The Design of the Transis System* **

Danny Dolev*** and Dalia Malki†

Computer Science Institute
Hebrew University
Jerusalem
Israel

Abstract. Transis is a high availability distributed system, being developed in the Hebrew University. It supports reliable group communication for high availability applications. The system provides enhanced services for information dissemination and replication in a dynamic environment where machines may crash, for arbitrarily long periods, and may recover; where the network may partition and re-merge. Transis contains novel protocols for reliable message delivery, it optimizes the performance for existing network hardware, and offers a variety of different handles to upper applications. The paper presents the experience gained in the design and the implementation of the Transis communication subsystem.

1 Introduction

This paper presents the design and implementation of the Transis communication subsystem [3]. Transis supports reliable group communication for high availability applications. The system provides enhanced services for information dissemination and replication in a dynamic environment where machines may crash, for arbitrarily long periods, and may recover; where the network may partition and re-merge. Transis contains novel protocols for reliable message delivery, it optimizes the performance for existing network hardware, and offers a variety of different handles to upper applications.

The communication needs of distributed applications have increased considerably in recent years. Today, applications require high speed advanced communication services that support intricate interactions. An application programmer requires an advanced tool that disseminates information reliably and efficiently to multiple destinations. The tool must sustain high communication rates while automatically regulating the communication flow. Such a tool must be optimized for each available hardware, and relieve the programmer from adapting the program to different communication platforms.

The Transis approach is novel is several ways:

* This paper will appear in the proceedings of Dagstuhl Workshop on Unifying Theory and Practice in Distributed Computing, September 1995.

*** Elec mail: dolev@cs.huji.ac.il

† Elec mail: dalia@cs.huji.ac.il

** This work was supported in part by GIF I-207-199.6/91

- Transis employs a reliable multicast protocol, based on the Trans protocol [22], that utilizes the available hardware broadcast or multicast, in order to disseminate messages to multiple destinations with minimal overhead. Performance results show that, indeed, the protocol can sustain extremely high communication rates among a large number of participants.
- The system tackles partitions in the communication network and provides a consistent view of the system state to all the system components[5]. Transis guarantees that members of the same group who remain connected receive the same set of messages in the same order, despite all faults. When partitions recover, Transis implements a novel methodology for providing consistent merging. The principles that underlie the Transis approach for handling partitions have been adopted by several additional projects already: Totem [5] and Horus [28].
- The system regulates the flow of messages in the network to prevent message loss, as much as possible, and avoid flooding. The flow control mechanism is novel in that, unlike point-to-point flow control methods that regulate the flow among each pair of processes separately, Transis regulates the network-wide flow.
- Service layers built on top of the transport layer within Transis enable the use of a consistent message dissemination despite partitioning. Thus, upon re-merging, all processes recover their joint consistent state.

The Transis protocols are optimized for applications that require message dissemination within broadcast domains. Transis saves in message traffic to multiple destinations. Transis has a provision for WAN, though it is not implemented in this version of the prototype: The Transis design proposes to model the system as composed of a collection of *multicast clusters*, as depicted in Figure 1. Each multicast cluster represents a domain of machines that are capable of communicating via hardware multicast. In reality, such a cluster could be within a single Local Area Network (LAN), or multiple LANs interconnected by transparent gateways or bridges. We do not elaborate further on WAN protocols in this paper.

Transis is a modular and extendible system. Many higher level services can benefit from the underlying transport services. For example, in the current prototype, Transis supports a *Persistent Replication Service*, designed in collaboration with the Horus project [21]. The Persistent Replication Service Layer (PRSL) provides the application builder with *long term* services such as message logging and replaying, and reconciliation of states among recovered and reconnected endpoints.

The persistent message replication service that is built within Transis provides a fast and clean development path for distributed application builders that need to replicate information among a subscribed set of targets. The PRSL takes care of all message replication and recoveries within the group of machines

[5] When the network *partitions*, we use the term *component* to refer to each connected subnet.

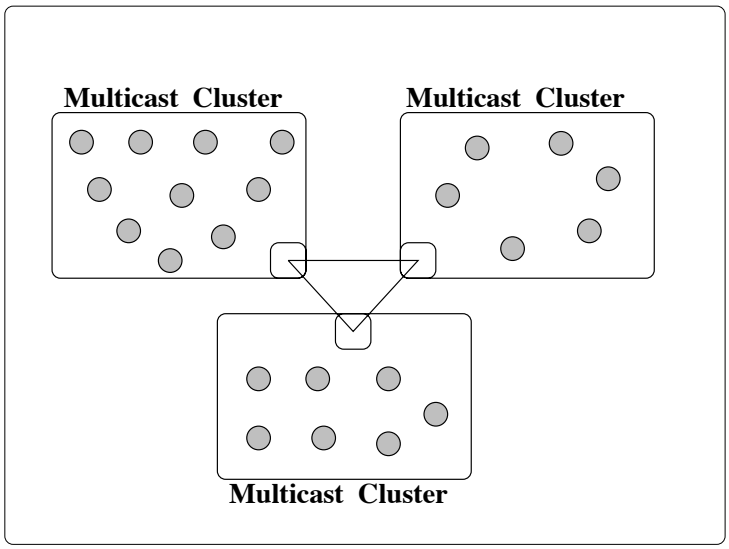

Fig. 1. The Transis Communication Model

spanning through all membership changes. The PRSL provides various levels of consistency guarantees, such as message ordering and stability. PRSL contains two algorithms implementing persistent and ordered message replication, optimized for two extremes:

- Minimizing end-to-end communication. In this optimization, applications require to exchange end-to-end acknowledgment only after a group membership-change (see [4]). Thus, as long as no changes take place,the overhead is minimized.
- Guaranteed majority progress. In this optimization, a primary partition service guarantees that whenever a primary partition of machines is connected for sufficiently long, they will be able to proceed, despite all past failures (see [17]). In other systems, certain sequences of failures or a complete system crash might lead the system to block until full recovery, whereas in our service, any recovered majority is guaranteed to be viable.

2 Reliable Group Communication Services

The construction of distributed applications such as replicated information systems, message-based parallel processes, and on-line conferencing tools, can be made substantially easier with a strong group-communication substrate. The merits of group communication services is substantiated by vast research in this area in recent years (*e.g.*, Isis [10], Horus [29], Totem [5], Psync [25] and Consul [23], Amoeba [15], [18], Delta-4 [26], and many others). The idea is that a

communication substrate can be made to work on new networks, making the porting of the applications easy. The optimizations required for sustaining high throughput communication are made by the architects of the communication layer, and allow for specialized optimizations, that any application programmer can benefit from.

The Transis project addresses the question: What are the properties required of such a communication substrate?

The basic need underlying many distributed applications is the need to deliver the same message to a set of destinations *reliably*. In point-to-point communication, the definition of reliability is obvious: If both parties stay operational, then a message from the first party will eventually reach the second party. In the context of group communication, there may be several possible interpretations to the reliability requirement. For our needs, we choose an *all-or-none* guarantee: if any member of the destination set delivers a message, then all the other members of the set will deliver it, despite some pre-defined failures (but not despite *all* kinds of failures).

Often, the all-or-none guarantee is misunderstood, or oversimplified: the fact is that, in an asynchronous environment, no protocol can guarantee true all-or-none semantics despite arbitrary failures, without saving messages on stable storage. If it could, the act of message-delivery would require common-knowledge, which is impossible to attain in an asynchronous environment [14]. In Transis, we therefore provide two levels of reliable service:

Atomic: An Atomic message delivered at any member is guaranteed to be delivered at all the currently operational members of the application, despite message omissions. However, if any member crashes or disconnects soon after the atomic message is transmitted or delivered, then it might not deliver the message.

Safe: A Safe message delivered at any member is guaranteed to be delivered at all the currently operational members of the application, provided that they do not crash. The Safe service is strictly stronger than the Atomic service, because even if the system partitions, the message is guaranteed to be delivered to them.[6]

The practical question is what is the cost incurred by reliable delivery: The loss rate exhibited by today's networks is extremely low. Therefore, the overhead in recovering lost messages is not high. On the other hand, the message loss rate may increase significantly under heavy network loads. Therefore, in order to prevent excessive message loss, the system must regulate the flow of messages in the entire network, and coordinate the transmission of messages by different processors.

One of the lessons we learned in the design of Transis, is that in order to implement a flow-controlled communication layer within local area networks, with early omission-detection and failure notification, one needs a reliable multicast

[6] Note that in the literature, this service is sometimes called *uniform*, and Safe messages are called *stable*.

layer. Thus, even if some application does not require *atomic* or *safe* delivery guarantees, it benefits from the enhanced performance of the reliable multicast substrate.

2.1 Multicast Message Ordering

Some applications, notably, applications based on the State Machine approach, require that all the members of a group deliver the messages in total order. For example, when messages are totally ordered, a replicated information service can perform updates locally as soon as they are delivered from the communication layer, without need for further coordination. Transis provides the following ordered multicast service:

Agreed: The Agreed multicast service guarantees that any two messages m, m' delivered to multiple destinations, will be delivered in the same order everywhere. This means that either m is delivered before m' or vice versa, but there is no mixing of their orders at different sites.

Agreed messages in Transis maintain this requirement even when partitions occur. Thus, in case detached components continue operating shortly after a partition occurs, consistency is preserved.

For applications that do not utilize totally ordered multicast, Transis provides a weaker form of ordering:

Causal: A causally ordered multicast service preserves the *causal* order of messages (see [19]). Intuitively, causal communication guarantees that a response to a certain message will never be delivered before the message.

Causal-preserving delivery order makes programming somewhat easier, much in the same way that FIFO guarantees make it easier to program two-party interaction.

2.2 Failure Notification

Suppose that machines may crash, and the network may partition. In networking environments, it is currently impossible to distinguish for sure between machine crashes and network partitions; thus, machines may appear crashed, but in fact they are only detached.

The communication substrate needs to provide failure notification to the upper layer, informing it when machines in the configuration appear detached. Furthermore, in order to provide for efficient *state transfer* upon recovery, the failure notification mechanism should indicate when, in the stream of messages, the process group has detached, and when the partition has recovered. To this end, Transis provides failure notification and guarantees by the *virtual synchrony* programming model, and its extension to partitionable operation, see below (Section 2.3).

2.3 Virtual Synchrony and Partitionable Operation

Transis provides the application programmer with a programming environment that is conceptually *Virtually Synchronous*, as defined by Ken Birman *et. al* in the early work on the Isis system [7], and extended into partitionable environments in [24]. The virtual synchrony model and its extension to partitionable operation encompasses the relation between message passing operations in a process-group, and between control messages provided by the system about process failures and joins in the group.

A process group in Transis is dynamic: A group is created with some initial membership, and subsequently its membership changes as processes join (are added) and leave (are deleted or fail). Whenever the membership of a group changes (and initially, created), all the processes of the new membership observe a *membership change* event. This event is provided as a special Transis message, a *membership change message*.

The essence of the virtual synchrony programming model is in guaranteeing that membership changes are observed in the same order by all the members of a group. Furthermore, membership changes are totally ordered with respect to all the regular messages in the system. This means that every two processes that observe the same two consecutive membership changes, receive the **same** set of multicast messages between the membership changes. For example, let's consider a group that changes from the configuration $\{A, B, C\}$ to $\{A, B, D\}$. Then processes A and B will first receive the membership-change indication of $\{A, B, C\}$, then they both receive exactly the same set of regular messages, and finally they receive the second configuration change, $\{A, B, D\}$. (In this case, C might receive, after the first configuration change, any subset or superset of the regular messages that A and B received). In this sense, membership changes are virtually synchronous, as the processes have identical contexts when messages arrive. This allows the processes to act upon the messages they receive in a consistent way. For a formal definition of the virtual synchrony model, refer to [7].

As an important extension to the Isis virtual synchrony model, Transis allows *partitionable operation*: If a group partitions into two components, such that communication between the components is impossible, then each component continues observing the virtual synchrony model separately. Furthermore, upon re-merging, the merged set will be virtually synchronized starting with the membership change event that denotes the joining. More details on the semantics of the partitionable membership are given below, in Section 4.

3 High Performance Reliable Multicast

We experimented with several protocols and ideas for supporting reliable multicast communication. We found out that the way to support reliability with high performance is based on several principles:

- In systems that exhibit low loss rate, it is preferable to use a negative-acknowledgment based protocol. Thus:

- Messages are not retransmitted unless explicitly asked to, through a negative acknowledgment.
- Positive acknowledgment, required for determining the arrival of messages to all their destinations, are piggybacked on regular messages that go to the required destination. In case no regular message is transmitted, then periodically, an empty message containing only acknowledgment and an "I am alive" indication will go out.

These ideas are not new, and are utilized also, in various forms, in [25, 22, 16, 6]. Their importance is great in today's networks, that exhibit extremely low message loss rates.

– Detection of message losses must be made as soon as possible. Suppose that machines A, B and C send successive messages. If machine D maintains reliability guarantees against each machine separately, and misses the message from A, then it will not detect the loss until A transmits another message. If, on the other hand, there are additional relationships between messages sent by different processors, then D can possibly detect the loss as soon as B transmits its message. Early detection saves on buffer space by allowing prompt garbage collection, regulates the flow better, and prevents cascading losses.

  In the Transis project, this principle led to two separate versions of the system: One, based on the Trans protocol [22], relies on causal relationships among messages (see [3]), and the other relies on a revolving token that forms a total order on message transmission events (see [5]).
– Our protocols rely on very low message loss rates of the networks. However, under high communication loads, the networks and the underlying protocols can be driven to high loss rates. For example, we conducted experiments using UDP/IP communication, between Sun Sparc machines, interconnected by 10-Mbit Ethernet; Under normal load, the loss rate is approximately 0.1%, but under extreme conditions, the loss rate went up to 30%. Such loss rates would make the recovery from message losses costly. Furthermore, this can cause an avalanche effect, where under high loads, the reliable communication protocols further increase the load to overcome omissions. Therefore, to prevent this situation, it is crucial to control the flow of messages in the network.
– In order to achieve extremely high throughput (approaching the physical limits of the network), the pipelining principle needs to be exploited: One or more machines can "feed" messages to the network in a rate that approximates its maximum capacity. But if the machines delay (*e.g.*, for acknowledgment) between successive transmissions, then they fail to utilize the full network capacity.

Generally, we note that, optimizations done at the low levels of the system usually (not always) make the system much faster than it would be, had the entire role been that of application developers. Thus, even though the system (as really, any service) may provide *more* than what is needed by some specific application, it may well still lead to an overall better implementation of the

useful parts of the service. In addition, it employs system-wide flow control considerations, instead of being limited to a per-application basis.

3.1 Flow Control

Transis employs a novel method for controlling the flow of messages and for bounding the amount of memory consumed by the protocol. Define a *network sliding window* as consisting of all the received messages that are not acknowledged by all the machines yet. Each machine computes this window from its local information. Note that this window contains messages from *all* the machines in the current Transis configuration, unlike the traditional sliding-window that maintains only the machine's own messages. The network sliding window determines an adaptive delay for transmission by the window size, ranging from the minimal delay at small sizes and slowing up to infinite delay (blocked from sending anything but "I am alive" messages) when the window exceeds a maximal size. The window cannot remain stuck for long, because the background membership algorithm will remove from the configuration machines that do not participate (see the Membership Section below). This releases the sliding-window block and the flow of messages resumes. The network sliding window roughly determines the maximum number of messages that need to be kept for retransmission.

3.2 Agreed Multicast

One of the characteristics of the Trans and the Transis protocols, is that they allow completely spontaneous transmission of messages by any machine. Consequently, two machines may send messages within a small interval apart, none receiving each other's message first. In this case, there will be no acknowledgment between these messages. This means that additional processing is required if there is a requirement to deliver the messages in the same total order at all their common destinations.

The Agreed multicast service guarantees that messages arrive reliably and in the same total-order to all their destinations. Since we currently have three versions of Transis that differ in their implementation of the Agreed multicast service [12, 5], we chose to present the tradeoffs in this issue.

There are several completely distributed algorithms that build a total order ¿from the local information and reach agreement [22, 12, 25]. It is perhaps easiest to understand the *all-ack* algorithm of Transis, that is also completely distributed. The above referred algorithms are essentially optimizations on this principle. The all-ack idea is:

- Wait until at least one message is received from each machine.
- Then go through the machines in ascending order, and deliver the first message from each machine unless it directly acknowledges another message.

The common characteristic of these algorithms, is that they do not incur any extra message exchange for achieving agreement on the total order. They have *post-transmission* delay, from the time a message is transmitted and received until it is ordered in the right place. Interestingly, this cost is most apparent when the system is relatively idle, and waiting for responses from all (or some) of the machines incurs the worst-case delay. On the other hand, these methods can sustain steady transmission loads that are close to the network limits, when all the machines are fairly uniformly active.

A different family of protocols orders the messages in a total order by contending for an ordering capability to order messages [9, 5, 16]. The Isis ABCAST protocol [9] employs a *token-holder* within each group of communicating processes. ABCAST messages are multicast at will, and their delivery is delayed by all the receiving processes except for the token holder. Periodically, the token holder sends a message indicating its order of delivery for all received ABCAST messages, and all the other processes comply with it. The Token may also migrate to the sender.

The Amoeba system contains a different variation of this scheme, implemented within the operating system kernel [16]. A sequencer kernel is designated as the central controller. Every message is sent to it via point to point communication, and the sequencer multicasts it to all the machines. The FIFO order of sequencer-transmissions determines a total order for all the messages.

The Totem protocol [5] uses a revolving token that holds a sequence-number for messages. The holder of the token can emit one or more multicast messages, and update the token sequence accordingly. In order to transmit a multicast message, a processor must obtain the token. The token itself regularly revolves among all the processors.

The cost in these protocols is in obtaining access to the ordering capability, be it migratable or static. This cost is apparent both in the delay occurring until the control is obtained, and in extra messages exchanged. Once it is obtained, transmission and ordering is done immediately. Therefore, we say that they have a *pre-transmission* delay. The advantage of a control scheme like the revolving token of [5] is that it regulates the flow of messages efficiently.

It is not entirely clear what are the tradeoffs between pre-transmission ordering and post-transmission ordering in these protocols. In particular, the behavior of these protocols when the communication pattern is "chaotic" is an active area for future research.

4 Membership Maintenance

A fundamental issue in the design of a reliable multicast layer is the maintenance of the *membership* of operational machines. Transis contains a membership protocol that is integrated in the communication system, such that the notifications of *membership changes* are delivered to the application among the stream of regular messages [2, 13]. Changes to the membership are coordinated with the delivery of regular messages in the system.

Most membership protocols allow only one component to continue operation, in case of network partitions [27, 11]. In large and critical systems, this approach is not realistic, and it is essential to enable operation in face of partitions. In the case that partitioned operation is not desired, it is easy to add a layer on top of our membership that disallows all but one component to operate. Thus, our protocol does not mandate partitioned operation, but provides more flexibility.

When the network partitions, each component continues operating separately. The machines in each component are in agreement about membership among themselves, but not with the machines in other components. This might sound chaotic, at first. However, we require that:

- Every pair of machines that go through two consecutive membership changes, receive the same set of messages between the two changes. (This is a generalization of the principle called *virtual synchrony*, see [8]).
- Upon re-merging, all the machines in the new membership start with a consistent view of the membership, and agree on the messages that immediately follow it.

In this way, the membership service associates a membership-context with each message. The application can use this to perform consistent operations on received messages, and to merge the histories of joined components. For example, [1] describes a replicated mail server that exploits the Transis membership for efficiently implementing a partitionable service.

Intuitively, at the basis of our membership protocol, there are two stages: *(1) suggest a new membership set, (2) wait for agreement ¿from every machine in the set.* This simple protocol is complicated by the following issues:

- All the machines in a new membership set need to terminate the preceding membership in a consistent way. The problem is that if a certain machine, q, is taken out of the previous membership, the "last" message from q might have reached only a subset of the new membership.
  We use the following rule: Let m_q be a message from q. If **any** machine in the new membership set receives m_q before committing to the new membership, **all** of them deliver m_q. Otherwise, all of them discard it.
- Our design allows the flow of regular messages to continue during transition to a new membership. This design goal is important for handling cascading membership changes, during periods of instability or frequent changes. It also makes the protocol flexible to support messages from external sources (an *open* group communication), because auxiliary sources are not part of the protocol and cannot cease sending messages during its operation.
  In our protocol, the context of regular messages that are sent during membership transitions is determined by their order with respect to the messages used within the membership protocol.

In [13], we present the full solution that supports partitioned operation and rejoining. Joining is done multi-way, in a completely symmetrical fashion. In this way, the joining provides a solution to the *startup* problem as well: Each

machine starts up as a singleton set on its own. Then, all the machines that start up *merge* into a larger set.

5 Higher Layer: Persistent Replication Services

Transis is a modular and extendible system. We envision many higher level services that can benefit from the underlying transport services. In this section, we describe the *Persistent Replication Service* designed for Transis in collaboration with the Horus project [21].

The Persistent Replication Service Layer (PRSL) provides the application builder with *long term* services such as message logging and replaying, and reconciliation of states among recovered and reconnected endpoints.

The PRSL supports multicast messages among members of a designated group. The long-term guarantees made by the PRSL are intended to overcome the limitations of the underlying multicast transport services offered by Transis (and similar systems), namely, that their delivery guarantees are affected by transient failures: When a message gets posted at the underlying transport communication layer, its delivery is only guaranteed within the "currently connected component" of the system. Furthermore, message stability indicates only that the message is deposited at the transport-layers in the destination sites, but no end-to-end acknowledgment at the application level is provided. Therefore, there is no real guarantee that any destination application actually acted upon it.

The PRSL multicast services extend the underlying communication delivery and ordering guarantees within the entire replication group.

The basis of all the PRSL operations is the *replication group*: The replication group is a static set of processes, defined by the user at startup time. Except for the case of startup and shutdown of members (see below), the replication group remains static throughout the execution, and there are no *view-change* reports by this layer. The replication group is distinguished from the *multicast group*, which consists of the currently connected and communicating members of the replication group. The multicast group is transient, and is maintained automatically by the membership mechanism of the transport layer of Transis. As described above, Transis allows the membership of multicast groups to partition, and supports merging automatically, while providing the partitionable virtual synchrony guarantees.

The PRSL supports several operations within the replication group:

- A *uniform* multicast service, guarantees the uniform delivery of messages within the replication group, despite arbitrary crashes and communication failures. By uniform delivery we mean that if any member delivers a message, it must eventually be delivered by all the other members.
- A *totally ordered uniform* multicast, guarantees, in addition to the uniformity guarantees, total ordering of messages throughout the replication group. The PRSL totally-ordered multicast requires making agreement decisions, and may delay the delivery of multicast messages until such decisions are

possible. Typically, this requires a majority of the processes (or, in more general terms, a *primary component*) to be able to reach a consensus decision. The CoRel replication protocol [17], developed for the Transis project, supports total ordering within a primary-component, and provides automatic merging of message-histories upon recovery of partitions. The protocol guarantees that progress of total-ordering decisions is always possible within a primary-component. To the best of our knowledge, this is the only existing protocol that has this desirable property. Within each connected component, CoRel requires an application-level acknowledgment for each message before it can be delivered in total-order.

Another protocol implementing persistent total ordering in partitionable groups is provided in [4]. Their protocol does not require application level acknowledgments for reaching total ordering decisions, and thus does not need to delay the delivery of totally ordered messages until the application handles them. In this protocol, however, it is not always possible to guarantee progress when one or some of the participating members are disconnected (although, in practice, the situations in which the protocol is blocked from making progress are very rare).

- A *stable* multicast service guarantees that messages are delivered to the application only after they are acknowledged by the PRSL software at all the participating members of the replication group. Stable messages behave like the underlying Safe messages, except that their delivery is guaranteed within the persistent group, whereas Safe messages are guaranteed only within the current membership. Additional difference rises when the stable service is used in combination with the explicit-acknowledgment facility of the PRSL(see below): Then, stability reflects user-level acknowledgement, whereas Safety can only reflect acknowledgement by the transport layer itself.
- An *explicit acknowledgment* operation allows the application to withhold message acknowledgment until explicit confirmation. When the application specifies this option, the PRSL software does not acknowledge received messages until after they are delivered to the application and explicitly acknowledged by it. In this way, the stability of messages actually reflects the application-dependent notion of stability, and can allow the application to perform operations on messages before they are considered stable.
- A *shutdown* operation removes a member from the replication group. Unlike lower level membership services, members are never taken out of the replication group in the course of automatic activity, but only at an explicit request by the application. The PRSL guarantees that information about shutdown members reaches all the remaining participating members eventually, and in this way prevents indefinite blocking for acknowledgments from the removed members.

  A system manager may use the shutdown operation (along with *startup*) upon long service outages, when for example it may be more efficient to bring back a failed server by doing a checkpoint-transfer rather than replaying the

entire history of messages that accumulate during the outage.
- A *startup* operation allows bringing up new members into the replication group. When requested to join a new member, the PRSL first guarantees that the new member becomes up to date with the current state of the system, and then guarantees that information about the new process eventually reaches all the existing members.

5.1 Using PRSL

A persistent replication layer is desired in many applications. In any application that needs to replicate information among a persistent set of replicas, the PRSL provides an easy development path: the application developer simply needs to deposit update messages to the layer and specify ordering requirements. The PRSL automatically takes care of propagating the update messages to all the members of the persistent replication group, and preserves the specified ordering requisites.

Our approach allows the application to deposit messages 'asynchronously' to the PRSL without waiting for stability or consistent ordering. We see special advantage in this approach for future mobile and wireless environments: In such environments, often a primary-partition or a particular token site is unaccessible for long durations. Thus, an application can deposit messages asynchronously, such that later, they become stable and consistently delivered to all the participating members.

The advantage in using the underlying Transis communication substrate for supporting the PRSL is twofold: Firstly, during periods of stability, the lower layer transport services provide for dissemination of multicast messages within any connected component of the network. This relieves the PRSL programmer from the task of implementing reliable multicast, and takes advantage of the highly tuned protocols of the communication substrate. Secondly, the recovery from transient partitions and failures is made efficient by the underlying support of the virtual synchrony execution model (extended for partitionable operation). Using this execution model enables each connected component to be represented as a unit upon merging with other components, and prevents reiterating old message by multiple members.

6 Conclusions

Computer communication networks play a crucial role in today's computer environments. Using advanced communication facilities, one can replicate information cheaply, conveniently and more quickly.

The Transis approach to advanced group communication has acquired a wide recognition in the academic community, mainly due to the following desirable properties:

1. It employs a highly efficient multicast protocol, based on the Trans protocol [22], that utilizes available hardware multicast.

2. It can sustain extremely high communication throughput due to its effective flow control mechanism, and its simple group design (for performance results, see [20]).
3. Is supports partitionable operation, and provides the means for consistently merging components upon recovery.

As of approximately two years ago, Transis has been operational at the Hebrew University, and supports various applications as well as the "Distributed Algorithms" course.

Acknowledgements

The Transis project started in early 1991. It evolves with contributions of many students via course projects, MSc theses, PhD work, or just from mere special interest. The main contributors to the project, as of today, are Yair Amir, Idit Keidar, Shlomo Kramer. The project gained from the experience gained by Ken Birman and Robbert van Renesse in developing Isis and HORUS, and by Michael Melliar-Smith and Louise Moser in developing Trans and Totem.

References

1. O. Amir, Y. Amir, and D. Dolev. A Highly Available Application in the Transis Environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France (LNCS 774)*, June 1993.
2. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.
3. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.
4. Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.
5. Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Intl. Conference on Distributed Computing Systems*, pages 551–560, May 1993.
6. K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.
7. K. P. Birman. *Reliable Distributed Computing with the Isis Toolkit*, chapter Virtual Synchrony Model. IEEE Press, 1994. to appear.
8. K. P. Birman, R. Cooper, and B. Gleeson. Programming with Process Groups: Group and Multicast Semantics. TR 91-1185, dept. of Computer Science, Cornell University, Jan 1991.
9. K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991.
10. K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Press, 1994.

11. F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, April 1991.
12. D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *23rd Annual International Symposium on Fault-Tolerant Computing*, pages 544–553, June 1993.
13. D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. submitted for publication. Available as CS TR94-6, Institute of Computer Science, the Hebrew University of Jerusalem, 1994.
14. J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *3rd Annual ACM Symp. on Principles of Distributed Computing*, pages 50–61, 1984.
15. M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *11th Intl. Conference on Distributed Computing Systems*, pages 882–891, May 1991.
16. M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.
17. I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Inst. of Computer Science, The Hebrew University of Jerusalem, 1994. Also available as Technical Report CS95-5. submitted for publication.
18. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. In *9th Ann. Symp. Principles of Distributed Computing*, pages 43–58, August 90.
19. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 78.
20. D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. TR 94-14, Inst. of Comp. Sci., The Hebrew University of Jerusalem, June 1994.
21. D. Malki and R. van Renesse. The Replication Service Layer. internal manuscript, 1994.
22. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel & Distributed Syst.*, 1(1):17–25, Jan 1990.
23. S. Mishra, L. L. Peterson, and R. L. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. TR 91-32, dept. of Computer Science, University of Arizona, 1991.
24. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the Fourteenth Intl. Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994. IEEE. Also available as technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.
25. L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.
26. D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer-Verlag, 1991.
27. A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.
28. R. van Renesse, R. Cooper, B. Glade, and P. Stephenson. A RISC Approach to Process Groups. In *Proceedings of the 5th ACM SIGOPS Workshop*, pages 21–23, September 1992.

29. R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1442, Cornell University, Dept. of Computer Science, Aug. 1994.