Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

TIPC: Providing Communication for Linux Clusters

Jon Paul Maloy
Ericsson Research, Montreal
`jon.maloy@ericsson.com`

Abstract

Transparent Inter Process Communication (TIPC) is a protocol specially designed for efficient intra cluster communication, leveraging the particular conditions present within clusters of loosely coupled nodes.

TIPC provides a powerful infrastructure for designing distributed, site-independent, scalable, highly- available and high-performing applications, as well as a good support for cluster, network and software management functionality. In this paper, we will discuss the motives for developing TIPC and describe its architecture. Then, we will present the most important features of TIPC, such as its functional, location transparent, addressing scheme, lightweight reactive connections, reliable multicast, signalling link protocol, topology subscription services and more. We will also discuss the various design decisions that influenced the implementation of these features. We conclude by describing the current implementation status and our planned roadmap for TIPC.

1 Introduction

For the last six years, telecom equipment vendor Ericsson has been developing and deploying a tailor-made reliable communication protocol, TIPC,for their cluster-based products. This protocol has recently undergone a significant redesign, and is now available as a portable source code package of about 12,500 lines of C code. The code implements a Linux kernel driver, a design that has made it possible to improve performance (35% faster than TCP) and minimize code footprint.

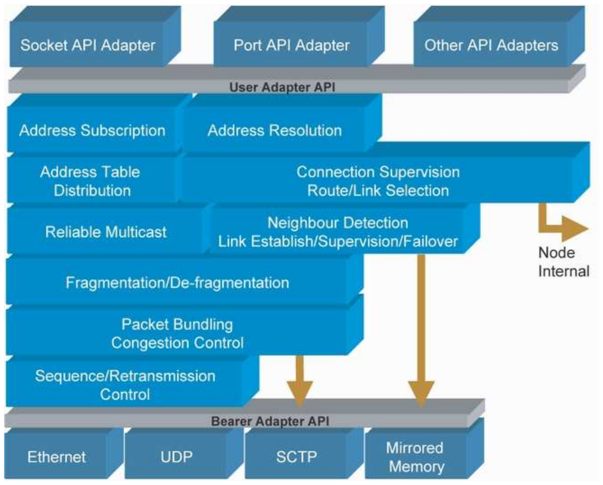
Socket API Adapter
Port API Adapter
Other API Adapters
User Adapter API
Address Subscription
Address Resolution
Address Table Distribution
Connection Supervision Route/Link Selection
Reliable Multicast
Neighbour Detection Link Establish/Supervision/Failover
Node Internal
Fragmentation/De-fragmentation
Packet Bundling Congestion Control
Sequence/Retransmission Control
Bearer Adapter API
Ethernet
UDP
SCTP
Mirrored Memory

Figure 1: *Functional View of TIPC*

The current version is available under a dual BSD/GPL license from [1]. TIPC is supported on Linux 2.4 and 2.6; and several proprietary portations to other OS'es (OSE, True64, VxWorks) also exist.

TIPC offers an interesting combination of features, some of them quite unique, to achieve the overall goal: to make the cluster act as one single computer from a communication viewpoint, while helping applications to keep track of and adapt to topology changes. Figure 1 illustrates a functional view of TIPC.

2 Motivation

There are no standard protocols available today that fully satisfy the special needs of application programs working within highly available, dynamic cluster environments. Clusters may grow or shrink by orders of magnitude; member nodes may crash and restart, routers may fail and be replaced, services may be moved around due to load balancing considerations, etc. All this must be possibe to handle without significant disturbances of the service offered by the cluster. In order to minimize the effort by the application programmers to deal with such situations, and to maximize the chance that they are handled in a correct and optimal way, the cluster internal communication service should provide special support, helping the applications to adapt to changes in the cluster. It should also, when possible, leverage the special conditions present within cluster environments to present a more efficient and fault-tolerant communication service than more general protocols are capable of.

2.1 Existing Protocols

TCP [2] has the advantage of being ubiquitous, proven, and wellknown by most programmers. Its most significant shortcomings in a real-time cluster environment are the following:

- TCP lacks any notion of functional addressing and addressing transparency. Mechanisms exist (DNS, CORBA Naming Service) for transparent and dynamic lookup of the correct IP-adress of a destination, but those are in general too static and too inefficient to be useful in a dynamic, real-time environment.

- Performance is not as good as it could be, especially for intra-node communication and for short messages in general. For intra-node communication, other more efficient mechanisms are available, at least on Unix, but then the location of the destination process has to be assumed, and can not be changed. It is desirable to have a protocol working efficiently for both intra-node and inter-node messaging, without forcing the user to distinguish between these cases in his code.

- The heavy connection setup/shutdown scheme of TCP is a disadvantage in a dynamic environment. The minimum number of packets exchanged for even the shortest TCP transaction is nine (SYN, SYNACK, etc.), while with TIPC this can be reduced to two, or even to one if connectionless mode is used.

- The connection-oriented nature of TCP makes it impossible to support true multicast.

Stream Control Transmission Protocol (SCTP) [3] is message oriented; it provides some level of user connection supervision, message bundling, loss-free changeover, and a few more features that may make it more suitable than TCP as an intra-cluster protocol. Otherwise, it has all the drawbacks of TCP already listed above.

Apart from these weaknesses, neither TCP nor SCTP provide any topology information/subscription service, something that has proven very useful both for applications and for management functionality operating within cluster environments.

Both TCP and SCTP are general purpose protocols, in the sense that they can be used safely over the Internet as well as within a closed cluster. This virtual advantage is also their major weakness: they require funtionality and header space to deal with situations that will never happen, or only infrequently, within clusters.

2.2 Assumptions

The TIPC design is based on the following assumptions, empirically known to be valid within most clusters.

- Most messages cross only one direct hop.
- Transfer time for most messages is short.
- Most messages are passed over intra-cluster connections.
- Packet loss rate is normally low; retransmission is infrequent.
- Available bandwidth and memory volume is normally high.
- For all relevant bearers packets are checksummed by hardware.
- The number of inter-communicating nodes is relatively static and limited at any moment in time.
- Security is a less crucial issue in closed clusters than on the Internet.

Because of the above one can use a simple, traffic-driven, fixed-size sliding window protocol located at the signalling link level, rather than a timer-driven transport level protocol. This in turn gives a lot of other advantages, such as earlier release of transmission buffers, earlier packet loss detection and retransmission, and earlier detection of node unavailability, only to mention some. Of course, situations with long transfer delays, high loss rates, long messages, security issues, etc. must be dealt with as well, but rather from the viewpoint of being exceptions than as the general rule.

3 Five-Layer Network Topology

From a TIPC viewpoint the network is organized in a five-layer structure (Figure 2).

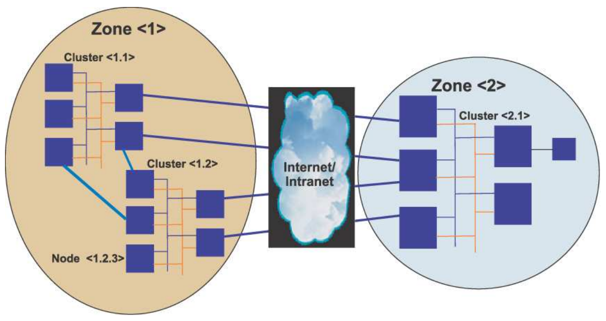

Figure 2: *TIPC Network Topology*

The top level is the *TIPC network*. This is the ensemble of all computers (nodes) interconnected via TIPC, i.e., the domain where any node can reach any other node by using a TIPC network address. A TIPC network is distringuished from other networks by its *network identity*, a 32-bit value that is known by all nodes.

The next level in the hierarchy is an entity called *zone*. This "cluster of clusters" is the maximum scope of location transparency within a network, i.e., the domain where any process can reach any other process by using a functional address rather than a network addresses.

The third level is what we call the *cluster*. This is a group of nodes interconnected all-to-all via one or two TIPC links.

The fourth level is the individual *system node*, or just *node*. There may be up to 2047 system nodes in a cluster.

The lowest level is the *slave node*. Slave nodes provide the same properties regarding location transparency and availability as system nodes, but they don't need full physical connectivity to the rest of the cluster. One link to one system node is sufficient, although there may be more for redundancy reasons.

All entities within a TIPC network are accessed using a TIPC network address, a 32-bit value subdivided into a zone, cluster, and node field. This address is internally mapped to the address type for the communication media actually used, e.g., an Ethernet address or an IP-address/port number tuplet.

4 Location-Transparent Functional Addressing

To present a cluster as one computer, the addressing scheme used must hide the physical location of a requested service to its users. To achieve this, TIPC provides a functional address type, called *port name*, to be used both for connectionless messaging and connection setup calls. Binding a socket to a port name corresponds to binding it to a port number in other protocols, except that the port name is unique and has validity for the whole cluster, not only the local node. A caller wanting to set up a connection needs only to specify this address, and the TIPC internal translation service ensures that the request ends up in the right socket, on the right node.

A port name consists of two 32-bit fields. The first field is called the *name type* and typically identifies a certain service type or function. The second field is the *name instance* and is used as a key for accessing a certain instance of the requested service. This address structure gives excellent support for both service partitioning and service load sharing.

Further support for service partitioning is provided by an address type called *port name sequence*. This is a three-integer structure defining a range of port names, i.e., a name type plus the lower and upper boundary of the instance range. By allowing a socket to bind to a sequence, instead of just an individual port name, it is possible to partition a service's scope of responsibility into sub-ranges, without having to create a vast number of sockets to do so.

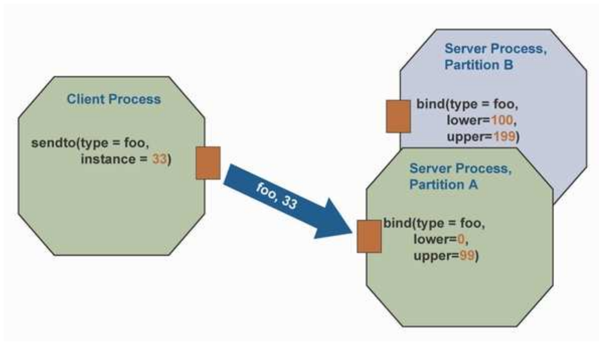
Server Process, Partition B
bind(type = foo, lower=100, upper=199)
Client Process
sendto(type = foo, instance = 33)
foo, 33
Server Process, Partition A
bind(type = foo, lower=0, upper=99)

Figure 3: *Functional Addressing*

This addressing scheme is illustrated by the example in Figure 3. Two processes, partition A and partition B of the service *foo*, bind their sockets to the port name sequences *[foo,0,99]* and *[foo,100,199]* respectively (*foo* represents the name type part of the sequence). A process wanting to send a message to instance number 33 of that service, uses the port name *[foo,33]* as destination address. The TIPC name translation function will find that the indicated instance is within the range bound to by partition A, and directs the message to A's socket.

There are very few limitations on how name sequences may be bound to sockets. One may bind many different sequences, or many instances of the same sequence, to the same socket, to different sockets on the same node, or to different sockets anywhere in the cluster.

4.1 Binding Scope

Although complete location transparency is desirable and sufficient for most applications, there must be ways to control this property for those who may need to do so. Hence, when binding a name sequence to a socket, it's possible to qualify it with a *binding scope* parameter, indicating how far the knowledge of the binding should be distributed in the network. The typical behavior is to spread it to the nodes in the binder's cluster, but it is possible to extend the scope to the whole zone, or limit it to the local node.

4.2 Lookup Domain

Similarly, a client may indicate a *lookup domain* for a message or connection setup request. This is a TIPC network address not only indicating where the lookup, i.e., the translation from a port name to socket address, should first be done, but implicitly even the lookup algorithm to be used.

Two such algorithms are available: 1) *round-robin* lookup is used when the lookup domain is non-zero and there is more than one matching server. Internally TIPC selects the server from a circular list; which root entry is stepped between each lookup. 2) *Closest-first* lookup is used when the lookup domain is zero. Here, the translation is always performed at the client's node and will first look for a matching socket on the local node. If none such is found, the algorithm will successively look for matches elsewhere in the cluster and finally in the whole zone.

5 Reliable Functional Multicast

Functional addressing is also used to provide a *reliable multicast* service. If the sender of a message indicates a port name sequence instead of a port name as destination, a replica of the message is sent to all sockets bound to a name sequence fully or partially overlapping with that sequence (Figure 4).

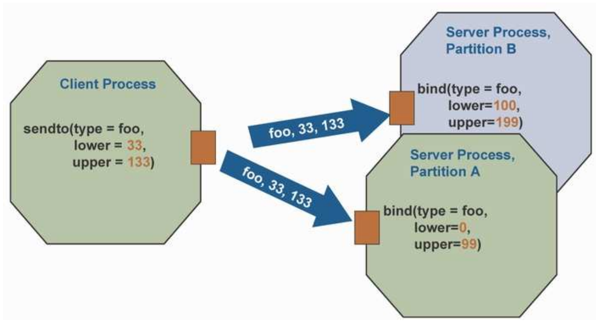
Client Process
sendto(type = foo, lower = 33, upper = 133)
foo, 33, 133
foo, 33, 133
Server Process, Partition B
bind(type = foo, lower=100, upper=199)
Server Process, Partition A
bind(type = foo, lower=0, upper=99)

Figure 4: *Reliable Functional Multicast*

Only one replica of the message is sent to each identified target port, even if it is bound to more than one matching sequence. Whenever possible, this function will make use of the multicast/broadcast properties of the carrying media. In such cases, reliability is ensured by a special *reliable cluster broadcast* [4][5] protocol implemented internally in TIPC.

6 Name Translation Table

Translation from port name to socket addresses is performed transparently and on-the-fly via an internal translation table, replicated on each node. When a socket is bound to a port name sequence, a corresponding table entry is distributed to all nodes within the binding scope, i.e., the local cluster in most cases.

7 Topology Services

TIPC also provides a mechanism for inquiring or subscribing for the availability of port names or ranges of port names.

7.1 Functional Topology Service

This *functional topology service* is built on and uses the contents of the local instance of the name translation table.

To access this service, a user makes a blocking or nonblocking request to TIPC, asking it to indicate when a name sequence within the requested range is bound to or unbound. The request is associated with a timer, giving the duration of the subscription. A timer value of zero causes the call to return or issue a subscription event immediately, making it a pure inquiry, while a value of -1 makes it stay forever, indicating every change pertaining to the requested name sequence.

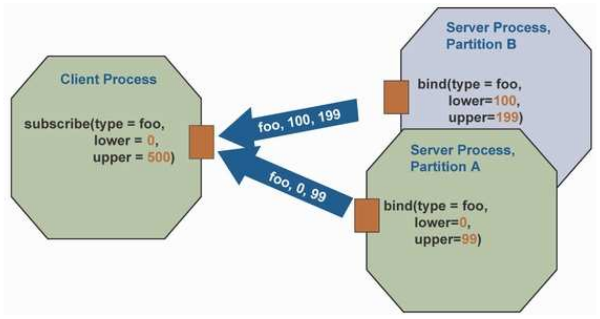

Figure 5: *Functional Topology Subscription*

Figure 5 illustrates this service: If the client process (see also example in Figure 3) wants to syncronize itself with the servers before starting any communication he issues a *subscribe()* call to TIPC, telling it to indicate when a server overlapping with the subscribed range becomes available. Since both ranges of partition A and B are within the given range *[foo,0,500]*, the client will receive two such indications, informing about the exact range of the new bindings. If there is only a partial overlap, e.g., if the client should subscribe for *[foo,0,150]* instead, he will only be informed about the actual overlap, i.e., *[foo,100,150]* for partition B.

7.2 Physical Topology Service

The physical network topology may be considered a special case of the functional topology, and can be kept track of in the same way. Hence, to subscribe for the availability/disappearance of a specific node, a group of nodes, or a whole cluster, the user specifies a dedicated port name sequence, representing this function and the range of nodes he wants to subscribe for. A special name type (zero) is used for this purpose, while the lower and upper boundaries are represented by TIPC network addresses—as described earlier, those are in reality 32-bit numbers.

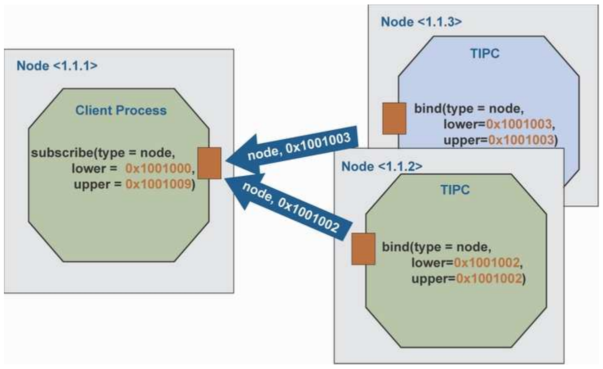

Figure 6: *Physical Topology Subscription*

In the example in Figure 6, the client process subscribes for the node range [0,9] within zone number 1, cluster number 1. Hence, when node <1.1.3> (i.e., zone 1, cluster 1, node 3) establishes a link to the client's node, the client will immediately be informed about this. For this particular service, TIPC will by itself bind/unbind the corresponding port name as soon as it discovers or loses contact with a node.

8 Lightweight Connections

The number of active user connections within a big cluster may be extremely large, and each cluster node must be able to establish and terminate thousands of such connections per second.

8.1 Simple Setup/Shutdown

To deal with this dynamism, TIPC connections are made very lighweight, in reality leaving the user to decide the setup/shutdown sequence. The protocol as such does not specify how connections are established and shut down, so an application caring about performance is free to use its own scheme, e.g., only exchanging payload-carrying messages.

For convenience an alternative, TCP-style connection type is also provided on Linux, with exchange of hidden protocol messages and stream-oriented data exchange.

8.2 Reactive Connections

TIPC connections are highly reactive and give the users almost immediate failure indication if anything should happen at the endpoints, or to the media between them. This is due to a connection supervision and abortion mechanism, which takes advantage of the properties of the local operating system to detect process crashes, or the status of the concerned links to detect node crashes or carrier failure. When any of this happens, a special *connection shutdown* message is spontaneously generated by TIPC and sent to the affected endpoint or endpoints, along with an appropriate error code. This error code delivered up to the user in the failure indication. In some cases, when the failure is detected due to inability to deliver a message, the original message is returned to the sender along with the error code, to further enable him to analyze the situation and take proper action. This *message rejection* mechanism is also used when connection-less messages are undeliverable.

9 Link-Level Protocol

Assuming that most clusters are relatively static in size, some of the tasks normally performed at the transport protocol level have been moved down to the signalling link level.

9.1 Link-level Retransmission

Implementing the retransmission protocol at this level has several advantages. First, it gives better resource utilization since all packets, connectionless and connection oriented, are funneled into one single packet sequence per node pair. Each packet can hence carry the acknowledge of many received packets, regardless of their origin, and we need not keep transmission buffers longer than strictly necessary. Second, packet losses can be detected and restransmission performed earlier than would otherwise be the case. Third, packet delivery and sequentiality guaranteed at the link level eliminates any need for per packet timers at the transport level—a background timer per link is sufficient to ensure those properties. As a result, we obtain a packet flow that is both smoother and more "traffic driven" than with corresponding transport level protocols, which often rely on timers to keep traffic running.

9.2 Link-level Node Supervision

Internode connectivity is also ensured at the link level. First, a background timer for each link endpoint supervises the traffic flow on the link and initiates a probing procedure if the peer is silent too long. Second, if a link is found to have failed after probing, there is a mechanism to steer its traffic over to the remaining link to the same node, if there is one.

9.3 Link-level Redundancy and Load Sharing

In fact, having two links and two carriers between each node pair is considered the normal configuration when using TIPC, as it eliminates any single point of failure in the communication service. The failover procedure used on such occations is completely transparent to the users, and complies to the same QOS as is guaranteed by each individual link: no message losses, no duplicates, and in-sequence delivery. The relationship between dual links is configurable; while full load sharing is the default behavior, an active-standby scheme is also supported.

Detection time for a failed link, and consequently for a crashed node, is configurable and is by default set to 1500 ms in the current implementation.

10 Automatic Neighbour Detection

Signalling links may be configured manually, but this is a tedious task if the size of a cluster runs up to dozens or even hundreds of nodes. Therefore, TIPC uses a designated neighbour detection protocol to establish links between nodes. Within a cluster this protocol is very simple. Each starting node uses the multicast or broadcast capability of the carrying media to tell about its existence, and expects a corresponding unicast response from all nodes recognizing it as part of the cluster.

Between clusters, both multicast and a unicast "pilot" link may be used, and results in a link pattern where each node in one cluster has links to a configurable (default two) number of nodes in the other cluster.

11 Performance

The performance figures we have are from the Linux-2.4 version of TIPC. We have not yet been able to do code optimizations and corresponding measurements on the Linux-2.6 version.

Performance was measured by letting a set of 16 process pairs on two nodes exchange messages in a ping-pong like manner at full speed. This ensures that the CPUs always runs at 100% load, and we can assume that almost all execution time is spent on transferring TIPC messages. We measured the time it took to exchange a message of a certain size 16 X 10 000 000 times, and divided the obtained value with number of messages. The result gives pure CPU execution time per message, automatically excluding latency times on the network and in the OS's sceduling queues, which is anyway the same for all protocols. For comparison, a similar measurement sequence was done for TCP, on the same OS and hardware.

Table 1 shows measured execution time for transferring a message process-to-process between two 750 Mhz Pentium III based nodes. The communication media used was two parallel 100 Mb Fast Ethernet switches.

Msg Size [bytes]	TIPC [μs]	TCP [μs]
64	25	38
256	29	42
1024	44	52
4096	176	178
16384	704	716
65408	3200	2800

Table 1: *Inter Node Execution time (send + receive) for TIPC and TCP messages*

The overall result shows that TIPC is around 35% faster than TCP for inter-node messages smaller than Ethernet MTU, while performance is about the same for larger messages. A similar measurement, where all processes were kept on the same node, showed that TIPC is about four times faster (6 μs vs 25 μs) than TCP for 64 byte intra-node messages; the difference decreasing linearly with message size. At 64 Kbyte messages performance was even here almost the same.

12 Implementation

12.1 Source Code

The latest implementation on Linux is available as a source code package of 12,500 lines of C-code from [1]. It compiles into a loadable module of 167 Kbyte for the Linux-2.6 kernel, and it requires no kernel patches to be installed. This version, just as an earlier one for Linux-2.4, is stable, but still has some limitations. Most notably, only single-cluster communication is supported for now; it is not possible to set up links between nodes in different clusters or different zones.

12.2 Standardization

Open Source Development Lab (OSDL) has defined TIPC as a cornerstone in their Carrier Grade Linux (CGL) strategy, and people from OSDL are contributing actively to the code. TIPC meet several Priority 1 requirements and many Priority 2 requirements in the clustering specifications of Carrier Grade Linux version 2.0 [7]. Within IETF, the ForCES Work Group is considering TIPC to be used as transport protocol between forwarding and control elements in distributed routers. An IETF-draft [4] with a complete specification was presented for the WG at IETF-59 for this purpose.

12.3 Roadmap

The goal is to have TIPC accepted as an integrated part of the Linux kernel in future releases (2.7/2.8). Before the end of 2004, we also want to have it accepted as the preferred protocol for intra cluster transport of the ForCES protocol. Also, before the end of this year, we plan to have developed full support for inter-cluster and inter-zone communication, as well as a redesigned slave node communication framework.

13 Conclusion

Within Ericsson, TIPC has proven to be a very useful toolbox for design of high-availability clusters. It is our hope that this experience will be repeated by others now as the potential of advanced clustering is becoming more widely recognized.

References

[1] TIPC code and documentation `http://tipc.sourceforge.net`

[2] Postel, J., Transmission Control Protocol, RFC 793 `http://www.ietf.org/rfc/rfc0793.txt`

[3] Stream Control Transmission Protocol, RFC 2960 `http://www.ietf.org/rfc/rfc2960.txt`

[4] Maloy, J., Transparent Inter Process Communication, IETF Draft `http://www.ietf.org/internet-drafts/draft-maloy-tipc-00.txt`

[5] Guo Min: TIPC Reliable Multicast Design

`http://www.linux-ericsson.ca/papers/tipc-multicast/tipc_multicast.pdf`

[6] Maloy, J., Make Clustering Easy With TIPC, Linux World Journal April 2004 `http://www.linux.ericsson.ca/papers/tipc_lwm/index.shtml`

[7] OSDL CGL Requirement Definition 2.0 `http://www.osdl.org/lab_activities/carrier_grade_linux/documents.html`