# Transparent Inter-Process Communications (TIPC) libraries

Jeffrey Rosenwald
E-mail: JeffRose@acm.org

July 3, 2009

**Abstract**

TIPC provides a framework for cooperation between federations of trusted peers that are operating as a unit. It was developed by Ericsson AB, as a means to provide for communications between Common Control Systems processes and Network Elements in telephone switching systems, sometimes operating at arm's length on different line cards or mainframes. Delegation of responsibility in this way is one of the fundamental precepts of the Erlang programming system, also developed at Ericsson. TIPC represents a more generalized version of the same behavioral design pattern.

# Contents

# 1 Transparent Inter-Process Communications (TIPC)

These pages are not intended as a comprehensive tutorial in the use of TIPC services. The TIPC Programmer's Guide, `http://tipc.sf.net/doc/Programmers_Guide.txt`, provides assistance to developers who are creating applications that utilize TIPC services. The TIPC User's Guide, `http://tipc.sf.net/doc/Users_Guide.txt`, provides an administrator of a TIPC cluster with the information needed to operate one. A TIPC server loadable module, that may be used to make a host available as a TIPC enabled node, has been a part of the Linux kernel since 2.6.16. Please see: `http://tipc.sf.net`

## 1.1 Overview

In a TIPC network, a Node is comprised of a collection of lightweight threads of execution operating in the same process, or heavyweight processes operating on the same machine. A Cluster is a collection of Nodes operating on different machines, and operating indirectly by way of a local Ethernet or other networking medium. Clusters may be further aggregated into Zones, and Zones into Networks. The address space of two TIPC networks is completely disjoint. Zones on different networks may coexist on the same LAN but they may not communicate directly with one another.

TIPC provides connectionless, connection-oriented, reliable, and unreliable forwarding strategies for both stream and message oriented applications. But not all strategies can be used in every application. For example, there is no such thing as a multicast byte stream. The strategy is selected by the user for the application when the socket is instantiated.

TIPC is not TCP/IP based. Consequently, it cannot signal beyond a local network span without some kind of tunneling mechanism. TIPC is designed to facilitate deployment of distributed applications, where certain aspects of the application may be segregated, and then delegated and/or duplicated over several machines on the same LAN. The application is unaware of the topology of the network on which it is running. It could be a few threads operating in the same process, several processes operating on the same machine, or it could be dozens or even hundreds of machines operating on the same LAN, all operating as a unit. TIPC manages all of this complexity so that the programmer doesn't have to.

Unlike TCP/IP, TIPC does not assign network addresses to network interfaces; it assigns addresses (e.g. port-ids) to sockets when they are instantiated. The address is unique and persists only as long as the socket persists. A single Node therefore, may typically have many TIPC addresses active at any one time, each assigned to an active socket. TIPC also provides a means that a process can use to bind a socket to a well-known address (e.g. a service). Several peers may bind to the same well-known address, thereby enabling multi-server topologies. And server members may exist anywhere in the Zone. TIPC manages the distribution of client requests among the membership of the server group. A server instance responds to two addresses: its public well-known address that it is bound to, and that a client may use to establish a communication with a service, and its private address that the server instance may use to directly interact with a client instance.

TIPC also enables multicast and "publish and subscribe" regimes that applications may use to facilitate asynchronous exchange of datagrams with a number of anonymous sources that may come and go over time. One such regime is implemented as a naming service managed by a distributed topology server. The topology server provides surveillance on the comings and goings of publishers, with advice to interested subscribers in the form of event notifications, emitted when a publisher's status changes. For example, when a server application binds to a TIPC address , that address is automatically associated with that server instance in topology server's name table. This has the side

effect of causing a "published" event to be emitted to all interested subscribers. Conversely, when the server's socket is closed or when one of its addresses is released using the "no-scope" option of `tipc_bind/3`, a "withdrawn" event is emitted. See `tipc_service_port_monitor/2`.

A client application may connect to the topology server in order to interrogate the name table to determine whether or not a service is present before actually committing to access it. See `tipc_service_exists/2` and `tipc_service_probe/2`. Another way that the topology server can be applied is exemplified in Erlang's "worker/supervisor" behavioral pattern. A supervisor thread has no other purpose than to monitor a collection of worker threads in order to ensure that a service is available and able to serve a common goal. When a worker under the supervisor's care dies, the supervisor receives the worker's "withdrawn" event, and takes some action to instantiate a replacement. The predicate, `tipc_service_port_monitor/2`, is provided specifically for this purpose. Using the service is optional. It has applications in distributed, high-availability, fault-tolerant, and non-stop systems.

Adding capacity to a cluster becomes an administrative function whereby new server hardware is added to a TIPC network, then the desired application is launched on the new server. The application binds to its well-known address, thereby joining in the Cluster. TIPC will automatically begin sending work to it. An administrator has tools for gracefully removing a server from a Cluster, without effecting the traffic moving on the Cluster.

An administrator may configure a Node to have two or more network interfaces. Provided that each interface is invisible to the other, TIPC will manage them as a redundant group, thus enabling high-reliability network features such as automatic link fail-over and hot-swap.

## 1.2 TIPC Address Structures

**name**(+*Type, +Instance, +Domain*)

A TIPC name address is used by servers to advertise themselves as services in unicast applications, and is used by clients to connect to unicast services. Type, Instance, and Domain are positive integers that are unique to a service.

**name_seq**(+*Type, +Lower, +Upper*)

A TIPC name-sequence address is used by servers to advertise themselves as services in multicast and "publish and subscribe" applications. Lower and Upper represent a range of instance addresses. Each server will receive exactly one datagram from a client that sends a name-sequence address that matches the server's Type, and where its Lower and Upper instance range intersects the Lower and Upper instance range bound to the server. Clients may send a datagram to any and all interested servers by providing an appropriate name-sequence address to `tipc_send/4`.

**port_id**(+*Ref, +Node*)

A TIPC port-id is the socket's private address. It is ephemeral in nature. It persists only as long as the socket instance persists. Port ids are generally provided to applications via `tipc_receive/4`. An application may discover its own port_id for a socket using `tipc_get_name/2`. Generally, others cannot discover the port-id of a socket, except by receiving messages originated from it. A server responds to a client by providing the received port-id as the sender address in a reply message. The client will receive the server's port-id via his own `tipc_receive/4`. The client can then interact with a specific server instance without having to perform any additional address resolution. The client simply sends all

subsequent messages related to a specific transaction to the server instance using the port-id received from the server in its replies.

Sometimes the socket's port-id alone is enough to establish an ad-hoc session anonymously between parent and child processes. The parent instantiates a socket, then forks into two processes. The child retrieves the port-id of the parent from the socket inherited from the parent using `tipc_get_name/2`, then closes the socket and instantiates a socket of its own. The child sends a message to the parent, on its own socket, using the parent's port-id as the destination address. The port-id received by the parent is unique to a specific instance of child. The handshake is complete; each side knows who the other is, and two-way communication may now proceed. A one-way communication (e.g. a message oriented pipe or mailbox) is also possible using only the socket inherited from the parent, provided that there is exactly one sender and one receiver on the socket. Both parent and child use the socket's own port-id, one side adopts the role of sender, and the other of receiver.

# 2 The TIPC libraries: `tipc/...`

## 2.1 tipc.pl – TIPC Sockets

**author** Jeffrey Rosenwald (JeffRose@acm.org)
**See also** `http://tipc.sf.net`, `http://www.erlang.org`
**Compatibility** Linux only
**license** LGPL

Transparent Inter-Process Communication (TIPC) provides a flexible, reliable, fault-tolerant, high-speed, and low-overhead framework for inter-process communication between federations of trusted peers, operating as a unit. It was developed by Ericsson AB, as a means to provide for communications between Common Control Systems processes and Network Element peers in telephone switching systems, sometimes operating at arm's length on different line cards or mainframes. Delegation of responsibility in this way is one of the fundamental precepts of the Erlang programming system, also developed at Ericsson. TIPC represents a more generalized version of the same behavioral design pattern. For an overview, please see: `tipc_overview.txt`.

**tipc_socket**(-*SocketId*, +*SocketType*) *[det]*

Creates a TIPC-domain socket of the type specified by *SocketType*, and unifies it to an identifier, *SocketId*.

Parameters

*SocketType* is one of the following atoms:

- rdm - unnumbered, reliable datagram service,
- dgram - unnumbered, unreliable datagram service,
- seqpacket - numbered, reliable datagram service, and
- stream - reliable, connection-oriented byte-stream service

**Errors** socket_error('Address family not supported by protocol') is thrown if a TIPC server is not available on the current host.

**tipc_close_socket**(+*SocketId*) *[det]*

Closes the indicated socket, making *SocketId* invalid. In stream applications, sockets are closed by closing both stream handles returned by `tipc_open_socket/3`. There are two cases where `tipc_close_socket/1` is used because there are no stream-handles:

- After `tipc_accept/3`, the server does a `fork/1` to handle the client in a sub-process. In this case the accepted socket is not longer needed from the main server and must be discarded using `tipc_close_socket/1`.
- If, after discovering the connecting client with `tipc_accept/3`, the server does not want to accept the connection, it should discard the accepted socket immediately using `tipc_close_socket/1`.

Parameters

| | |
|---|---|
| *SocketId* | the socket identifier returned by `tipc_socket/2` or `tipc_accept/3`. |

**Errors** socket_error('Invalid argument) is thrown if an attempt is made to close a socket identifier that has already been closed.

**tipc_open_socket**(+*SocketId, -InStream, -OutStream*) *[det]*

Opens two SWI-Prolog I/O-streams, one to deal with input from the socket and one with output to the socket. If `tipc_bind/3` has been called on the socket, *OutStream* is useless and will not be created. After closing both *InStream* and *OutStream*, the socket itself is discarded.

**tipc_bind**(+*Socket, +Address, +ScopingOption*) *[det]*

Associates/disassociates a socket with the `name/3` or `name_seq/3` address specified in *Address*. It also registers/unregisters it in the topology server name table. This makes the address visible/invisible to the rest of the network according to the scope specified in *ScopingOption*. *ScopingOption* is a grounded term that is one of:

**scope(Scope)** where Scope is one of: `zone`, `cluster`, or `node`. Servers may bind to more than one address by making successive calls to `tipc_bind/3`, one for each address that it wishes to advertise. The server will receive traffic for all of them. A server may, for example, register one address with node scope, another with cluster scope, and a third with zone scope. A client may then limit the scope of its transmission by specifying the appropriate address.

**no_scope(Scope)** where Scope is as defined above. An application may target a specific address for removal from its collection of addresses by specifying the address and its scope. The scoping option, `no_scope(all)`, may be used to unbind the socket from all of its registered addresses. This feature allows an application to gracefully exit from service. Because the socket remains open, the application may continue to service current transactions to completion. TIPC however, will not schedule any new work for the server instance. If no other servers are available, the work will be rejected or dropped according to the socket options specified by the client.

Connection-oriented, byte-stream services are implemented with this predicate combined with `tipc_listen/2` and `tipc_accept/3`. Connectionless, datagram services may be implemented using `tipc_receive/4`.

Note that clients do not need to bind to any address. Its port-id is sufficient for this role. And server sockets (e.g. those that are bound to `name/3` or `name_seq/3`, addresses) may not act as clients. That is, they may not originate connections from the socket. Please see the TIPC programmers's guide for other restrictions.

**tipc_listen**(+*Socket, +Backlog*) *[det]*

Listens for incoming requests for connections. *Backlog* indicates how many pending connection requests are allowed. Pending requests are requests that are not yet acknowledged using `tipc_accept/3`. If the indicated number is exceeded, the requesting client will be signalled that the service is currently not available. A suggested default value is 5.

**tipc_accept**(+*Socket, -Slave, -Peer*) *[det]*

Blocks on a server socket and waits for connection requests from clients. On success, it creates a new socket for the client and binds the identifier to *Slave*. *Peer* is bound to the TIPC address, `port_id/2`, of the client.

**tipc_connect**(+*Socket, +TIPC_address*) *[det]*

Provides a connection-oriented, client-interface to connect a socket to a given *TIPC_address*. After successful completion, `tipc_open_socket/3` may be used to create I/O-Streams to the remote socket.

**tipc_get_name**(+*Socket, -TIPC_address*) *[det]*

Unifies *TIPC_address* with the port-id assigned to the socket.

**tipc_setopt**(+*Socket, +Option*) *[det]*

Sets options on the socket. Defined options are:

**importance(+Priority)** Allow sockets to assign a priority to their traffic. Priority is one of : `low` (default), `medium`, `high`, or `critical`.

**src_droppable(+Boolean)** Allow TIPC to silently discard packets in congested situations, rather than queuing them for later transmission.

**dest_droppable(+Boolean)** Allow TIPC to silently discard packets in congested situations, rather than returning them to the sender as undeliverable.

**conn_timeout(+Seconds)** Specifies the time interval that `tipc_connect/2` will use before abandoning a connection attempt. Default: 8.000 sec.

**tipc_receive**(+*Socket, -Data, -From, +OptionList*) *[det]*

Waits for, and returns the next datagram. Like its UDP counterpart, the data are returned as a Prolog string object (see `string_to_list/2`). *From* is an address structure of the form `port_id/2`, indicating the sender of the message.

Defined options are:

**as**(+*Type*)

Defines the returned term-type. Type is one of atom, codes or string (default).

**nonblock**

Poll the socket and return immediately. If a message is present, it is returned. If not, then an exception, error(socket_error('Resource temporarily unavailable'), _), will be

thrown. Users are cautioned not to ”spin” unnecessarily on non-blocking receives as they may prevent the system from servicing other background activities such as XPCE event dispatching.

The typical sequence to receive a connectionless TIPC datagram is:

```
receive :-
        tipc_socket(S, dgram),
        tipc_bind(S, name(18888, 10, 0), scope(zone)),
        repeat,
            tipc_receive(Socket, Data, From, [as(atom)]),
            format('Got ~q from ~q~n', [Data, From]),
            Data == quit,
        !, tipc_close_socket(S).
```

**tipc_send**(+*Socket, +Data, +To, +Options*) *[det]*

Sends a TIPC datagram to one or more destinations. Like its UDP counterpart, *Data* is a string, atom or code-list providing the data to be sent. *To* is a `name/3`, `name_seq/3`, or `port_id/2` address structure. See `tipc_overview.txt`, for more information on TIPC Address Structures. *Options* is currently unused.

A simple example to send a connectionless TIPC datagram is:

```
send(Message) :-
        tipc_socket(S, dgram),
        tipc_send(S, Message, name(18888, 10,0), []),
        tipc_close_socket(S).
```

Messages are delivered silently unless some form of congestion was encountered and the `dest_droppable(false)` option was issued on the sender's socket. In this case, the send succeeds but a notification in the form of an empty message is returned to the sender from the receiver, indicating some kind of delivery failure. The port-id of the receiver is returned in congestion conditions. A `port_id(0,0)`, is returned if the destination address was invalid. Senders and receivers should beware of this possibility.

**tipc_canonical_address**(-*CanonicalAddress, +PortId*) *[det]*

Translates a `port_id/2` address into canonical TIPC form:

**tipc_address**(*Zone, Cluster, Node, Reference*)

It is provided for debugging an printing purposes only. The canonical address is not used for any other purpose.

**tipc_service_exists**(+*Address, +Timeout*) *[semidet]*

**tipc_service_exists**(+*Address*) *[semidet]*

Interrogates the TIPC topology server to see if a service is available at an advertised *Address*.

Parameters

| | |
|---|---|
| *Address* | is one of: `name(Type, Instance, Domain)` or `name_seq(Type, Lower, Upper)`. A `name/3`, address is translated to a `name_seq/3`, following, where Lower and Upper are assigned the value of Instance. Domain is unused and must be zero. A `name_seq(Type, Lower, Upper)` is a multi-cast address. This predicate succeeds if there is at least one service that would answer according to multi-cast addressing rules. |
| *Timeout* | is optional. It is a non-negative real number that specifies the amount of time in seconds to block and wait for a service to become available. Fractions of a second are also permissible. |

**tipc_service_probe(***?Address***)** *[nondet]*
**tipc_service_probe(***?Address, ?PortId***)** *[nondet]*

Allows a user to discover the instance ranges and/or port-ids for a particular service.

Parameters

| | |
|---|---|
| *Address* | is a `name_seq/3` address. The address type must be grounded. |
| *PortId* | is unified with the port-id for a specific name_sequence address. |

**tipc_service_port_monitor(+***Addresses, :Goal***)** *[det]*
**tipc_service_port_monitor(+***Addresses, :Goal, ?Timeout***)** *[det]*

Monitors a collection of worker threads that are bound to a list of *Addresses*. A single port monitor may be used to provide surveillance over workers that are providing a number of different services. For a given address type, discontiguous port ranges may be specified, but overlapping port ranges may not. *Goal* for example, may simply choose to broadcast the notification, thus delegating the notification event handling to others.

Parameters

*Addresses* is a list of `name/3` or `name_seq/3` addresses for the services to be monitored.

*Goal* is a predicate with an arity of one, that will be called when a worker's publication status changes. The *Goal* is called exactly once per event with the structure:

**published(-NameSeq, -PortId)** when the worker binds its socket to the address.

**withdrawn(-NameSeq, -PortId)** when the worker unbinds its socket from the address.

*Timeout* is optional. It is one of:

*Timeout* a non-negative real number that specifies the number of seconds that surveillance is to be continued.

**infinite** causes the monitor to run forever in the current thread (e.g. never returns).

**detached(-ThreadId)** causes the monitor to run forever as a separate thread. ThreadId is unified with the thread identifier of the monitor thread. This is useful when the monitor is required to provide continuous surveillance, while operating in the background.

## 2.2 tipc_broadcast.pl – A TIPC Broadcast Bridge

**author** Jeffrey Rosenwald (JeffRose@acm.org)
**See also** `tipc.pl`
**Compatibility** Linux only
**license** LGPL

SWI-Prolog's broadcast library provides a means that may be used to facilitate publish and subscribe communication regimes between anonymous members of a community of interest. The members of the community are however, necessarily limited to a single instance of Prolog. The TIPC broadcast library removes that restriction. With this library loaded, any member of a TIPC network that also has this library loaded may hear and respond to your broadcasts.

This module has no public predicates. When it initialized, it did three things:

- It started a listener daemon thread that is listening for broadcasts from others, received as TIPC datagrams, and
- It registered three listeners: `tipc_node/1`, `tipc_cluster/1`, and `tipc_zone/1`, and
- It registered three listeners: `tipc_node/2`, `tipc_cluster/2`, and `tipc_zone/2`.

A `broadcast/1` or `broadcast_request/1` that is not directed to one of the six listeners above, behaves as usual and is confined to the instance of Prolog that originated it. But when so directed, the broadcast will be sent to all participating systems, including itself, by way of TIPC's multicast addressing facility. The principal functors `tipc_node`, `tipc_cluster`, and `tipc_zone`,

specify the scope of the broadcast. The functor `tipc_node`, specifies that the broadcast is to be confined to members of a present TIPC node. Likewise, `tipc_cluster` and `tipc_zone`, specify that the traffic should be confined to members of a present TIPC cluster and zone, respectively. To prevent the potential for feedback loops, the scope qualifier is stripped from the message before transmission. See library module `tipc.pl`, for more information.

An example of three separate processes cooperating on the same Node:

```
Process A:

   ?- listen(number(X), between(1, 5, X)).
   true.

   ?-

Process B:

   ?- listen(number(X), between(7, 9, X)).
   true.

   ?-

Process C:

   ?- findall(X, broadcast_request(tipc_node(number(X))), Xs).
   Xs = [1, 2, 3, 4, 5, 7, 8, 9].

   ?-
```

It is also possible to carry on a private dialog with a single responder. To do this, you supply a compound of the form, Term:PortId, to a TIPC scoped `broadcast/1` or `broadcast_request/1`, where PortId is the port-id of the intended listener. If you supply an unbound variable, PortId, to broadcast_request, it will be unified with the address of the listener that responds to Term. You may send a directed broadcast to a specific member by simply providing this address in a similarly structured compound to a TIPC scoped `broadcast/1`. The message is sent via unicast to that member only by way of the member's broadcast listener. It is received by the listener just as any other broadcast would be. The listener does not know the difference.

Although this capability is needed under some circumstances, it has a tendency to compromise the resilience of the broadcast model. You should not rely on it too heavily, or fault tolerance will suffer.

For example, in order to discover who responded with a particular value:

```
Process A:

   ?- listen(number(X), between(1, 3, X)).
   true.

   ?-
```

```
Process B:

   ?- listen(number(X), between(7, 9, X)).
   true.

   ?-

Process C:

   ?- broadcast_request(tipc_node(number(X):From)).
   X = 7,
   From = port_id('<1.1.1:3971170279>') ;
   X = 8,
   From = port_id('<1.1.1:3971170279>') ;
   X = 9,
   From = port_id('<1.1.1:3971170279>') ;
   X = 1,
   From = port_id('<1.1.1:3971170280>') ;
   X = 2,
   From = port_id('<1.1.1:3971170280>') ;
   X = 3,
   From = port_id('<1.1.1:3971170280>') ;
   false.

?-
```

### 2.2.1 Caveats:

While the implementation is mostly transparent, there are some important and subtle differences that must be taken into consideration:

- Prolog's `broadcast_request/1` is nondet. It sends the request, then evaluates the replies synchronously, backtracking as needed until a satisfactory reply is received. The remaining potential replies are not evaluated. This is not so when TIPC is involved.
- A TIPC `broadcast/1` is completely asynchronous.
- A TIPC `broadcast_request/1` is partially synchronous. A `broadcast_request/1` is sent, then the sender balks for a period of time (default: 250 ms) while the replies are collected. Any reply that is received after this period is silently discarded. An optional second argument is provided so that a sender may specify more (or less) time for replies.
- Replies are collected using `findall/3`, then the list of replies is presented to the user as a choice-point, using `member/2`. If a listener is connected to a generator that always succeeds (e.g. a random number generator), then the broadcast request will never terminate and trouble is bound to ensue.
- `broadcast_request/1` with TIPC scope is *not* reentrant (at least, not now anyway). If a listener performs a `broadcast_request/1` with TIPC scope recursively, then disaster

looms certain. This caveat does not apply to a TIPC scoped `broadcast/1`, which can safely be performed from a listener context.

- TIPC's capacity is not infinite. While TIPC can tolerate substantial bursts of activity, it is designed for short bursts of small messages. It can tolerate several thousand replies in response to a `broadcast_request/1` without trouble, but it will begin to encounter congestion beyond that. And in congested conditions, things will start to become unreliable as TIPC begins prioritizing and/or discarding traffic.
- A TIPC `broadcast_request/1` term that is grounded is considered to be a broadcast only. No replies are collected unless the there is at least one unbound variable to unify.
- A TIPC `broadcast/1` always succeeds, even if there are no listeners.
- A TIPC `broadcast_request/1` that receives no replies will fail.
- Replies may be coming from many different places in the network (or none at all). No ordering of replies is implied.
- Prolog terms are sent to others after first converting them to atoms using `term_to_atom/2`. Passing real numbers this way may result in a substantial truncation of precision. See prolog flag option, 'float_format', of `current_prolog_flag/2`.

## 2.3 tipc_paxos.pl – A Replicated Data Store

**author** Jeffrey Rosenwald (JeffRose@acm.org)
**See also** `tipc_broadcast.pl`
**Compatibility** Linux only, tipc_broadcast
**license** LGPL

This module provides a replicated data store that is coordinated using a variation on Lamport's Paxos concensus protocol. The original method is described in his paper entitled, "The Part-time Parliament", which was published in 1998. The algorithm is tolerant of non-Byzantine failure. That is late or lost delivery or reply, but not senseless delivery or reply. The present algorithm takes advantage of the convenience offered by multicast to the quorum's membership, who can remain anonymous and who can come and go as they please without effecting Liveness or Safety properties.

Paxos' quorum is a set of one or more attentive members, whose processes respond to queries within some known time limit ($<$ 20ms), which includes roundtrip delivery delay. This property is easy to satisfy given that every coordinator is necessarily a member of the quorum as well, and a quorum of one is permitted. An inattentive member (e.g. one whose actions are late or lost) is deemed to be "not-present" for the purposes of the present transaction and consistency cannot be assured for that member. As long as there is at least one attentive member of the quorum, then persistence of the database is assured.

Each member maintains a ledger of terms along with information about when they were originally recorded. The member's ledger is deterministic. That is to say that there can only be one entry per functor/arity combination. No member will accept a new term proposal that has a line number that is equal-to or lower-than the one that is already recorded in the ledger.

Paxos is a three-phase protocol:

1: A coordinator first prepares the quorum for a new proposal by broadcasting a proposed term. The quorum responds by returning the last known line number for that functor/arity combination that is recorded in their respective ledgers.

2: The coordinator selects the highest line number it receives, increments it by one, and then asks the quorum to finally accept the new term with the new line number. The quorum checks their respective ledgers once again and if there is still no other ledger entry for that functor/arity combination that is equal-to or higher than the specified line, then each member records the term in the ledger at the specified line. The member indicates consent by returning the specified line number back to the coordinator. If consent is withheld by a member, then the member returns a `nack` instead. The coordinator requires unanimous consent. If it isn't achieved then the proposal fails and the coordinator must start over from the beginning.

3: Finally, the coordinator concludes the successful negotiation by broadcasting the agreement to the quorum in the form of a `paxos_changed(Term)` event. This is the only event that should be of interest to user programs.

For practical reasons, we rely on the partially synchronous behavior (e.g. limited upper time bound for replies) of `broadcast_request/1` over TIPC to ensure Progress. Perhaps more importantly, we rely on the fact that the TIPC broadcast listener state machine guarantees the atomicity of `broadcast_request/1` at the process level, thus obviating the need for external mutual exclusion mechanisms.

*Note that this algorithm does not guarantee the rightness of the value proposed. It only guarantees that if successful, the value proposed is identical for all attentive members of the quorum.*

**tipc_paxos_set(***?Term***)** *[semidet]*
**tipc_paxos_set(***?Term, +Retries***)** *[semidet]*

negotiates to have *Term* recorded in the ledger for each of the quorum's members. This predicate succeeds if the quorum unanimously accepts the proposed term. If no such entry exists in the Paxon's ledger, then one is silently created. `tipc_paxos_set/1` will retry the transaction several times (default: 20) before failing. Failure is rare and is usually the result of a collision of two or more writers writing to the same term at precisely the same time. On failure, it may be useful to wait some random period of time, and then retry the transaction. By specifying a retry count of zero, `tipc_paxos_set/2` will succeed iff the first ballot succeeds.

On success, `tipc_paxos_set/1` will also broadcast the term `paxos_changed(Term)`, to the quorum.

Parameters

| | |
|---|---|
| *Term* | is a compound that may have unbound variables. |
| *Retries* | (optional) is a non-negative integer specifying the number of retries that will be performed before a set is abandoned. |

**tipc_paxos_get(***?Term***)** *[semidet]*

unifies *Term* with the entry retrieved from the Paxon's ledger. If no such entry exists in the member's local cache, then the quorum is asked to provide a value, which is verified for

consistency. An implied `tipc_paxos_set/1` follows. This predicate succeeds if a term with the same functor and arity exists in the Paxon's ledger, and fails otherwise.

Parameters

| | |
|---|---|
| *Term* | is a compound. Any unbound variables are unified with those provided in the ledger entry. |

**tipc_paxos_replicate**(*?Term*) *[det]*

declares that *Term* is to be automatically replicated to the quorum each time it becomes grounded. It uses the behavior afforded by `when/2`.

Parameters

| | |
|---|---|
| *Term* | is an ungrounded *Term* |

**tipc_paxos_on_change**(*?Term, :Goal*) *[det]*

executes the specified *Goal* when *Term* changes. `tipc_paxos_on_change/2` listens for `paxos_changed/1` notifications for *Term*, which are emitted as the result of successful `tipc_paxos_set/1` transactions. When one is received for *Term*, then *Goal* is executed in a separate thread of execution.

Parameters

| | |
|---|---|
| *Term* | is a compound, identical to that used for `tipc_paxos_get/1`. |
| *Goal* | is one of:<br>• a callable atom or term, or<br>• the atom `ignore`, which causes monitoring for *Term* to be discontinued. |

# Index