can send and receive messages in either a connection-oriented manner or a connectionless manner.

Connection-oriented messaging allows a port to establish a connection to a peer port elsewhere in the network, and then exchange messages with that peer. A connection can be established using an *explicit handshake* mechanism prior to the exchange of any application messages (a variation of the SYN/ACK mechanism used by TCP) or an *implicit handshake* mechanism that occurs during the first exchange of application messages. Once a connection has been established it remains active until it is terminated by one of the ports, or until the communication path between the ports is severed (for example, by the failure of the node on which one of the ports is running); TIPC then immediately notifies the affected port (or ports) that the connection has terminated.

Connectionless messaging allows a port to exchange messages with one or more ports elsewhere in the network. A given message can be sent to a single port (unicast) or to a collection of ports (multicast), depending on the destination address specified when the message is sent.

TIPC is designed to be a reliable messaging mechanism, in which an application can send a message and assume that the message will be delivered to the specified destination as long as that destination is reachable. If a message cannot be delivered the message sender can specify whether it should be returned to its point of origin or discarded.

THINGS TO REMEMBER:

- In a TIPC network containing nodes that are running on different CPU types, or using different operating systems, applications must ensure that the content of their messages use a consistent format. In some cases it may be necessary to do endianness conversion or to force message fields to be a fixed size.
- In some cases, the same conditions that prevent TIPC from delivering a message may also prevent it from returning the message. See the section on "Message Delivery" that appears later in this document for more information.

1.3. TIPC Addressing

TIPC uses 3 distinct forms of addressing within a network.

1.3.1. Network Address

The network address concept (introduced in section 1.1) is used to identify a portion of a TIPC network. It can take any of the following forms:

- <Z.C.N> indicates a network node
- <Z.C.0> indicates a network cluster
- <Z.0.0> indicates a network zone
- <0.0.0> has special meaning, which is operation-specific

When programming, a network address is represented as an unsigned 32-bit value, comprising 3 fields: an 8-bit zone field, a 12-bit cluster field, and a 12-bit node field. Section 1.6 describes routines that can be used to construct and deconstruct networks addresses.

1.3.2. Port Identifier

Each port in a TIPC network has a unique *port identifier* or *port ID*, which is assigned automatically by TIPC when the port is created. The port ID is typically denoted as <Z.C.N:ref>, and consists of the 32-bit

network address of the port's node and a 32-bit *reference value*. The reference value is guaranteed to be unique on a per-node basis and will not be reused for a substantial period of time once the port ceases to exist.

1.3.3. Port Naming

While a TIPC port can send messages to another port by specifying the port ID of the destination port, it is usually more convenient to use a functional address that does not require the sending port to know the physical location of the destination within the network. This simplifies communication when server ports are being created, deleted, or relocated dynamically, or when multiple ports are providing a given service.

The basic unit of functional addressing within TIPC is the *port name*, which is typically denoted as {type,instance}. A port name consists of a 32-bit type field and a 32-bit instance field, both of which are chosen by the application. Often, the type field indicates the class of service provided by the port, while the instance field can be used as a sub-class indicator.

Unlike port IDs, port names do not have to be unique within a TIPC network. Applications can assign a given port name to multiple ports, or assign multiple port names to a given port, or both. Port names can also be unbound from a port when they are no longer required.

Whenever an application binds a port name to a port, it must specify the level of visibility, or *scope*, that the name has within the TIPC network: either *node scope*, *cluster scope*, or *zone scope*. TIPC then ensures that only applications within that portion of the network (i.e. the same node, the same cluster, or the same zone) can access the port using that name.

To simplify the task of specifying a range of similar port name instances TIPC supports the concept of the *port name sequence*, which is typically denoted as {type,lower bound,upper bound}. A port name sequence consists of a 32-bit type field and a pair of 32-bit instance fields, and represents the set of port names from {type,lower bound} through {type,upper bound}, inclusive. The lower bound of a name sequence cannot be larger than the upper bound.

Programmers are given a great deal of freedom in utilizing port names, however there are some limitations in how they can be used.

- Type values from 0 to 63 are reserved by TIPC.

  These names cannot be used by programmers to designate an application service.

- Port names and name sequences are designed for use by server ports, not client ports.

  TIPC does not allow a named sever port to initiate a connection (as if it were a client port), nor does it allow the assignment of names to a connected client port (as if it were a server port).

- TIPC does not allow the creation of partially overlapping port name sequences, unless the name sequences cannot be seen simultaneously.

  For example, if a node has a port with the name sequence {100,1000,2000}, it cannot assign the name sequences {100,500,1200} or {100,1100,1500} to any of its ports (including the port with name sequence {100,1000,2000} itself). However, the node is permitted to assign {100,1000,2000} to other ports, or to use name sequences having other type values such as {150,500,1200}.

  The same limitations exist if the node is simply made aware that another node has a port with a given name sequence. For example, if any node has a port that publishes name sequence {100,1000,2000} with cluster scope, then no other node in that cluster can subsequently create a port with name sequence {100,500,1200} or {100,1100,1500}.

Overlapping name sequences *are* permitted if they are published by different nodes and are published with non-overlapping scopes. For example, it is possible to publish {100,500,1200} on node <1.1.1> and {100,1100,1500} on node <1.1.2> as long as they both are published with node scope.

THINGS TO REMEMBER:

- Programmers typically only have to worry about the selection of TIPC names and name sequences. (Recall that network addresses are chosen by the TIPC network adminstrator, while port ID's are chosen by TIPC automatically.)
- Well-known names (or name sequences) are used in a TIPC network in the same way that well-known port numbers are used in an IP network.
- An application must use TIPC names (or name sequences) that do not conflict with the names used by other applications.

1.4. Using Port Names

This section discusses more advanced aspects of TIPC's functional addressing.

1.4.1. Address Resolution

Whenever an application specifies a port name as the destination address of a message, it must also indicate where within the network TIPC should look to find the destination by specifying a *lookup domain*.

The most commonly used lookup domain is <0.0.0>, which tells TIPC to use a closest-first approach. TIPC first looks on the sending node to find a port having the specified port name; if more than one such port exists, TIPC selects one in a round-robin manner. If the sending node does not contain a matching port, TIPC then looks to all other nodes in the sending node's cluster to see if any ports have published that name using cluster scope or zone scope; again, if more than one such port exists, TIPC selects one in a round-robin manner. Finally, if no matching port is found within the sending node's cluster, TIPC looks at all other nodes in the sending node's zone for ports with a matching name and having zone scope, and selects one in a round-robin manner. (In short, address resolution is performed using 3 lookup domains in succession: first using <Z.C.N>, then <Z.C.0>, and finally <Z.0.0>.) This algorithm results in the message being delivered to a suitable destination as quickly as possible, and also load sharing similarly named messages among all such destinations at the same distance from the sender.

Alternatively, an application can specify a single lookup domain to be used for address resolution. Specifying a lookup domain of the form <Z.C.N>, <Z.C.0>, or <Z.0.0> tells TIPC to take all ports with compatible name and scope values within the specified node, cluster, or zone, respectively, and then select one in a round-robin manner. These forms can be useful in preventing a message from being sent off-node, or for evenly distributing work to all servers scattered throughout a cluster or zone.

It should be noted that the round-robin selection mechanism used by TIPC is shared by all applications using a given node. So, for example, if there are two ports within the specified lookup domain that have the desired port name, an application cannot assume that two successive messages it sends to that name will be distributed one to each port. This is because a similarly named message sent by another application may arrive in the interim, thereby causing the second message sent by the first application to go to the same destination as its first message.

1.4.2. Multicast Messaging

Whenever an application specifies a port name sequence as the destination address of a message (rather than a port name), this instructs TIPC to send a copy of the message to every port in the sender's cluster that has at least one port name within the destination name sequence.

This is most easily illustrated using an example. Suppose a multicast message is sent to {1000,100,200}, then the following ports will each receive exactly one copy of the message:

```
<1.1.10:1234> having {1000,100}                 - one matching name
<1.1.11:4321> having {1000,123} and {1000,175}  - two matching names
<1.1.10:5678> having {1000,150} and {2000,150}  - non-matching name ignored
<1.1.12:5555> having {1000,110,120}             - subset overlap
<1.1.10:8888> having {1000,50,500}              - superset overlap
<1.1.14:9999> having {1000,170,300}             - partial overlap
```

while the following ports will not receive a copy at all:

```
<1.1.10:1111> having {2000,100,200}             - name type mismatch
<1.1.10:4444> having {1000,50,75}               - no overlap
<1.1.10:6666> having no names bound to it       - no overlap
```

Note that a port never receives more than one copy of the multicast message, even if it has several port names (or port name sequences) bound to it that overlap the specified destination name sequence.

Also note that the requirement that the destination address for a multicast message be a name sequence does not prevent applications from multicasting to a single port name; an application can simply specify a name sequence that encompasses a single instance value, such as {1000,123,123}.

THINGS TO REMEMBER:

- Multicast messaging can only be done in a connectionless manner, as TIPC does not support the concept of a one-to-many or many-to-many connection.
- It is not possible to limit the distribution of a multicast message to the ports within a given node by specifying a lookup domain, as can be done with unicast messages.

1.4.3. Name Subscriptions

TIPC provides a network topology service that applications can use to receive information about what port names exist within the application's network zone.

An application accesses the topology service by opening a message-based connection to port name {1,1} and then sending one or more *subscription messages* to the topology service that indicate the port names of interest to the application; in return, the topology service sends *event messages* to the application when these names are published or withdrawn by ports within the network. Applications are allowed to have multiple subscriptions active at the same time; issuing a new subscription does not affect any existing subscription.

A subscription request message must contain the following information:

1. The port name sequence of interest to the application.

   Applications that are interested in a single port name can specify a port name sequence in which the lower and upper instance values are the same.

2. An event filter specifying which events are of interest to the application.

The value `TIPC_SUB_PORTS` causes the topology service to generate a `TIPC_PUBLISHED` event for each port name or port name sequence it finds that overlaps the specified port name sequence; a `TIPC_WITHDRAWN` event is issued each time a previously reported name becomes unavailable. The value `TIPC_SUB_SERVICE` causes the topology service to generate a single publish event for the first port it finds with an overlapping name and a single withdraw event when the last such port becomes unavailable. Thus, the latter event filter allows the topology service to inform the application if there are *any* ports of interest, while the former informs it about *all* such ports.

The special value `TIPC_SUB_CANCEL` can be used to instruct the topology service to cancel a previously requested subscription. The application simply resends the original subscription request with `TIPC_SUB_CANCEL` logically OR'd into the event filter.

3. A subscription timeout value.

   If the subscription is still active after the specified number of milliseconds, a `TIPC_SUBSCR_TIMEOUT` event message is sent to the application and the topology service deletes the subscription. (The value `TIPC_WAIT_FOREVER` can be specified if no time limit is desired.)

4. An 8 byte *user handle* that is application-defined.

   This value is returned to the application as part of all events associated with the subscription request. Applications may find it useful to use this field to hold a unique subscription identifier when multiple subscription requests are active simultaneously.

An event message contains the following information:

1. A code indicating the type of event that has occurred.

   This may be either `TIPC_PUBLISHED`, `TIPC_WITHDRAWN`, or `TIPC_SUBSCR_TIMEOUT`.

2. The instance values denoting the lower and upper bounds of the port name sequence that overlaps the name sequence specified by the subscription.

   The name type value is not supplied as it is always equal to the value specified by the subscription request.

3. The port ID of the associated port.

4. The subscription request associated with the event.

The exchange of messages between application and topology service is entirely asynchronous. The application may issue new subscription requests at any time, while the topology service may send event messages about these subscriptions to the application at any time.

The connection between the application and the topology service continues until the application terminates it, or until the topology service encounters an error that requires it to terminate the connection. When the connection ends, any active subscription requests are automatically cancelled by TIPC.

THINGS TO REMEMBER:

- It is not possible to limit the range of a subscription request to a specific node, cluster, or zone by specifying a lookup domain; the topology service always monitors the requestor's entire zone for matching port names.
- Every node in a TIPC network automatically publishes a port name of the form {0,<Z.C.N>}, where <Z.C.N> is the node's network address. Applications can determine what nodes currently comprise the network, and track the subsequent arrival and departure of nodes from the network, by creating a

subscription that tracks the publication and withdrawl of names for type 0. The dummy subscription example in section 5.1 below illustrates this technique.

1.5. Message Delivery

On the surface, message delivery in TIPC is a simple series of steps: a message is created by a sender, TIPC carries it to the specified destination, and the receiver consumes the message. And, in practice, this is exactly what happens most of the time. However, there are a number of places along the way where things can get complicated, and in these cases it is important for application designers to understand exactly what TIPC will do.

The sections that follow describe the various steps performed by TIPC during the exchange of a unicast message; the final section outlines how any differences that occur when dealing with a multicast message.

1.5.1. Message Creation

The first step in sending a message is to create it. The most common reason TIPC is unable to create a message is because the sender passes in one or more invalid arguments to the send routine. The term "invalid" refers both to values that are never acceptable under any circumstances (such as specifying a message length greater than 66000 bytes) and to values that are not acceptable for the current sender (such as requesting a send operation on a socket that has been turned into a listening socket).

Other reasons that TIPC may be unable to create a message:

- There are no more message buffers available that TIPC can use.
- The link TIPC selected to carry the message to its destination was congested and the sender did not want to block until the congestion cleared (see 1.5.3 below).
- The peer socket on a connection was congested (i.e. had too many unconsumed messages in its receive queue) and the sender did not want to block until the congestion cleared (see 1.5.6 below).

In all of these cases the send operation will return a failure code indicating that the intended message was not sent. If the message is created successfully the send operation returns a success indication.

THINGS TO REMEMBER:

- If the sender specifies a destination address that does not currently exist within the TIPC network, TIPC does *not* treat this as an invalid send request (i.e. it's not the sender's fault that the destination doesn't exist). Instead TIPC creates the message and then "rejects" it because it is undeliverable (see 1.5.6 below). The return value for the send operation will indicate success since the message was successfully created and processed by TIPC.

1.5.2. Source Routing

Once a message has been created, TIPC then determines what node the message should be sent to. If the specified destination address is a port ID, the destination node is pre-determined; if the address is a port name, TIPC performs a name table lookup to select a port (see 1.4.1 above), and then uses the node associated with that port. The message is then passed to a link for off-node transmission (see 1.5.3 below) or is handed off to the destination port directly if it is on the same node as the sender (see 1.5.5 below).

Problems that can arise during the source routine phase of message delivery:

- No matching port can be located during a name table lookup when sending by port name.
- No working link to the specified destination node can be found when sending by port ID.

In all cases of source routing failure, the message is rejected (see section 1.5.6 below).

THINGS TO REMEMBER:

- A message that specifies a port name and is sent off-node may not actually end up going to the port selected during the name table lookup, since the destination node will perform a second name table lookup when it receives the message (see 1.5.4 below).

1.5.3. Link Transmission

Once a message is given to a link for transmission to another node, the link will normally deliver the message to that node even if problems arise. For example, the link will automatically detect lost messages and retransmit them, or will re-route messages over an alternate link if it loses contact with its peer link endpoint on the other node. Such error recovery is possible because TIPC keeps a copy of each outgoing message in a transmit queue until it is notified that the message has been successfully received by the peer link endpoint.

If a link endpoint's transmit queue grows too large because the peer link endpoint falls behind in acknowledging the successful arrival of messages (typically around 50 messages), TIPC declares "link congestion" on that link. When a link becomes congested, the link only accepts a new message for transmission if it is important enough (i.e. the more important the message, the longer the queue is allowed to be).

Whenever a message cannot be sent because of link congestion, TIPC checks the "source droppable" setting of the sending port. If the setting is enabled (indicating that the message is being sent in an unreliable manner) TIPC discards the message, but provides no indication of this to the sender. If the source droppable setting is disabled (which is the default case), TIPC will normally block the sending application until the congestion clears, and then resume the send operation; however, if the application has requested a non- blocking send, the application will not block when link congestion occurs and the send operation returns a failure indication.

In the event that a link to a destination node fails and there are no other links available that can be used to re-route traffic, any messages in the link's transmit queue are simply discarded. The messages are *not* rejected (and potentially returned to their originating ports) because TIPC does not know whether or not they were successfully delivered.

1.5.4. Destination Routing

Once a message arrives at the specified destination node over a link, TIPC then determines what port it should be sent to on that node.

If the specified destination address is a port ID, the destination port is pre-determined; if no such port exists the message is considered undeliverable and rejected (see 1.5.6 below).

If the destination address is a port name, TIPC performs a name table lookup and selects a port (see 1.4.1 above). If no such port exists TIPC repeats the source routing operation and tries to send the message to another node; if no such node can be found, or if the message has been previously re-routed too many times, the message is considered undeliverable and rejected (see 1.5.6 below).

1.5.5. Message Consumption

When a message (finally!) reaches the destination port it is either consumed immediately (if the controlling application is using the native API) or added to a receive queue (if the controlling application is using the socket API). In the latter case, the message typically remains in the socket's receive queue until it is received by the application that owns the socket. Queued messages are consumed by the application in

a FIFO manner, and once the contents of a message have been passed to the application the message is discarded.

If an application terminates access to the socket (using either the `close()` or `shutdown()` APIs) before all messages in the receive queue are consumed, all unconsumed messages are considered undeliverable and are rejected (see 1.5.6 below).

It is very important that TIPC applications be engineered to consume their incoming messages at a rate that prevents them from accumulating in large numbers in any socket receive queue. Failure to do so can result in TIPC declaring either "port congestion" or "socket congestion".

Port congestion can occur once more than 512 messages have accumulated in the receive queue of a connection-oriented socket. Once this is detected, TIPC may block the peer socket from sending messages until the congestion clears (see 1.5.1 above) or, if the sender is sending in an unreliable manner, cause such messages to be discarded (see 1.5.3 above).

Socket congestion can occur once TIPC detects that too many unreceived messages exist on a node or on an individual socket. More precisely, a node can have up to 5000 messages sitting in socket receive queues before congestion handling kicks in; once this happens, low importance messages will be rejected (see 1.5.6 below) but higher importance messages will continue to be accepted. Medium importance messages get screened out once the number of pending messages hits 10000, and high priority messages at 500,000 messages; critical priority messages are always accepted. Similar congestion handling occurs on a per-socket basis, but the thresholds are one half the global threshold values (i.e. at 2500, 5000, and 250,000).

Since the impact of socket congestion is more significant for a connection- oriented socket than port congestion (i.e. it terminates the connection), the smaller port congestion threshold has been chosen so that it will normally kick in first and prevent the socket receive queue from growing larger. However, the existence of the per-node socket congestion threshold means that it is possible for socket congestion to occur before port congestion occurs.

> *NOTE:* These message congestion thresholds may be more configurable in future releases of TIPC since it's not really realistic to have a one-size-fits-all solution that will work well on a wide variety of hardware configurations (i.e. a resource-constrained DSP will probably need lower thresholds than a resource- rich Linux box).

1.5.6. Message Rejection

When a message is *rejected* because it cannot be delivered, TIPC checks the message's *destination droppable* setting to see what the sender wanted done with the message.

- If the destination droppable setting is enabled, TIPC silently discards the message.
- If the destination droppable setting is disabled, TIPC attempts to send the *returned message* back to the message originator. Messages less than or equal to 1024 bytes are returned in their entirety, while longer messages are truncated to 1024 bytes; an error code is also associated with each returned message to allow the sender to determine why the message was returned. In the case of a connection-oriented message, the return of an undeliverable message also causes the connection to be terminated at both ends.

TIPC's socket API has been designed so that applications that don't want to concern themselves with returned messages can easily ignore them; this approach simplifies the job of porting applications written for TCP or UDP to use TIPC. However, the ability for a sending application to examine returned messages can sometimes be helpful in debugging problems during the design and testing of a new TIPC application.

- By default, all undeliverable messages sent by a connectionless socket are discarded. If desired, an application can request the return of undelverable messages by disabling the sending socket's destination droppable option.

- By default, the first undeliverable message sent using a connection-oriented socket is returned, and helps to break the connection. If desired, an application can request that undelverable messages be silently discarded by enabling the sending socket's destination droppable option.

To ensure that an application can distinguish between the arrival of a normal message and the return of an undeliverable message, TIPC's socket API utilizes the following conventions.

- If an application only needs to be aware of the arrival of a returned message, this is typically done by examining the return value of `recv()` or `recvfrom()`. A connectionless socket indicates the arrival of a returned message by returning a size of zero (which can never occur for normal messages), while a connection-oriented socket returns a size of -1 (which indicates that connection was broken abnormally, which can happen for a variety of reasons).

- If an application wants to examine the error code of a returned message, its content, or its intended destination, it *must* use `recvmsg()` to obtain this information through ancillary data. In the case of a connection-oriented socket, the return value of `recvmsg()` is 0 rather than -1, which means that the application must examine the error code to determine if it has actually received a returned message (since this value is also returned when the connection is voluntarily shut down at the far end).

THINGS TO REMEMBER:

- The returned message capability of TIPC must *not* be used by a sending application to try to determine what messages were successfully consumed by the receiving application! While the return of a message indicates that the receiver did not consume the message, the non-return a message does not indicate that is was successfully consumed. For example, if a destination node suffers a power failure, TIPC will be unable to return any messages that are sitting unprocessed in a socket receive queue; likewise, if a message cannot be delivered because of congestion within the TIPC network, this same congestion may also prevent TIPC from returning the message to the originator. The only way for a sending application to know that a message was consumed is for it to receive an explicit acknowledgement message generated by the receiving application.

1.5.7. Multicast Message Delivery

Multicast message creation is done the same way as for unicast messages, but since multicasting is always done in a connectionless manner it is not possible for peer port congestion to occur.

Multicast source routing always involves a name table lookup of a port name sequence. If no port within the cluster overlaps the specified name sequence the message is simply discarded. Otherwise, TIPC sends a copy of the message to each overlapping port on the sending node and also determines if any off-node ports have an overlap; if there is at least one such port then the messsage is passed to a special broadcast link.

The broadcast link operates much like a regular unicast link, except that it sends its messages to *all* nodes in the cluster rather than just one, and has a smaller congestion threshold (around 20 messages).

Whenever the broadcast link delivers the message to a node, TIPC repeats the name table lookup and sends a copy of the message to all overlapping ports it finds on that node; if there are no such ports the message is discarded. Once a multicast message arrives at a destination port, it is treated just like a unicast message and is subject to the same socket congestion and message rejection handling.

THINGS TO REMEMBER:

- TIPC currently does not permit an application to send a multicast message with the "destination droppable" setting disabled. Consequently, TIPC will never try to return an undeliverable multicast message to its sender.

1.6. Routines

The following utility routines are available to programmers:

tipc_addr() - combine zone, cluster, and node numbers into a TIPC address
tipc_cluster() - extract cluster number from a TIPC address
tipc_node() - extract node number from a TIPC address
tipc_zone() - extract zone number from a TIPC address

Further information about each of these routines is provided in the following sections.

1.6.1. tipc_addr()

```
unsigned u32 tipc_addr(unsigned int zone, unsigned int cluster, unsigned int n
```

This routine takes individual zone, cluster, and node identifiers and combines them into a 32-bit TIPC network address.

1.6.2. tipc_cluster()

```
unsigned int tipc_cluster(u32 addr)
```

This routine takes a 32-bit TIPC network address and returns the cluster identifier contained in the address.

1.6.3. tipc_node()

```
unsigned int tipc_node(u32 addr)
```

This routine takes a 32-bit TIPC network address and returns the node identifier contained in the address.

1.6.4. tipc_zone()

```
unsigned int tipc_zone(u32 addr)
```

This routine takes a 32-bit TIPC network address and returns the zone identifier contained in the address.

2. Socket API

The TIPC socket API allows programmers to access the capabilities of TIPC using the well-known socket paradigm.

IMPORTANT: TIPC does *not* support all socket API capabilities available with other socket-based protocols; it also introduces some new capabilities by adapting the socket API to accommodate them. Programmers who have used a socket API with other protocols are strongly advised to read this section carefully to ensure they understand where the TIPC socket API differs from their previous experiences.

2.1. Routines

The following socket API routines are supported by TIPC:

accept() - accept a new connection on a socket
bind() - bind or unbind a TIPC name to the socket
close() - close the socket
connect() - connect the socket
getpeername() - get the port ID of the peer socket
getsockname() - get the port ID of the socket
getsockopt() - get the value of an option for the socket
listen() - listen for socket connections
poll() - monitor input/output on multiple sockets
recv() - receive a message from the socket
recvfrom() - receive a message from the socket
recvmsg() - receive a message from the socket
select() - monitor input/output on multiple sockets
send() - send a message on the socket
sendmsg() - send a message on the socket
sendto() - send a message on the socket
setsockopt() - set the value of an option for the socket
shutdown() - shut down socket send and receive operations
socket() - create an endpoint for communication

Further information about each of these routines can be found in the following sections.

2.1.1. accept()

```
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen)
```

Accepts a new connection on a socket.

THINGS TO NOTE:

- If non-`NULL`, `cliaddr` is set to the port ID of the peer socket. This info can be used to perform basic validation of the connection requestor's identity (eg. disallow connections originating from certain network nodes).

2.1.2. bind()

```
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen)
```

Binds or unbinds a TIPC name (or name sequence) to the socket. A bind operation is requested by setting `myaddr->scope` to `TIPC_NODE_SCOPE`, `TIPC_CLUSTER_SCOPE`, or `TIPC_ZONE_SCOPE`, as appropriate. An unbind operation is requested by setting `myaddr->scope` to arithmetic inverse of the scope used when the name was bound (eg. `-TIPC_NODE_SCOPE`). Specifying zero for `addrlen` unbinds all names and name sequences currently bound to the socket.

THINGS TO NOTE:

- It is legal to bind more than one TIPC name or name sequence to a socket.

- If a socket is connected to a peer it cannot use `bind()` to bind additional TIPC names to itself since it is unable to serve any other clients.
- `bind()` cannot be used to changed the port ID of a socket.

2.1.3. close

```
int close(int sockfd)
```

Closes the socket.

THINGS TO NOTE:

- Any unprocessed messages remaining in the socket's receive queue are rejected (i.e. discarded or returned), as appropriate.

2.1.4. connect()

```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen)
```

Attempts to make a connection on the socket using TIPC's explicit handshake mechanism.

THINGS TO NOTE:

- If `servaddr` is set to a TIPC name (but not a TIPC name sequence or a TIPC port ID), the `addr.name.domain` field can be used to affect the name lookup process. The `scope` field of `servaddr` is ignored by `connect()`.
- If a socket has a name bound to it, it is considered to be a server and cannot use `connect()` to initiate a connection as a client.
- TIPC does not support the use of `connect()` with connectionless sockets. (POSIX non-conformity)
- TIPC does not support the non-blocking form of `connect()`; use `sendto()` to establish connections using implicit handshaking to achieve this effect. (POSIX non-conformity)

2.1.5. getpeername()

```
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen)
```

Gets the port ID of the peer socket.

THINGS TO NOTE:

- This routine cannot be used to determine the TIPC names or name sequences that have been bound to the peer socket.

2.1.6. getsockname()

```
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen)
```

Gets the port ID of the socket.

THINGS TO NOTE:

- This routine cannot be used to determine the TIPC names or name sequences that have been bound to the socket.

2.1.7. getsockopt()

```
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen)
```

Get the current value of a socket option. Currently, the following values of `optname` are supported when `level` is `SOL_TIPC`:

- `TIPC_CONN_TIMEOUT`

  See `setsockopt()` below.

- `TIPC_DEST_DROPPABLE`

  See `setsockopt()` below.

- `TIPC_IMPORTANCE`

  See `setsockopt()` below.

- `TIPC_NODE_RECVQ_DEPTH`

  This option indicates the number of messages in the receive queues of all TIPC sockets on the node.

  This option is a read-only value that cannot be altered using `setsockopt()`.

- `TIPC_SOCK_RECVQ_DEPTH`

  This option indicates the number of messages in the receive queue of the associated socket.

  This option is a read-only value that cannot be altered using `setsockopt()`.

- `TIPC_SRC_DROPPABLE`

  See `setsockopt()` below.

THINGS TO NOTE:

- TIPC does not currently support many socket options for level `SOL_SOCKET`, such as `SO_SNDBUF`. Options that are supported include `SO_RCVTIMEO` (for all socket types) and `SO_RCVLOWAT` (for `SOCK_STREAM` only).
- TIPC does not currently support socket options for level `IPPROTO_TCP`, such as `TCP_MAXSEG`. Attempting to get the value of these options on a `SOCK_STREAM` socket returns the value 0.

2.1.8. listen()

```
int listen(int sockfd, int backlog)
```

Enables a socket to listen for connection requests.

THINGS TO NOTE:

- The `backlog` parameter is currently ignored.

2.1.9. poll()

```
int poll(struct pollfd *fdarray, unsigned long nfds, int timeout)
```

Indicates the readiness of the specified TIPC socket(s) for I/O operations, using the standard `poll()` mechanism.

TIPC currently sets the returned event flags as follows:

- `POLLRDNORM` and `POLLIN` are set if the socket's receive queue is non-empty. They are also set for a connection-oriented socket if it is not connected (i.e. if it has never been connected, or is now disconnected).
- `POLLOUT` is set if it is not in a transmit congested state (i.e. is not blocked trying to send over a congested link or to a congested peer).
- `POLLHUP` is set if a socket's connection has been terminated.

THINGS TO NOTE:

- `poll()` does *not* guarantee that the associated I/O operation will be successful; it typically indicates only that the operation will not block (see next point).
- The `POLLOUT` event flag does not guarantee that a send operation performed on the socket will not block, since transmit congestion management is partially done on a per-link basis, and the state of the link used by the send cannot be determined at the time `poll()` returns. The only way to ensure that an application is not blocked during a send operation is to use the MSG_DONTWAIT flag or set the socket for nonblocking I/O using `fcntl()`.
- A socket that is connecting asynchronously is considered writeable, since attempting a second send operation during an implied connection setup will immediately fail. (POSIX non-conformity)

2.1.10. recv()

```
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags)
```

Attempts to receive a message from the socket.

THINGS TO NOTE:

- When used with a connectionless socket, a return value of 0 indicates the return of an undelivered data message that was originally sent by this socket.

- When used with a connection-oriented socket, a return value of 0 or -1 indicates connection termination. A value of 0 indicates that the connection was terminated by the peer using `shutdown()`; connection termination by any other means causes a return value of -1.
- Applications can determine the exact cause of connection termination and/or message non-delivery by using `recvmsg()` instead of `recv()`.
- TIPC supports the `MSG_PEEK` flag when receiving, as well as the `MSG_WAITALL` flag when receiving on a `SOCK_STREAM` socket; all other flags are ignored.

2.1.11. recvfrom()

```
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen)
```

Attempts to receive a message from the socket. If successful, the port ID of the message sender is returned in `from`.

THINGS TO NOTE:

- See the comments section for `recv()`.

2.1.12. recvmsg()

```
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)
```

Attempts to receive a message from the socket. If successful, the port ID of the message sender is captured in the `msg_name` field of `msg` (if non-`NULL`) and ancillary data relating to the message is captured in the `msg_control` field of `msg` (if non-`NULL`).

The following ancillary data objects may be captured:

- `TIPC_ERRINFO` - The TIPC error code associated with a returned data message or a connection termination message, and the length of the returned data. (8 bytes: error code + data length)
- `TIPC_RETDATA` - The contents of a returned data message, up to a maximum of 1024 bytes.
- `TIPC_DESTNAME` - The TIPC name or name sequence that was specified by the sender of the message. (12 bytes: type + lower instance + upper instance; the latter two values are the same for a TIPC name, but may differ for a name sequence)

Each of these objects is only created where relevant. For example, receipt of a normal data message never creates the `TIPC_ERRINFO` and `TIPC_RETDATA` objects, and only creates the `TIPC_DESTNAME` object if the message was sent using a TIPC name or name sequence as the destination rather than a TIPC port ID. Those objects that are created will always appear in the relative order shown above.

If ancillary data objects capture is requested (i.e. `msg->msg_control` is non-`NULL`) but insufficient space is provided, the `MSG_CTRUNC` flag is set to indicate that one or more available objects were not captured.

THINGS TO NOTE:

- When used with a connectionless socket, a return value of 0 indicates the arrival of a returned data message that was originally sent by this socket.
- When used with a connection-oriented socket, a return value of 0 or -1 indicates connection termination. The exact return value upon connection termination is influenced by the `msg_control` field of `msg`. If `msg_control` is `NULL`, a return value of 0 indicates that the connection was terminated by the peer using `shutdown()`; connection termination by any other means causes a return value of -1. If `msg_control` is non-`NULL`, a return value of 0 is always used; the application must examine the `TIPC_ERRINFO` object to determine if the connection was explicitly terminated by the peer. (POSIX non-conformity)
- When used with connection-oriented sockets, `TIPC_DESTNAME` is captured for each data message received by the socket if the connection was established using a TIPC name or name sequence as the destination address. Note: There is currently no way for the destination socket to capture `TIPC_DESTNAME` following `accept()` until the originator sends a data message.
- TIPC supports the `MSG_PEEK` flag when receiving, as well as the `MSG_WAITALL` flag when receiving on a `SOCK_STREAM` socket; all other flags are ignored.

2.1.13. select()

```
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timevale *timeout)
```

Indicates the readiness of the specified TIPC socket(s) for I/O operations, using the standard `select()` mechanism.

TIPC currently determines readiness as follows:

- A socket is considered "readable" if its receive queue is non-empty. A connection-oriented socket is also readable if the socket is not connected (i.e. if it has never been connected, or is now disconnected).
- A socket is considered "writeable" if it is not in a transmit congested state (i.e. is not blocked trying to send over a congested link or to a congested peer). A connection-oriented socket is also writeable if the socket's connection has been terminated.

THINGS TO NOTE:

- `select()` does *not* guarantee that the associated I/O operation will be successful; it typically indicates only that the operation will not block (see next point).
- The results of `select()` do not guarantee that a send operation performed on the socket will not block, since transmit congestion management is partially done on a per-link basis, and the state of the link used by the send cannot be determined at the time `select()` returns. The only way to ensure that an application is not blocked during a send operation is to use the `MSG_DONTWAIT` flag or set the socket for nonblocking I/O using `fcntl()`.
- A socket that is connecting asynchronously is considered writeable, since attempting a second send operation during an implied connection setup will immediately fail. (POSIX non-conformity)

2.1.14. send()

```
ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags)
```

Attempts to send a message from the socket to its peer socket.

THINGS TO NOTE:

- `send()` should not be used until a connection has been fully established using either explicit or implicit handshaking.
- TIPC supports the `MSG_DONTWAIT` flag when sending; all other flags are ignored.

2.1.15. sendmsg()

```
ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags)
```

Attempts to send a message from the socket to the specified destination. If the destination is denoted by a TIPC name or a port ID the message is unicast to a single port; if the destination is denoted by a TIPC name sequence the message is multicast to all ports having a TIPC name or name sequence that overlaps the destination name sequence.

THINGS TO NOTE:

- See the comments section for `sendto()`.
- TIPC does not currently support the use of ancillary data with `sendmsg()`.

2.1.16. sendto()

```
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
               const struct sockaddr *to, socklen_t addrlen)
```

Attempts to send a message from the socket to the specified destination. If the destination is denoted by a TIPC name or a port ID the message is unicast to a single port; if the destination is denoted by a TIPC name sequence the message is multicast to all ports having a TIPC name or name sequence that overlaps the destination name sequence.

THINGS TO NOTE:

- If the destination address is a TIPC name the `addr.name.domain` field indicates the search domain used during the name lookup process. (In contrast, if the destination address is a TIPC name sequence the default closest-first algorithm is always used; if it is a TIPC port ID no name lookup occurs.) The `scope` field of the destination address is always ignored when sending.
- TIPC supports the `MSG_DONTWAIT` flag when sending; all other flags are ignored.
- A connection-oriented socket that is unconnected can initiate connection establishment using implicit handshaking by simply sending a message to a specified destination, rather than using `connect()`. However, the connection is not fully established until the socket successfully receives a message sent by the destination using `recv()`, `recvfrom()`, or `recvmsg()`.

2.1.17. setsockopt()

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen)
```

Sets a socket option to the specified value. Currently, the following values of `optname` are supported when `level` is `SOL_TIPC`:

- `TIPC_CONN_TIMEOUT`

  This option specifies the number of milliseconds `connect()` will wait before aborting a connection attempt because the destination has not responded. By default, 8000 (i.e. 8 seconds) is used.

  This option has no effect when establishing connections using `sendto()`.

- `TIPC_DEST_DROPPABLE`

  This option governs the handling of messages sent by the socket if the message cannot be delivered to its destination, either because the receiver is congested or because the specified receiver does not exist. If enabled, the message is discarded; otherwise the message is returned to the sender.

  By default, this option is disabled for `SOCK_SEQPACKET` and `SOCK_STREAM` socket types, and enabled for `SOCK_RDM` and `SOCK_DGRAM`, This arrangement ensures proper teardown of failed connections when connection-oriented data transfer is used, without increasing the complexity of connectionless data transfer.

- `TIPC_IMPORTANCE`

  This option governs how likely a message sent by the socket is to be affected by congestion. A message with higher importance is less likely to be delayed or dropped due to link congestion, and also less likely to be rejected due to receiver congestion. The following values are defined: `TIPC_LOW_IMPORTANCE`, `TIPC_MEDIUM_IMPORTANCE`, `TIPC_HIGH_IMPORTANCE`, and `TIPC_CRITICAL_IMPORTANCE`.

  By default, `TIPC_LOW_IMPORTANCE` is used for all TIPC socket types.

- `TIPC_SRC_DROPPABLE`

  This option governs the handling of messages sent by the socket if link congestion occurs. If enabled, the message is discarded; otherwise the system queues the message for later transmission.

  By default, this option is disabled for `SOCK_SEQPACKET`, `SOCK_STREAM`, and `SOCK_RDM` socket types (resulting in "reliable" data transfer), and enabled for `SOCK_DGRAM` (resulting in "unreliable" data transfer).

THINGS TO NOTE:

- TIPC does not currently support many socket options for level `SOL_SOCKET`, such as `SO_SNDBUF`. Options that are supported include `SO_RCVTIMEO` (for all socket types) and `SO_RCVLOWAT` (for `SOCK_STREAM` only).
- TIPC does not currently support socket options for level `IPPROTO_TCP`, such as `TCP_MAXSEG`. Setting these options on a `SOCK_STREAM` socket has no effect.

2.1.18. shutdown()

```
int shutdown(int sockfd, int howto)
```

Shuts down socket send and receive operations on a connection-oriented socket. The socket's peer is notified that the connection was deliberately terminated by the application (by means of the `TIPC_CONN_SHUTDOWN` error code), rather than as the result of an error.

THINGS TO NOTE:

- Applications should normally call `shutdown()` to terminate a connection before calling `close()`.
- Applications must set `how` to `SHUT_RDWR`, to terminate both reading and writing. TIPC does not support partial shutdown of a connection.
- A socket that has been `shutdown()` cannot be re-used for a new connection; this prevents any "stale" incoming messages from an earlier connection from interfering with the new connection.

2.1.19. socket()

```
int socket(int family, int type, int protocol)
```

Creates an endpoint for communication.

TIPC supports the following values for `type`:

- `SOCK_DGRAM` - for unreliable connectionless messages
- `SOCK_RDM` - for reliable connectionless messages
- `SOCK_SEQPACKET` - for reliable connection-oriented messages
- `SOCK_STREAM` - for reliable connection-oriented byte streams

THINGS TO NOTE:

- The `family` parameter should always be set to `AF_TIPC`.
- The `protocol` parameter should always be set to 0.

2.2. Examples

A variety of demo programs can be found at http://tipc.sf.net , which may be useful in understanding how to write an application that uses TIPC.

3. Native API

TIPC's native API is an alternative to the socket API that can be used by applications running in the operating system kernel.

Benefits of the native API:

1. Can achieve significantly faster execution than socket API in some cases.
2. Can reduce object code size by excluding socket API code from system.

Limitations of the native API:

1. Not available to user-space applications.

2. Does not provide all capabilities of the socket API.

3. Typically more difficult for programmers to write programs using the native API than the socket API.

4. Some routines are not reentrant in the current (TIPC 1.7.6) implementation.

3.1. Key Concepts

There are a number of important conceptual differences between programming with the native API and programming with the socket API. Understanding these concepts is an essential pre-requisite to using the native API effectively.

3.1.1. Using Ports

Applications using the native API have direct access to TIPC ports, rather than accessing them indirectly through the socket API. Because a TIPC port is the most basic communication endpoint in TIPC, applications using the native API are required to deal with a number of aspects of the TIPC protocol that are hidden by the socket API, including handling undeliverable messages that are returned to the sending port and managing the handshaking required to set up and tear down port-to-port connections.

Applications create a port using the native API using `tipc_createport()`. At the time of port creation the application typically specifies a number of callback routines that are used by TIPC to tell the application when certain events involving the port arise. If the port is successfully created TIPC sets one of the arguments of `tipc_createport()` to the port's unique reference value, which is analogous to a socket's file descriptor.

> *Note:* All native API routines that subsequently need to access a TIPC port must use the port reference value to identify the port, rather than a pointer to the actual port data structure; this allows TIPC to gracefully handle cases where an application inadvertently attempts to utilize a port that no longer exists.

Once created, a TIPC port typically continues to exist until the application calls `tipc_deleteport()` to destroy it. (See Section 3.1.4, “User Registration” for information on an alternative method for destroying ports.)

To make the programmer's life easier, TIPC's native API ensures that most operations on a port are multithread-safe by automatically locking each port's data structure during critical sections; however, there are some exceptions that are noted in the following sections.

3.1.2. Sending Messages

Applications send messages using the native API using a routine of the form `tipc_send()`, `tipc_sendXXX()`, or `tipc_multicast()`, depending on whether connection-based, connectionless, or multicast messaging is being performed.

> *Note:* The native API does not support byte stream-type communication between ports (equivalent to that provided by the socket API using SOCK_STREAM).
>
> *WARNING:* Native API send routines *are not multithread-safe!* Applications must ensure that two or more threads do not attempt to send messages using the same TIPC

port at the same time, nor should one thread attempt to send a message while another thread is changing the configuration of the port, as unpredictable results may occur.

The data portion of a message is normally specified as a set of one or more byte arrays (using the "iovec" structure), but it can also be a socket buffer (using the "sk_buff" structure); in either case, the data must not exceed TIPC's 66000 byte limit on message size.

Note: The latter form can improve performance by eliminating the need for TIPC to copy the data into a socket buffer, but for best results the application that creates the buffer should reserve enough headroom to allow a TIPC message header and data link header to be prepended easily.

If a native API send routine returns a positive value this indicates that the message has been accepted by TIPC for delivery. TIPC will then add and strip message headers, deal with message fragmentation and reassembly, retransmit lost messages, reorder incoming messages and discard duplicates, and so on, wherever these operations are required during the course of message delivery.

If a native API send routine returns `-ELINKCONG` this indicates that the message could not be sent because of congestion, and that the application should try again later. TIPC allows the application that creates a native API port to specify a routine to be invoked when congestion abates, which can be used to trigger the resend operation, if desired.

If a native API send routine returns a negative value other than `-ELINKCONG` this indicates that one of the arguments supplied by the application is invalid.

3.1.3. Receiving Messages

The native API does not provide any synchronous mechanism for receiving messages sent to a port, so there is no equivalent of the socket API's `recv()`, `recvfrom()`, or `recvmsg()` routines.

Note: If desired, an application can emulate a synchronous receive capability using the technique outlined in Section 3.3.2, “Synchronous message receive” .

Each time a message is received by a port TIPC invokes one of the callback routines that was passed to `tipc_createport()` when the port was created. Individual callback routines allow the application to handle:

- a direct message (i.e. a message sent to a port ID)
- a named message (i.e. a message sent to a port name or name sequence)
- a connection message (i.e. a message sent on an established connection)
- an errored direct message (i.e. a direct message that was returned)
- an errored named message (i.e. a named message that was returned)
- an errored connection message (i.e. a connection message that was returned)

An application's callback routine executes in a TIPC kernel thread, rather than an application thread, so the programmer must take care to handle any critical section issues that arise between threads. (Alternatively, the application can emulate a synchronous receive to avoid issues caused by multithreading.)

TIPC only requires an application to supply callback routines for the types of messages that the port expects to handle. If TIPC receives a message for which the callback routine is `NULL`, the message is automatically rejected or discarded, as appropriate.

3.1.4. User Registration

Any application that utilizes TIPC's native API can optionally become a registered user by calling `tipc_attach()` and obtaining a "user identifier". There are two instances where this capability may prove useful.

- TIPC can simplify the work involved in terminating a kernel-based application by deleting a registered application's ports automatically when the application ends. This feature allows the application to utilize a constantly changing set of ports without requiring it to keep track of which ports it currently owns. To use this capability, the application needs to specify its user identifier each time it creates a port using `tipc_createport()`. When the application later terminates, it need only deregister itself using `tipc_detach()` and TIPC will delete all of the ports currently owned by the specified user.

  > *Note:* An application that does not require automatic port deletion can create anonymous ports by specifying using a user identifier of 0.

- Registration can simplify synchronization issues when a TIPC node starts up by allowing a registered application to delay its startup until TIPC is sufficiently initialized. TIPC notifies each registered application whenever it changes from "not running" to "standalone" mode, or from "standalone" mode to "networking" mode, by invoking the callback routine specified by the application when it registers with TIPC.

3.2. Routines

The following native API routines are supported by TIPC. Information about the arguments and return value of each routine can be found by looking at the function prototypes in `tipc.h`.

Unfortunately, a comprehensive description of each routine is not currently available. (Feel free to contribute one!) Consult Section 3.3, “Examples” (or the source code for TIPC) to learn more about what each routine does and how to use it.

> *WARNING!* The native API is still under development at this time and has not been finalized. Expect changes in future versions of TIPC.

```
/* TIPC port manipulation routines */

tipc_createport()           - create a TIPC port & generate reference
tipc_deleteport()           - delete a TIPC port & obsolete reference
tipc_ref_valid()            - determine if port reference is valid
tipc_ownidentity()          - get port ID of port
tipc_set_portimportance()   - set port traffic importance level
tipc_portimportance()       - get port traffic importance level
tipc_set_portunreliable()   - set port traffic "source droppable" setting
tipc_portunreliable()       - get port traffic "source droppable" setting
tipc_set_portunreturnable() - set port traffic "destination droppable" setting
tipc_portunreturnable()     - get port traffic "destination droppable" setting
tipc_publish()              - bind name/name sequence to port
tipc_withdraw()             - unbind name/name sequence from port
tipc_connect2port()         - associate port with peer
tipc_disconnect()           - disassociate port with peer
tipc_shutdown()             - shut down connection to peer & disassociate
tipc_isconnected()          - determine if port is currently connected
tipc_peer()                 - get port ID of peer port
```

```
/* TIPC messaging routines */

tipc_send()             - send iovec(s) on connection
tipc_send_buf()         - send sk_buff on connection
tipc_send2name()        - send iovec(s) to port name
tipc_send_buf2name()    - send sk_buff to port name
tipc_send2port()        - send iovec(s) to port ID
tipc_send_buf2port()    - send sk_buff to port ID
tipc_multicast()        - multicast iovec(s) to port name sequence

tipc_forward2name()     - [may be obsoleted]
tipc_forward_buf2name() - [may be obsoleted]
tipc_forward2port()     - [may be obsoleted]
tipc_forward_buf2port() - [may be obsoleted]

/* TIPC subscription routines */

tipc_ispublished()     - determines if a specific name has been published
tipc_available_nodes() - [likely to be obsoleted]

/* TIPC operating mode routines */

tipc_get_addr() - get <Z.C.N> of own node
tipc_get_mode() - get TIPC operating mode
tipc_attach()   - register application as a TIPC user
tipc_detach()   - deregister TIPC user & free all associated ports
```

3.3. Examples

3.3.1. Common port operations

Create a port:

```
static u32 port_ref;

tipc_createport(0, NULL, TIPC_LOW_IMPORTANCE,
  NULL, NULL, NULL,
  NULL, named_msg_event, NULL,
  NULL, &port_ref);
```

Bind the name {100,123} with "cluster" scope to the port:

```
struct tipc_name_seq seq;

seq.type  = 100 ;
seq.lower = 123 ;
seq.upper = 123 ;
tipc_publish(port_ref, TIPC_CLUSTER_SCOPE, &seq);
```

Process messages sent to port {100,123}:

```
   /* Note: This callback routine was specified during port creation above */

   static void named_msg_event(void *usr_handle,
          u32 port_ref,
          struct sk_buff **buf,
          unsigned char const *data,
          unsigned int size,
          unsigned int importance,
          struct tipc_portid const *orig,
          struct tipc_name_seq const *dest)
   {
/* 'data' points to message content, 'size' indicates how much */

printk("%s", data);

/* can send reply message(s) back to originator, if desired */

struct iovec my_iov;
char reply_info[30];

strcpy(reply_info, "here is the reply");
my_iov.iov_base = reply_info;
my_iov.iov_len = strlen(reply_info) + 1;
tipc_send2port(port_ref, orig, 1, &my_iov);

/* TIPC discards the received message upon exit */
   }
```

Delete the port:

```
   tipc_deleteport(port_ref);
```

3.3.2. Synchronous message receive

Application thread:

```
   /* Initialize data structures */

   struct sk_buff_head message_q;
   wait_queue_head_t wait_q;

   skb_queue_head_init(&message_q);
   init_waitqueue_head(&wait_q);

   /* Wait for messages; process & discard each one in turn */

   while (1) {
```

```
        struct sk_buff *skb;

        if (wait_event_interruptible(&wait_q, (!skb_queue_empty(&message_q))))
     continue;

 skb = skb_dequeue(&message_q);

        < ... Process message as required ... >

 kfree_skb(skb);
    }
```

Callback routine converts asynchronous receive into synchronous receive:

```
    static void named_msg_event(void *usr_handle,
           u32 port_ref,
           struct sk_buff **buf,
           unsigned char const *data,
           unsigned int size,
           unsigned int importance,
           struct tipc_portid const *orig,
           struct tipc_name_seq const *dest)
    {
 /* Add message to queue of unprocessed messages */

 skb_queue_tail(&message_q, *buf);

 /* Tell TIPC *not* to discard the received message upon exit */

 *buf = NULL;

 /* Wake up application */

        wake_up_interruptible(&wait_q);
    }
```

3.3.3. TIPC user registration

Register TIPC user:

```
    static u32 user_ref;

    tipc_attach(&user_ref, NULL, NULL);
```

Create port and associate with registered TIPC user:

```
    static u32 port_ref;

    tipc_createport(user_ref, NULL, TIPC_LOW_IMPORTANCE,
```

```
      NULL, NULL, NULL,
      NULL, named_msg_event, NULL,
      NULL, &port_ref);
```

Deregister TIPC user (and all associated ports):

```
    tipc_detach(user_ref);
```

3.3.4. More examples

Demo programs utilizing the native API can be found at http://tipc.sf.net .

In addition, the TIPC source code itself contains a couple of sections that utilize the native API just like an application might:

1. `tipc_cfg_init()` in `net/tipc/tipc_cfgsrv.c`

   This file contains the TIPC configuration service (using port name {0,<Z.C.N>}, which handles messages sent by the tipc-config application. It utilizes a very simple connectionless request-and-reply approach to messaging.

2. `tipc_subscr_start()` in `net/tipc/tipc_topsrv.c`

   This file contains the TIPC topology service (using port name {1,1}), which handles subscription requests from applications and returns subscription events. It demonstrates the correct way to handle connection establishment (both explicit and implied) and tear down (both self-initiated and peer-initiated).

4. Tips and Techniques

This section illustrates some techniques that may be useful when designing applications using TIPC.

4.1. Determining which node a socket is running on

Use `getsockname()` to determine the port ID of the socket, then examine the "node" field to determine the network address its node.

For example, if `getsockname()` returns a port ID of <1.1.19:1234567> then the node field will be 0x01001013, which represents network address <1.1.19>.

4.2. When to use implied connect

The easiest way to decide whether to use the implied connect technique or the explicit connect approach is to design your program assuming that you will be using an explicit connect.

If you end up with:

- a `connect()`,
- followed by a send routine,

- followed by a receive routine,

then you should be able to combine the `connect()` and send routine into a single `sendto()`. This saves your program the overhead of an additional system call, and the exchange of empty handshaking messages during connection establishment.

On the other hand, if you end up with:

- a `connect()` followed by a receive, or
- a `connect()` followed by two sends,

then your program can't use the implied connect approach.

4.3. Dummy subscriptions

It is sometimes useful to issue a "dummy" subscription to the TIPC topology server, which has a time limit but never matches any published name. The dummy subscription never generates any publish or withdraw events, but it does generate a timeout event when it expires. This can be handy if your application processes events associated with other subscriptions, but needs to be able to continue processing even if no events occur.

For example, suppose an application wishes to report what nodes are present in the network every minute. This can be accomplished as follows:

```
connect to TIPC topology server
send subscription for {0,1,2^^31-1}, time limit = none
send subscription for {1,0,0}, time limit = 60 seconds
initialize set of available nodes to "empty"
loop
        receive event
        if (event == publish)
                add node to set of available nodes
        else if (event == withdraw)
                remove node from set of available nodes
        else
                send subscription for {1,0,0}, time limit = 60 seconds
                print list of available nodes
end loop
```

The use of a dummy subscription having the name sequence {1,0,0} allows the application to print the desired information at the required time without having to use additional timers or threads of control, and without having to deal with the complexities of determining how long to continue waiting when the main thread is awoken to process a publish or withdraw event (which can be an issue if the programmer tries using a time-limited`select()` to accomplish this).

> *Note:* This code ignores the processing overhead required to re-issue the dummy subscription, which will result in each successive display occurring slightly more than 60 seconds after the previous one. This could be compensated for by measuring the overhead and subtracting it from the specified time limit each time the dummy subscription is re-issued.

The name sequence specified in a dummy subscription can be anything that the application designer knows will not generate any publish or withdraw events, and need not be {1,0,0}.

4.4. Processing a returned message

The following routines have been created to assist a TIPC programmer who is trying to determine why an undelivered message has been returned to its sender. Simply modify the sending application so that it calls `recvfrom_anc()` instead of `recvfrom()`, or `recv_anc()` instead of `recv()`, and the routine will display any ancillary data object(s) associated with a returned message.

The code in `recvfrom_anc()` can also be used as a guide for programmers who need their application to parse the ancillary data objects so that it can respond to a returned message in different ways, depending on the content of the message.

```
/*
 * Enhanced forms of standard recvfrom() and recv() routines
 * that display any ancillary data accompanying a received message.
 */

ssize_t recvfrom_anc(int sd, void *buf, size_t nbytes, int flags,
       struct sockaddr *from, socklen_t *addrlen)
{
 struct msghdr msg;
 struct iovec iov;
 char anc_buf[CMSG_SPACE(8) + CMSG_SPACE(1024) + CMSG_SPACE(12)];
 struct cmsghdr *anc;
 unsigned char *cptr;
 __u32 *iptr;
 int i;
 int has_addr;
 socklen_t optlen;
 ssize_t sz;

 has_addr = (from != NULL) && (addrlen != NULL);

 iov.iov_base = buf;
 iov.iov_len = nbytes;

 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;
 msg.msg_name = from;
 msg.msg_namelen = (has_addr) ? *addrlen : 0;
 msg.msg_control = anc_buf;
 msg.msg_controllen = sizeof(anc_buf);

 sz = recvmsg(sd, &msg, flags);
 if (sz >= 0) {
  anc = CMSG_FIRSTHDR(&msg);
  while (anc != NULL) {
   cptr = CMSG_DATA(anc);

   if (anc->cmsg_type == TIPC_ERRINFO) {
    iptr = (__u32 *)cptr;
    printf("error code = %u, data size = %u bytes\n",
           iptr[0], iptr[1]);
    /*
```

```
     * provide the same return code value
     * that TIPC's recv() and recvfrom() generate
     */
                                i = -1;
    optlen = sizeof(i);
    getsockopt(sd, SOL_SOCKET, SO_TYPE, &i, &optlen);
    if (((i == SOCK_SEQPACKET) || (i == SOCK_STREAM))
        && (iptr[0] != TIPC_CONN_SHUTDOWN))
     sz = -1;
   } else if (anc->cmsg_type == TIPC_RETDATA) {
    for (i = anc->cmsg_len - sizeof(*anc); i > 0; i--) {
     printf("returned byte 0x%02x\n", *cptr);
     cptr++;
    }
   } else if (anc->cmsg_type == TIPC_DESTNAME) {
    iptr = (__u32 *)cptr;
    printf("destination name = {%u,%u,%u}\n",
           iptr[0], iptr[1], iptr[2]);
   } else {
    printf("unrecognized ancillary data type %u\n",
           anc->cmsg_type);
   }

   anc = CMSG_NXTHDR(&msg, anc);
  }

  if (has_addr)
   *addrlen = msg.msg_namelen;
 }

 return sz;
}

ssize_t recv_anc(int sd, void *buf, size_t nbytes, int flags)
{
 return recvfrom_anc(sd, buf, nbytes, flags, NULL, 0);
}
```

[END OF DOCUMENT]