Replication over a Partitioned Network

Yair Amir

Ph.D. Presentation

The Transis Project
The Hebrew University of Jerusalem
yairamir@cs.huji.ac.il
http://www.cs.huji.ac.il/papers/transis/yairamir/yairamir.html

This presentation was given at the Hebrew University of Jerusalem in partial fulfillment of the requirements for the degree “Doctor of Philosophy”.

This presentation is given six months prior to thesis submission.

More information about the Transis project can be found using the www in http://www.cs.huji.ac.il/papers/transis/transis.html.

Information about Yair Amir can be found in http://www.cs.huji.ac.il/papers/transis/yairamir/yairamir.html

An ftp site for Transis exists in cs.huji.ac.il under directory pub/misc/transis. It accepts anonymous login. Information about Yair Amir can be found under directory pub/misc/transis/yairamir

Comments are very welcome and can be mailed to yairamir@cs.huji.ac.il

Table of Contents

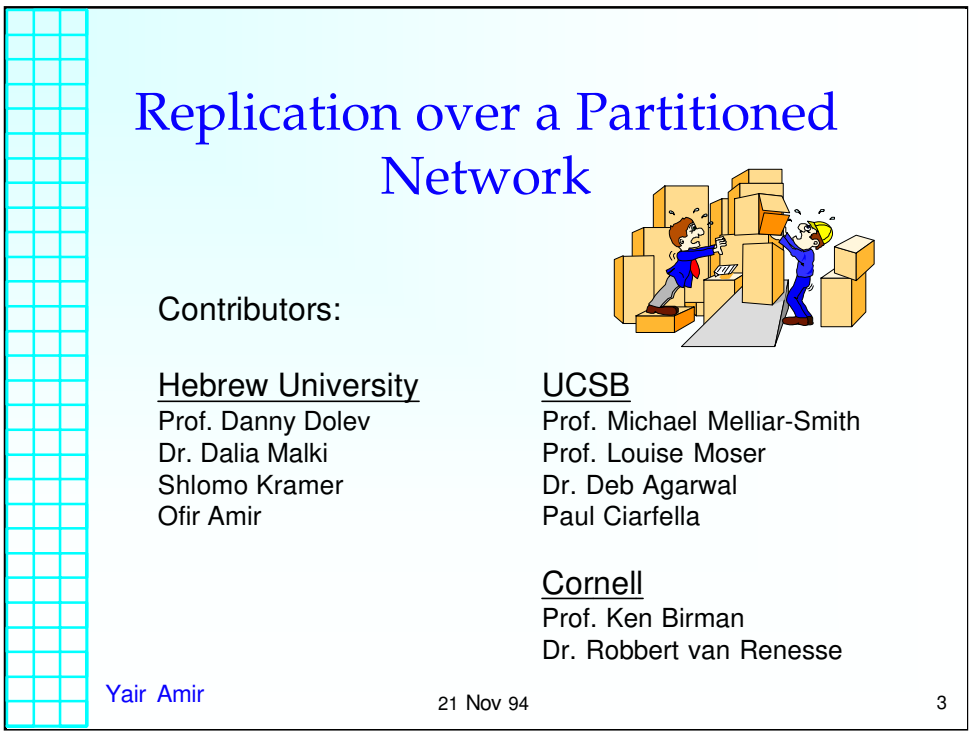
Introduction	3 - 4
The failure model	5
Motivation	6 - 9
The Client-Server model	10 - 12
Existing replication methods	13 - 15
The Architecture	16
Transis: A group communication layer	17 - 19
Extended virtual synchrony	20 - 23
Replication layer concepts	24
Propagation by eventual path	25 - 28
Amortizing end-to-end acknowledgments	29 - 33
Summary	34 - 35

Abstract

We present a new architecture and algorithm for distributed replication. The replication algorithm operates in the presence of message omission faults, processor crashes and recoveries, and network partition and re-merges. The architecture exploits the Transis group communication sub-system to minimize communication costs and to eliminate forced disk writes in the critical path, while preserving complete and consistent operation.

End-to-end agreement is required only after a change in the membership of the connected servers, rather than on a per action basis. Hence, the latency and throughput of updates in the system is determined by the latency and throughput of Transis. In contrast, the latency and throughput of existing database systems is determined by the end-to-end acknowledgments mechanisms, such as two-phase commit, and by forced disk writes. Refer to slide 19 for Transis performance figures.

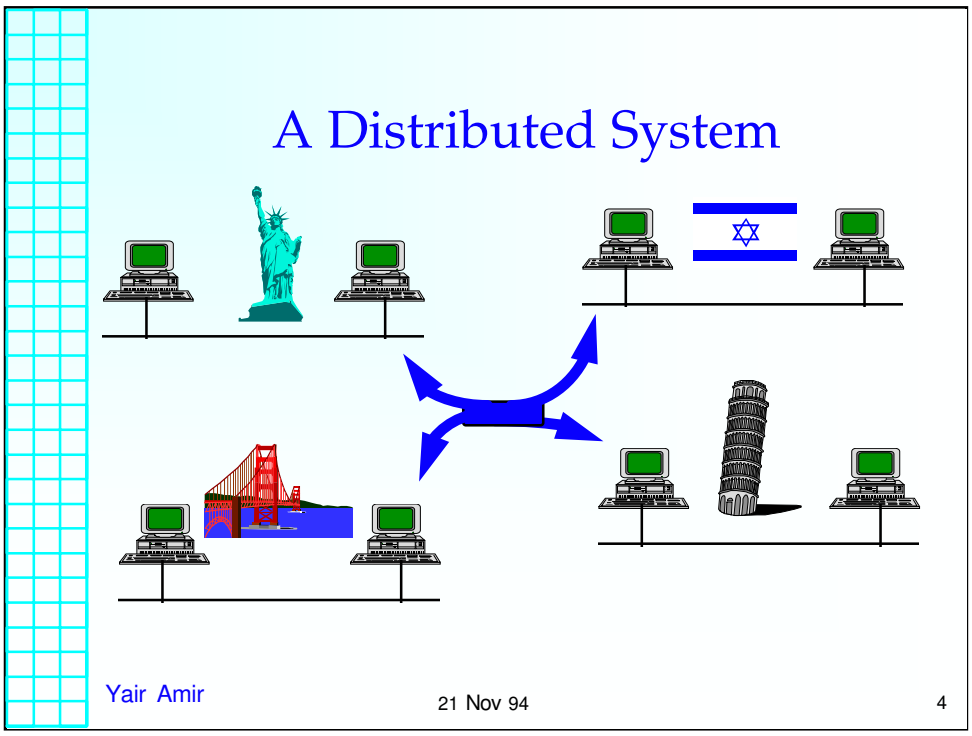
The updates are globally ordered, and if the system has partitioned, they are applied to the database when they become known to the primary component of the partitioned system. An application may, however, read data and initiate updates at any time, even in a component that is not the primary component.



My Ph.D. research is conducted at the Institute of Computer Science, The Hebrew University of Jerusalem under the warm supervision of Prof. Danny Dolev. Together with Dalia Malki and Shlomo Kramer we initiated and developed the Transis project, an advanced group communication transport service (and much more than that). Ofir Amir, in his lab project, helped me prove the feasibility of the replication concept. Discussions with Idit Keidar and others from the Transis group helped refine ideas in this research.

A substantial part of my research was done at Prof. Melliar-Smith's and Prof. Moser's lab at the Electrical and Computer Engineering Dept. University of California, Santa Barbara. During two summers and several mutual visits, Prof. Moser and Prof. Melliar-Smith were involved in almost every aspect of my research. The work with Deb Agarwal and Paul Ciarfella on the Totem protocol contributed a lot to my understanding of high speed group communication. The ring version of Transis is based on the Totem protocol.

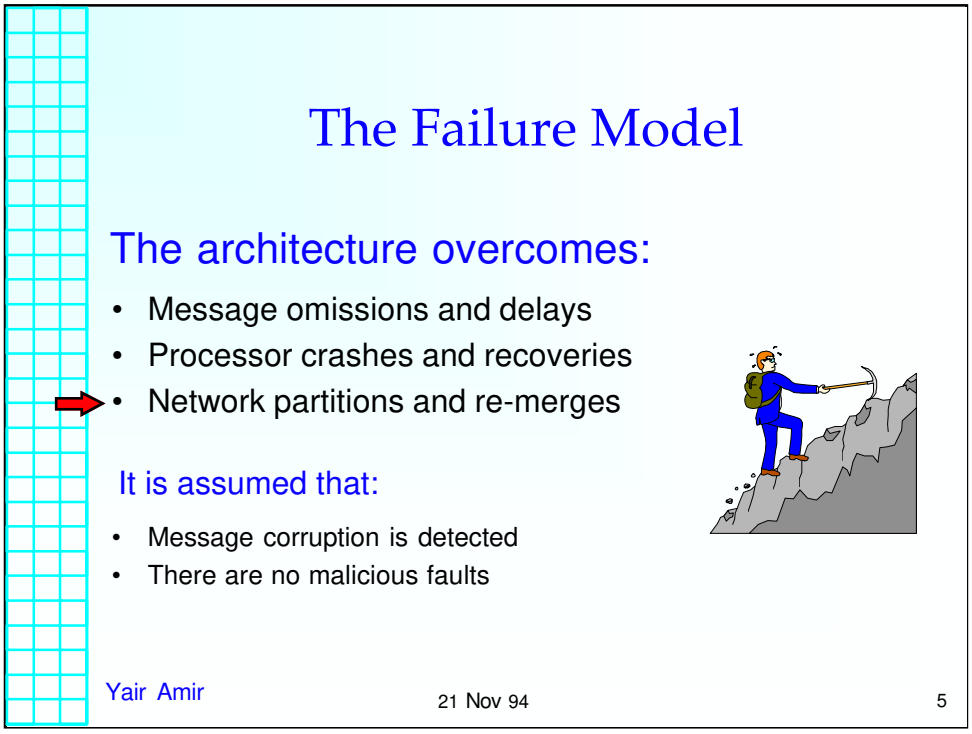
Prof. Birman and Dr. van Renesse from the Computer Science Dept. at Cornell University, were always willing to contribute their valuable advice to the Transis research. They integrated Transis ideas into their new Horus system, and vastly helped to expose them. Spending last summer with them at Cornell gave me many ideas for future research.



The technology presented here is most effective in systems composed of several local area networks, each of which contains several workstations. The local area networks are connected via some wide area network. These networks can be geographically scattered.

Examples for such systems are:

* Stock exchange systems.
* Flight reservation systems.
* Control systems.
* Military systems.



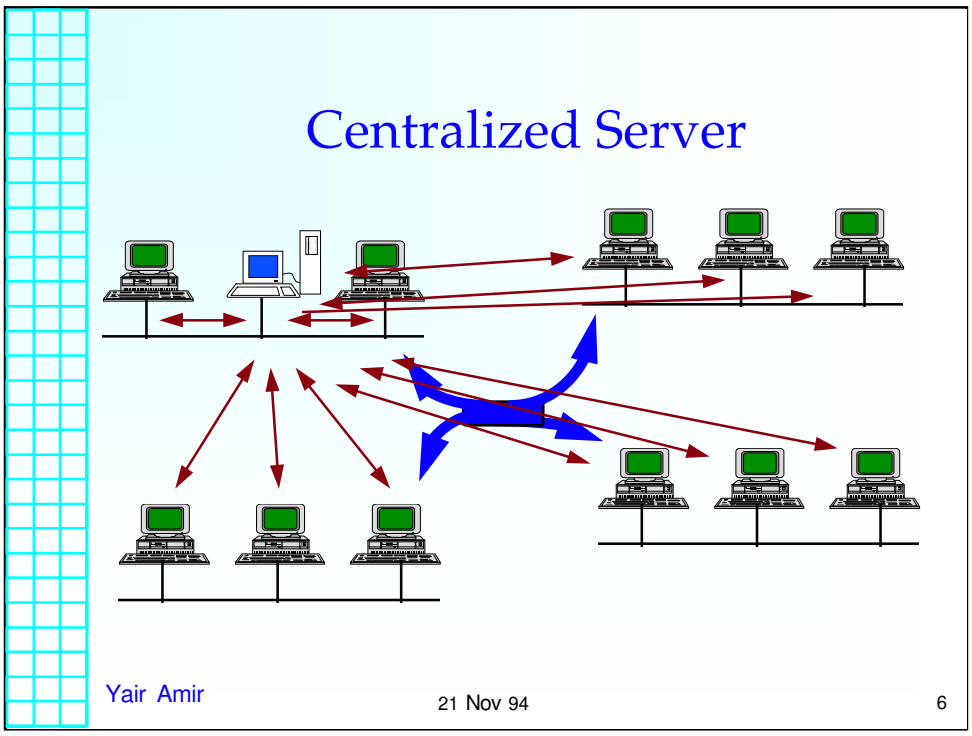
Our fault model contains the following possible faults:

• **Message omission and delay** - messages (or packets) can be lost or delayed when sent over the network. Message omissions occur mainly due to buffer overflow at the receiver. Standard reliable point-to-point protocols such as TCP/IP overcome these problems.

• **Processor crashes and recoveries** - A process can manage data on disks. If this process is killed or abnormally terminates, or if the computer is shut down or incurs a power failure, data on disk still survives. When the process is re-run (i.e. recovers) the data is available again. This model is more general than the Fail-Stop model because we cannot assume that a failed process never recovers. Therefore, we cannot ignore possible decisions a process makes just before it crashes. This fault models reality better than the Fail-stop model.

• **Network partitions and re-merges** - in systems such as the above, network partitioning is likely to happen.

The Transis project was the first group communication system to provide a complete solution for this elaborated and realistic fault model. Practically, if a system is designed according to a specific fault model, and a fault which does not belong to the fault model occurs, the system behavior can be unpredictable. We assume that packet corruption is detected by CRC (or similar) methods and we do not handle Byzantine failures.

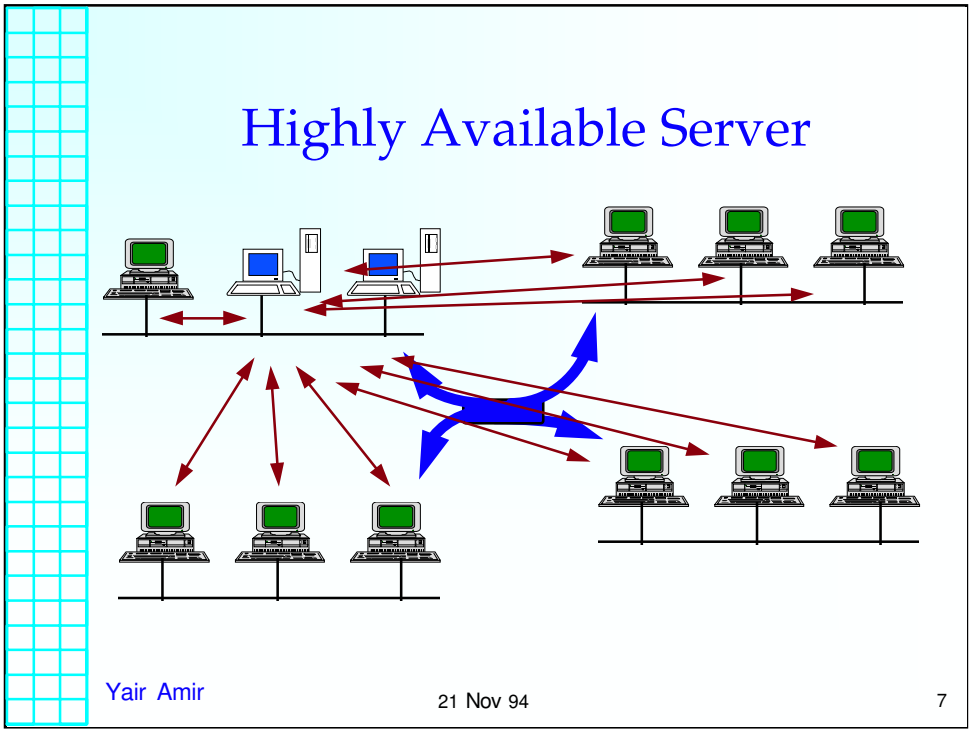


Where to store and manage the data is a major issue when designing a distributed system. A centralized approach keeps the data at one server. This approach is simple. There can be no inconsistencies or contradicting views of the data because it is kept only once.

However, this approach suffers from two major drawbacks:

- Performance problems:
  - The server has to satisfy many clients and therefore can be highly loaded.
  - Communication latency can be high for remote clients.

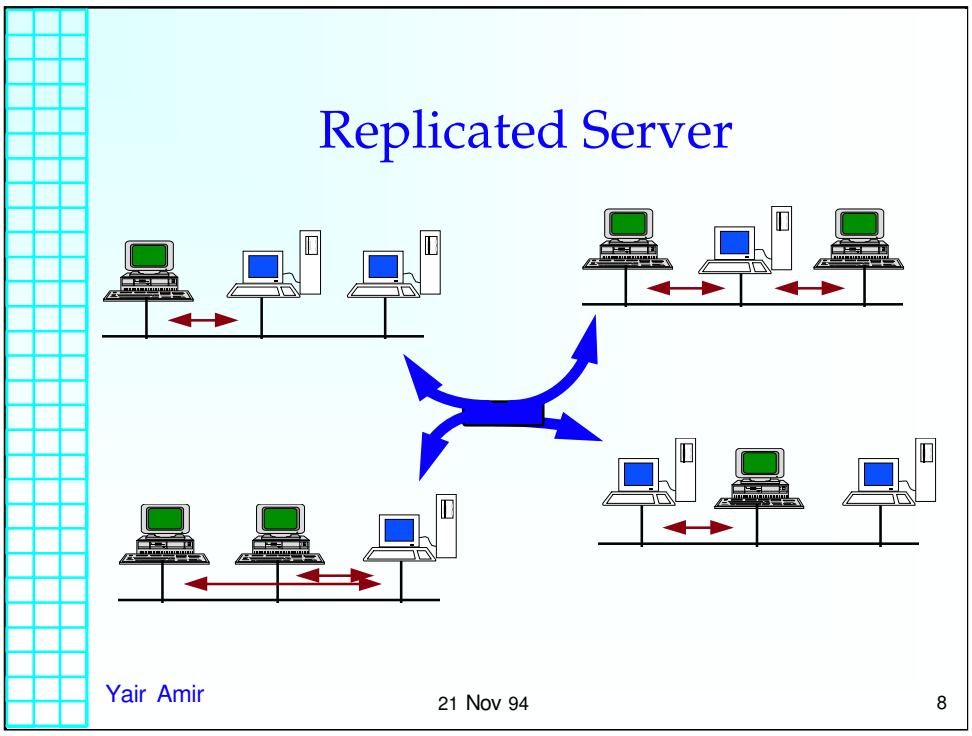
- Availability problems:
  - When the server is down there is no service.
  - Clients that reside in portions of the network that are temporarily disconnected from the server can get no service.



The centralized server can be duplicated in order to create a hot backup. The hot backup usually resides in the same room (or box) as the primary server. The backup takes over when it detects that the primary server fails.

As can be seen, such solutions do not attempt to answer the performance problem. Moreover, they cannot answer the availability problem inherent to the network.

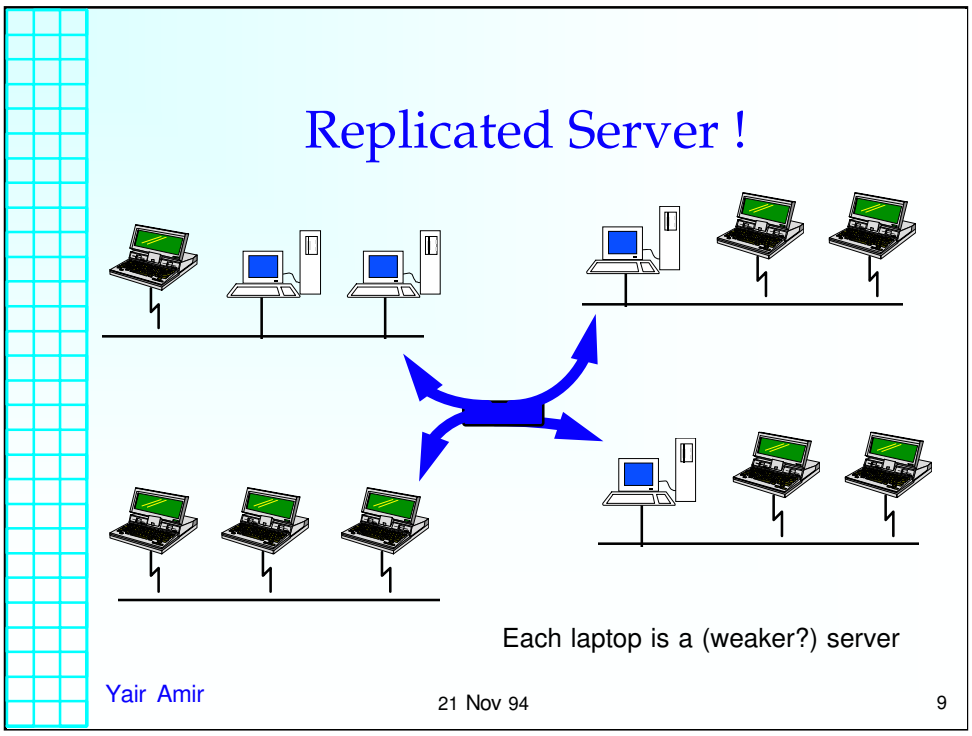
If a “disaster” that destroys the server site occurs, recovery can be problematic.



When the server (and the data) is replicated, each server serves fewer clients. The clients are more local to their server. This saves time and communication for queries. It also makes the service highly available.

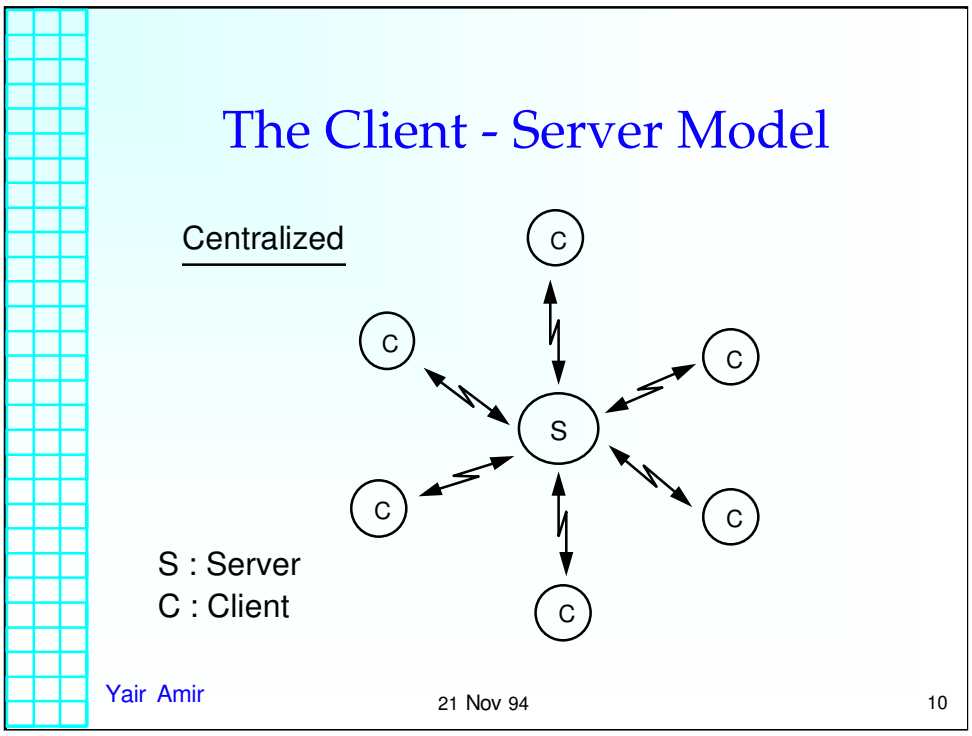
The main problems of this approach are the cost of updates, and assuring system consistency.

This work presents a new method for replication. We will show that consistent updates can be done efficiently using new techniques in both group communication and replication.



The replication approach may appear expensive due to the cost of many servers. Today each PC is potentially a strong server. We define a server to be an autonomous computer that manages data.

In today's world, each laptop has to contain at least part of the data needed by its owner. Allowing a laptop to connect to either one of several main servers that are geographically scattered can be valuable for nomadic systems.

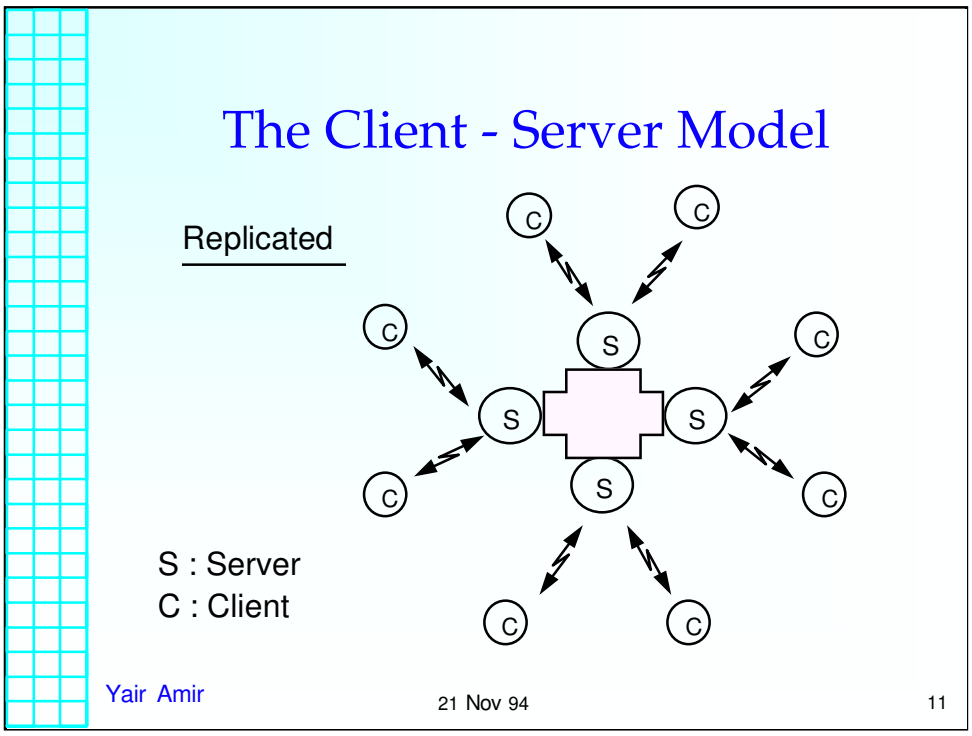


The Client-Server model is a valuable paradigm for designing distributed systems. According to this paradigm, a service is provided by a server process.

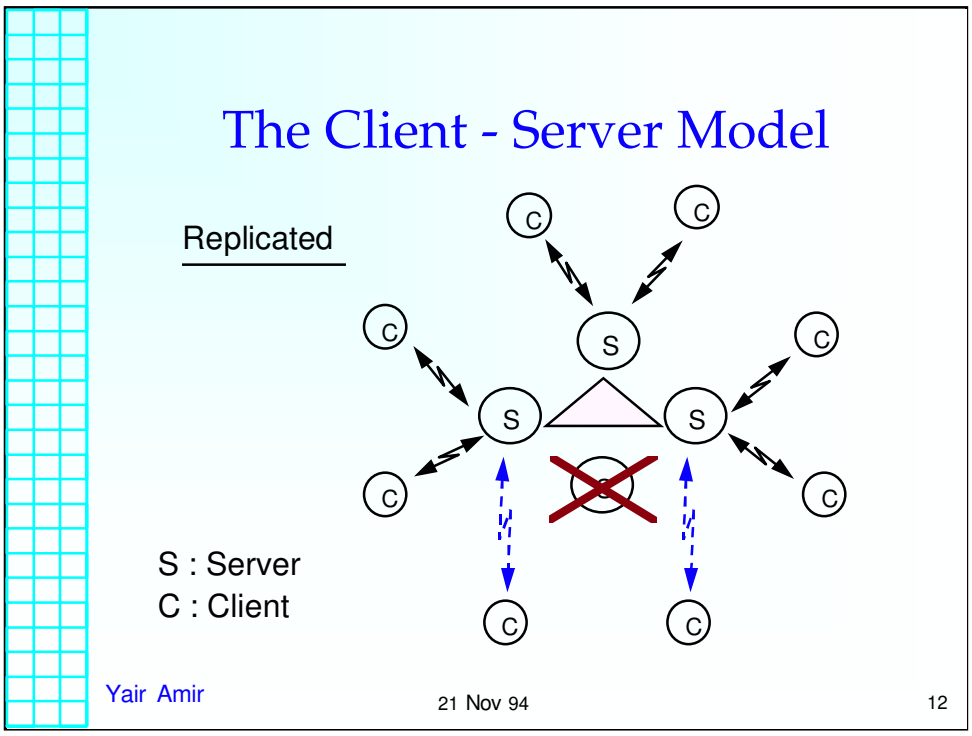
This server manages the data and service needed by client processes.

Clients communicate with the server according to a pre-defined interface. Data structures and methods that are internal to the server are hidden from the clients.

Client processes can reside either in the same machine as the server or in a different machine connected by a network. The method of communication can be any communication protocol interface. Remote Procedure Call and Unix sockets are good examples.



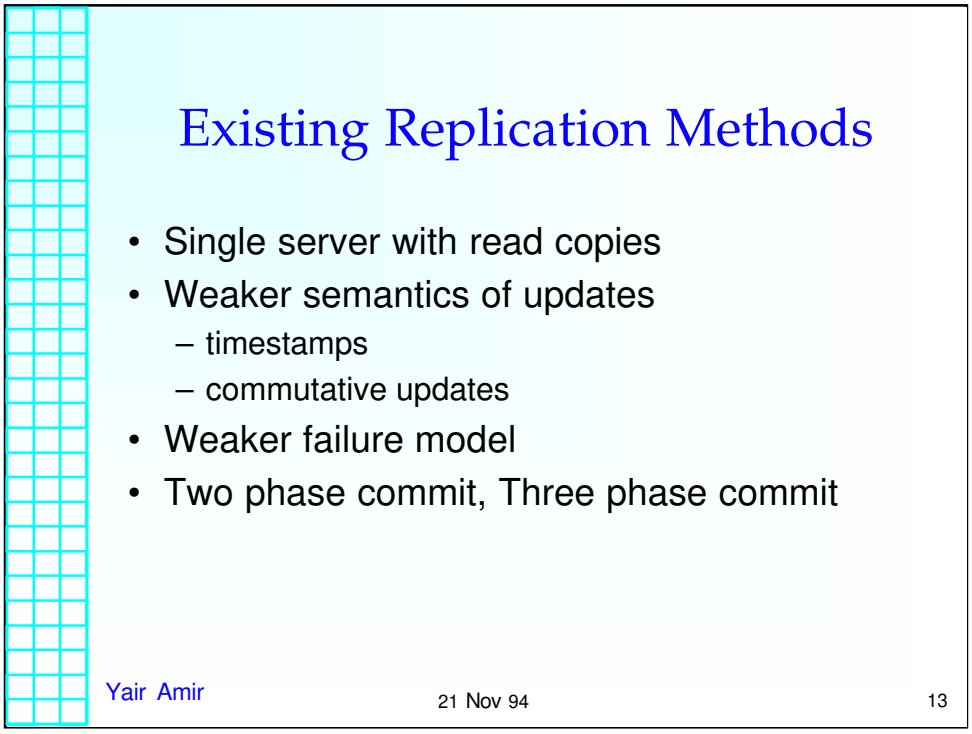
This work conforms with the Client-Server model. We show how a replicated server can be designed to work efficiently. We focus on the protocol between the servers that creates the replicated service.



When one server crashes, its clients can connect to a different server and continue the work. When the server recovers, it exchanges information with other connected servers and continues to service clients. Group communication is used for the interaction among servers.

Usually, a point to point protocol is used for the client-server interaction. However, group communication can be useful for the client to locate and pick a server to connect with. This allows for optimizations regarding which server serves which clients, load balancing, locality, and other criteria.

Alternatively, the servers can decide which server answers which client, and which server takes over a failed or disconnected server. This server connects to the clients of the failed server.



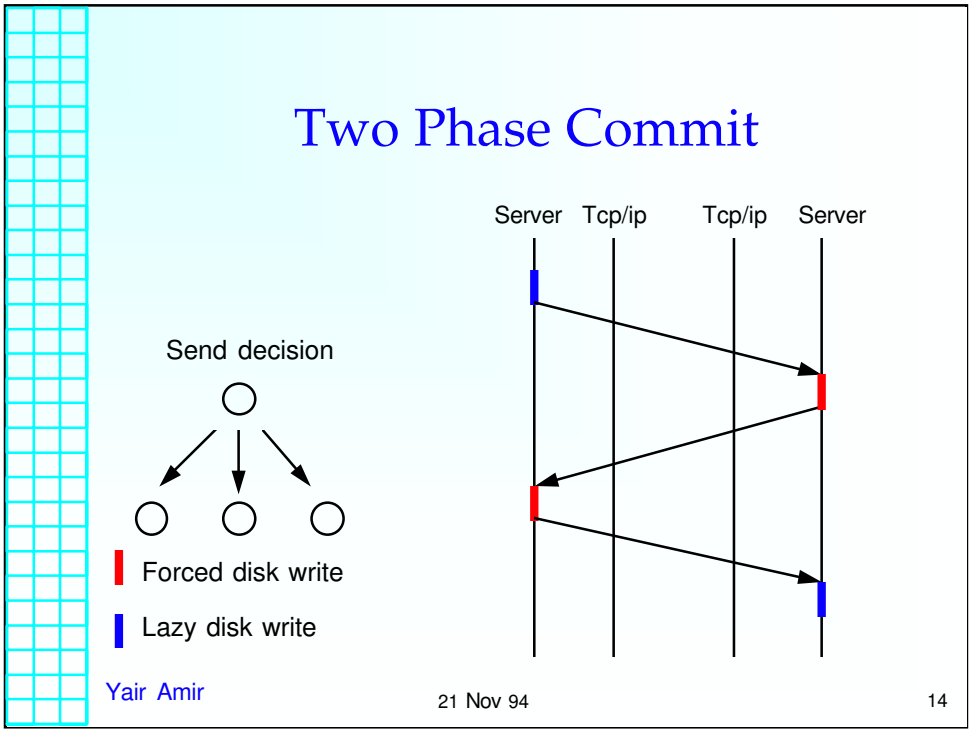
• **Single server with read copies** - updates are made to the master and queries can be made to the master or to each of the copies.

The master updates the copies using either a time interval strategy or on a per action basis. The master holds the consistent view of the data. If the client is not connected to the master it can not initiate updates.

• **Weaker update semantics** - if all updates are commutative, it is sufficient to apply the same set of updates to each of the replicas in order to achieve an identical state. i.e. the order with which the updates are applied is not important. This greatly simplifies the maintenance of consistent replication.

• **Weaker failure model** - If we assume no network partitions, then an unconnected replica can be considered as crashed. The ISIS system assumes "no partitions". This greatly simplifies the problem. However, assuming no partitions when more than one local area segment is involved, is not realistic. Assuming partitions means that a disconnected server is potentially partitioned.

• **Two phase commit and Three phase commit -** are protocols that are used to coordinate transactions in distributed databases. They can be used also for consistent replication.



Two Phase Commit is useful for coordinating transactions that span several sites in a distributed system. As a special case, it can be used to maintain consistency of a replicated database. Here, we refer to it only in this sense.

Upon an update request from a client to its server, the following steps take place:

1. The (one) server that received this update from a client lazily writes it to its disk, initiates an update action and sends it to the other servers using some form of a point-to-point communication (typically TCP/IP). The initiating server is called the "coordinator" of this action.

2. Each server that receives the message, forces it to disk. Only after it is physically written, can that server send an acknowledgment message (again, using TCP/IP) to the coordinator.

3. After the coordinator receives an acknowledgment from each of the servers, it forces a commit decision to disk. After the decision is physically written, the coordinator sends a commit to the other servers, again, using TCP/IP.

4. Each of the servers lazily applies the update to its copy of the database.

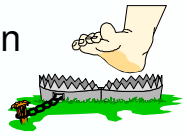
If even one of the servers fails to answer, the coordinator aborts the update.

To overcome this drawback, a more expensive protocol, Three Phase Commit, **usually** requires an acknowledgment from only a majority of the servers in order to commit. However, it requires an additional round of communication.

Traditional Replication Methods

Problems:

- Utilize point-to-point communication
- High level agreement per action
  - Inefficient use of disk
  - High latency
- Vulnerable to faults at critical points
- (Therefore, practically) not fully consistent



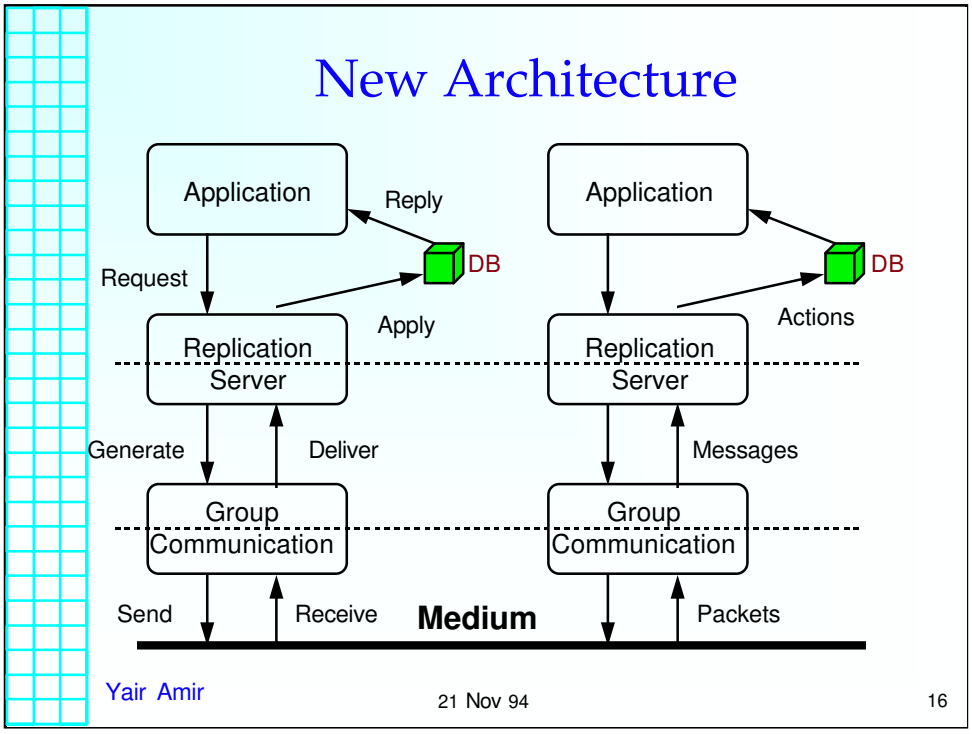
• Using TCP/IP, Decnet, and OSI reliable point-to-point communication for replication is not scaleable and does not utilize existing hardware capabilities for (non-reliable) multicasting. For example, Two Phase Commit (serially) sends each update *n* times for *n* replicas each round.

• When forced disk writes are used on a per action basis, the disk is bound to tens of writes per second. This results in high latency for disk writes. The need for *n* replicas to wait for their disk magnifies the problem.

• In the previous slide, If the coordinator detaches or crashes at step 3, the replicas can not commit or abort the action. We have to remember that at this stage they hold locks on data. These locks were acquired at step 2, before sending the first acknowledgment. Theoretically, they may not release these locks until they communicate with the coordinator. This leads to possible blocking of the whole system.

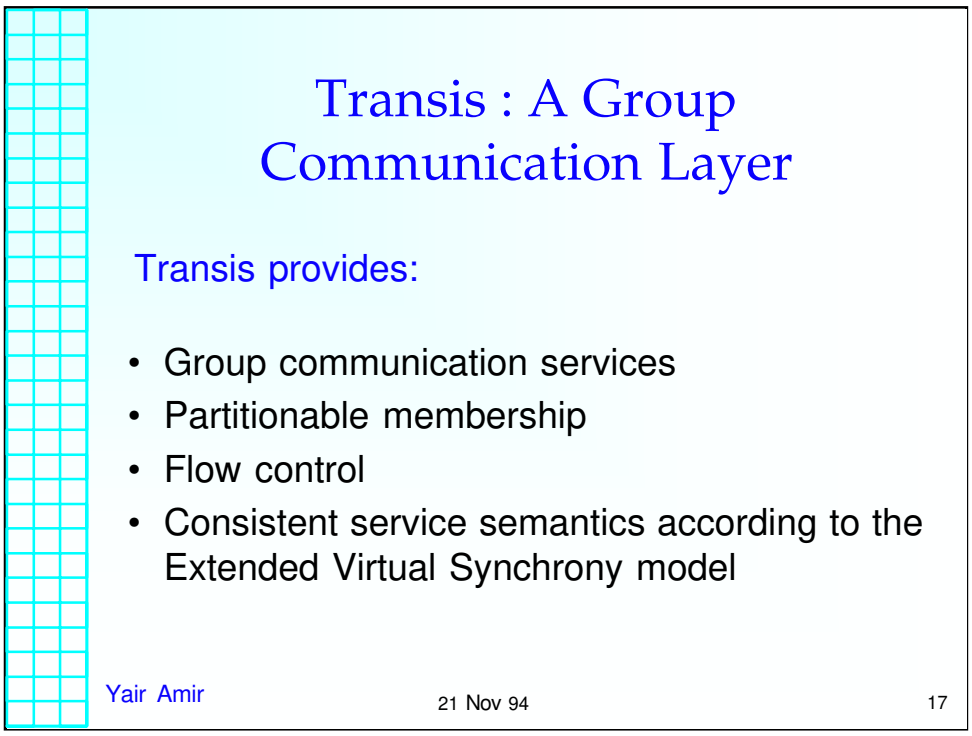
• Most practical implementations of database replication can not afford to block the system at situations similar to the above. Therefore, they presume either abort or commit of the action, thereby loosing full consistency.

For these reasons, replication is seldom used. Even when it is used, it is mainly used to increase the availability of a service, rather than to boost performance.



We present a different architecture for active replication, which drastically lowers the cost of replication, making it an appealing tool to boost performance, as well as to improve the service availability. This architecture carefully tailors advanced group communication techniques to a new replication algorithm.

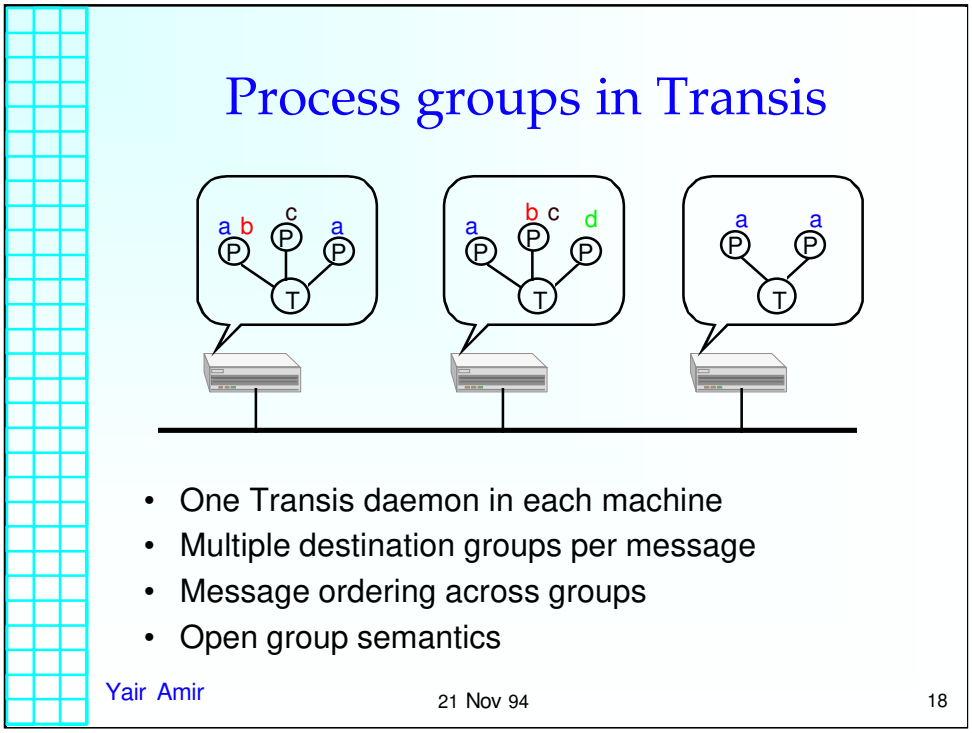
The application requests an action to be performed on the database. The action can be a query, an update, or a combination of both. The replication layer generates a message containing this action and uses the group communication layer to send the message (which might be broken into several packets) across the network. The group communication layer delivers the message to all currently connected servers. The replication layer guarantees that the same set of actions, at the same order, will be applied to each of the replicas even in face of network partitions and merges, and process crashes and recoveries. The group communication layer overcomes message omissions.



The following slides give a short overview of the Transis system.

We developed Transis, a group communication sub-system, and use it to develop several tools that address fundamental problems in distributed systems. In particular, we use Transis in our replication architecture.

We focus here on the Transis services, performance, and service semantics. The algorithms and internal protocols used in Transis deserve a presentation of their own and are not part of this presentation.



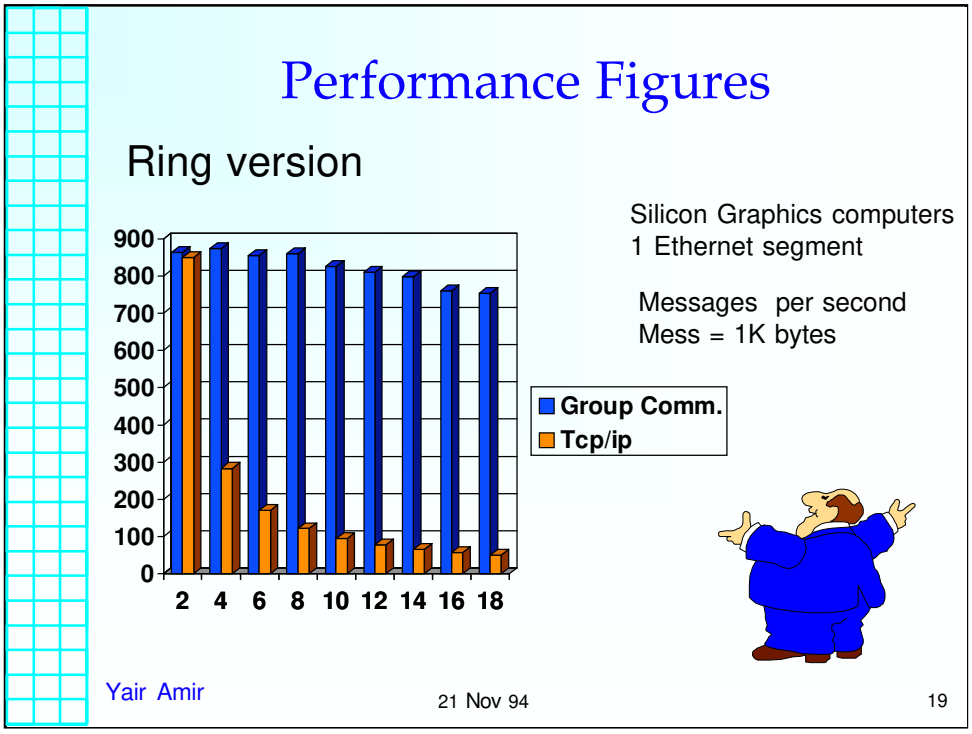
Each processor that may have processes participating in the group communication has one Transis daemon running. All the physical communication is handled by the Transis daemon. Each Transis daemon keeps track of the processes residing in its processor. The Transis daemons keep track of the processors' membership. This structure is in contrast to other group communication mechanisms, where the basic participant is the process rather than a daemon per processor.

The benefits of this structure are huge:

• Flow control is maintained globally, rather than on a per group basis.

• The membership protocol is invoked only if there is a change in the processors' membership. When a process crashes or joins, the daemon sends a notification message. When this message is ordered, the daemons deliver a membership change message containing the new group membership to the other members of the group.

• Order is maintained globally and not on a per group basis. Therefore, message ordering across groups is trivial. Moreover, it is easy to implement "open" group semantics (i.e. processes that are not members of a group can send messages to this group).



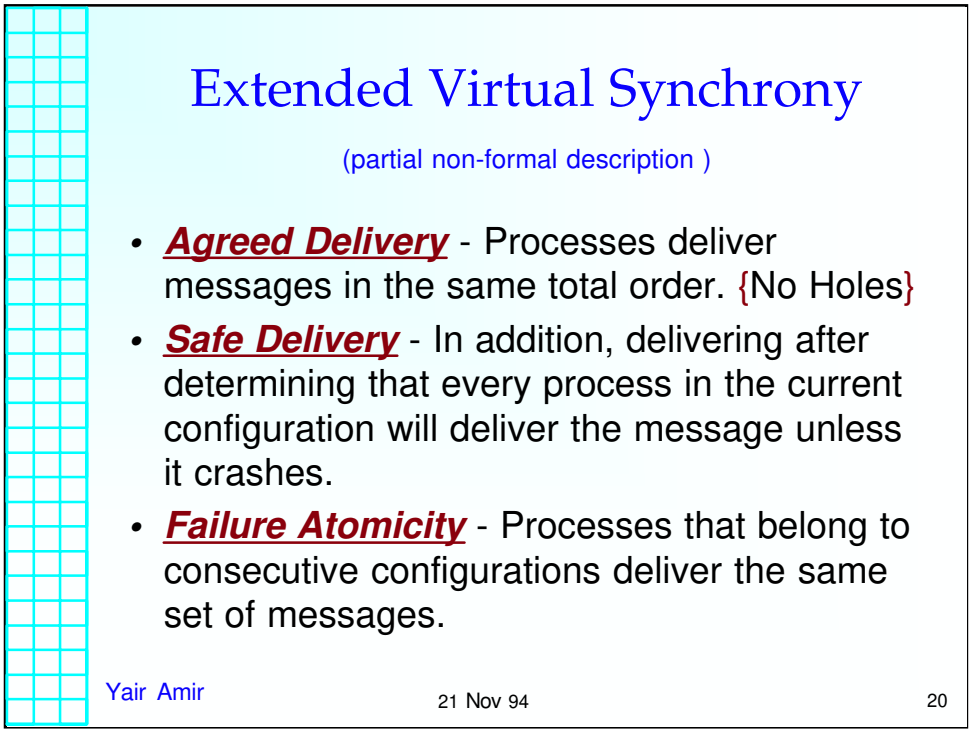
The measurements were taken on an unloaded system when Transis flow-control was tuned for best throughput.

Another important factor is latency. We measure latency starting from the time a message is generated (given to Transis at the sending processor), until the time it is delivered by Transis. Using the ring version of Transis, the latency for Safe delivery is about two rounds time, and the latency for Agreed delivery is about half a round time. In the above measurements, a round took about 92,96,106 milliseconds for a segment of 6,10,18 processors respectively.

If the focus is on latency, we can achieve a round time of 2*n milliseconds for n processors, when the desired throughput is around 400 messages per second. In this case, the latency is about 6,10,18 milliseconds for Agreed delivery and about 24,40,72 milliseconds for Safe delivery, for a segment of 6,10,18 processors respectively. Note that even 400 messages of 1Kbytes each, per second, is more than what most other group communication sub-systems achieve for reliable delivery.

When the desired throughput is around 100 messages per second, the latency is about 4,6,10 milliseconds for Agreed delivery and about 16,24,40 milliseconds for Safe delivery, for a segment of 6,10,18 processors respectively.

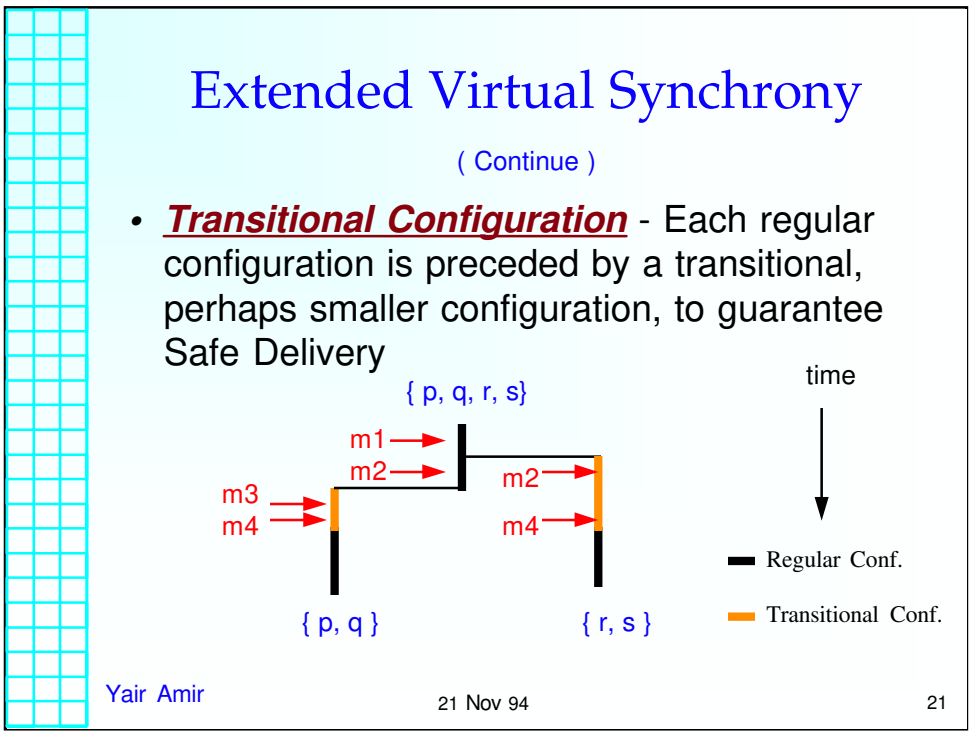
Refer to the next slide for the definitions of Agreed and Safe delivery.



The word "Process" in the Extended Virtual Synchrony slides refers to a Transis daemon.

Safe delivery guarantees that when Transis delivers a message (that contains an action) to the replication server, all the other Transis daemons in the current configuration already have this message and will deliver it to the upper layer unless they crash (**even if at this point the network is partitioned**). In our architecture, Transis delivers messages to the replication server.

When the configuration of connected processes shrinks, one cannot tell whether the disconnected processes crashed, or they are only partitioned away. However, Safe delivery guarantees process *p* that delivers message *m* that all the other processes in its configuration have *m*.

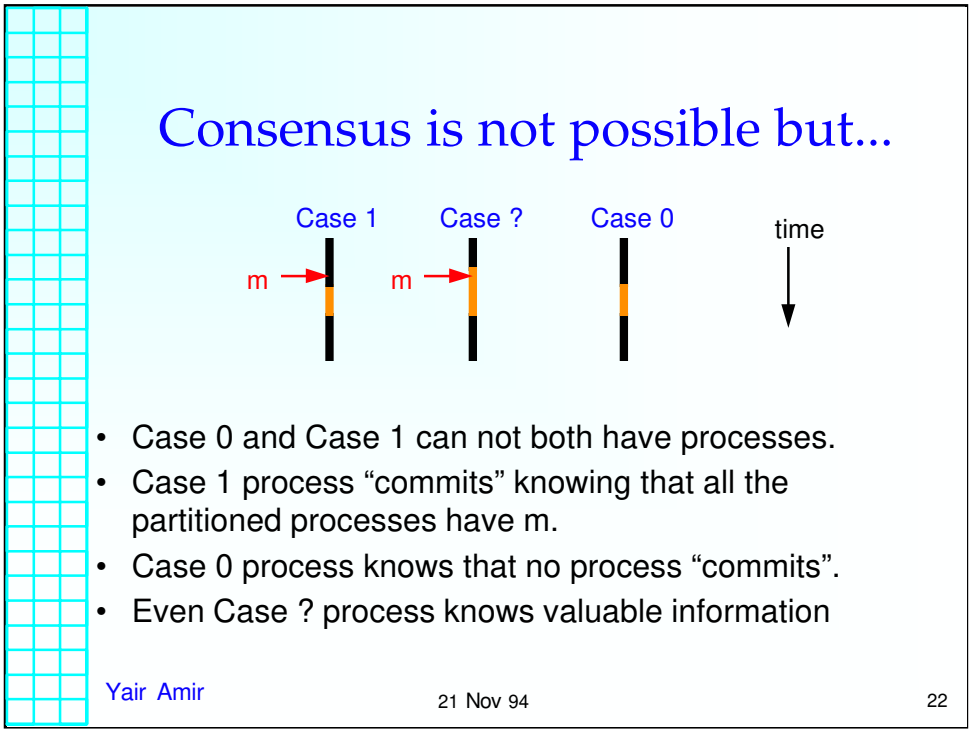


In an asynchronous system that is prone to partitions and crashes, we can not guarantee that a message meets the safe delivery criterion in the configuration in which it was sent (refer to the example).

In order to meet the requirement for safe delivery, we introduce the notion of *transitional* configuration. The transitional configuration contains processes (i.e. Transis daemons) that belong **both** to the (regular) configuration that precedes this Transitional configuration and to the one that follows this Transitional configuration. For these processes that are transitioning together from a (regular) configuration to the next (regular) configuration, the safe delivery criterion **is** guaranteed.

The slide presents an example for a network partition. Processes *p,q,r,* and *s* belong to a (regular) configuration. All four processes know that *m1* was received by each of them. Therefore, They all deliver *m1* in the regular configuration. *p* and *q* know *m2* and also got *r*'s and *s*'s acknowledgment for *m2*. Therefore *p* and *q* deliver *m2* in the regular configuration. However, *r* and *s* can not tell whether *p* and *q* have *m2* or not. They can only guarantee safe delivery for the Transitional configuration, therefore they deliver *m2* in that configuration. *m3*, sent by *p,* is not known by *p* and *q* to be received at *r* or *s*, and, as it appears, that message was lost (remember message omission?).

More details can be found in the next two slides.



It is well known that the consensus problem is not solvable in an asynchronous system that is prone to faults.

The safe delivery together with the transitional configuration provide a valuable tool to deal with faults and still be consistent in such a system.

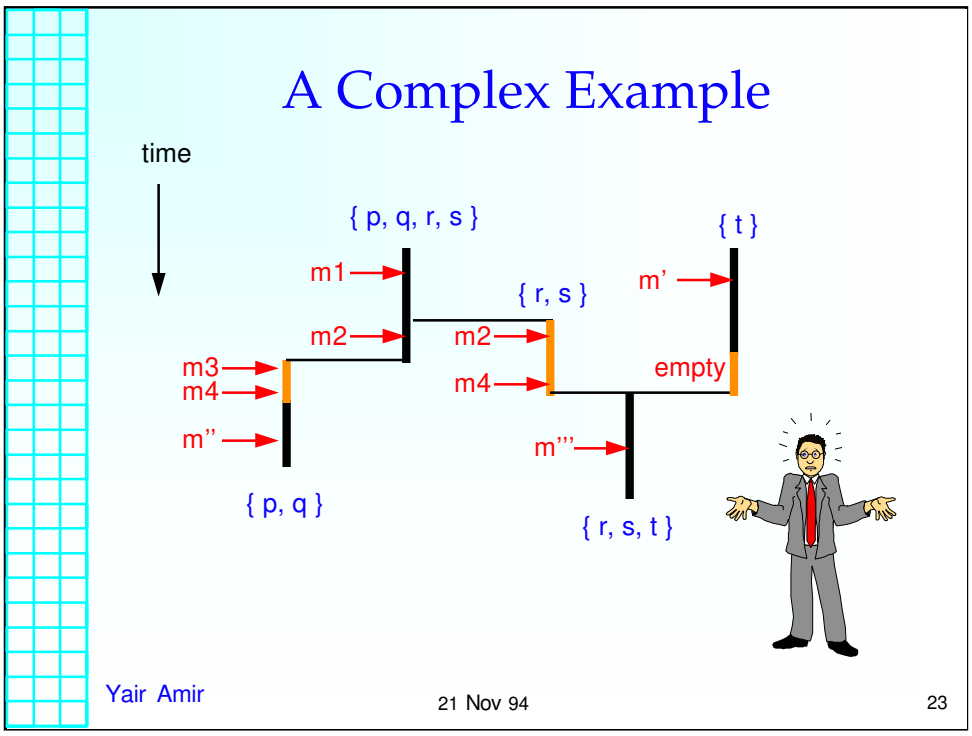
Instead of having to decide on one of two possible values (0 or 1), we have three possible values: (0, ?, or 1).

Case 1 reflects a case where a message is received in the regular configuration. **Case ?** reflects a case where a message is received just before a partition occurs and we cannot tell whether other components of the configuration (that splits) have that message. In this case it is delivered in the Transitional configuration.

Case 0 reflects a case where a message is sent in the regular configuration just before a partition was detected but it was not received by members of a detached component. These members are Case 0 processes.

The key here is that Case 0 and Case 1 can **never** coexist.

If later on a Case ? process meets a Case 1 process - it can "commit" and decide on 1. On the other hand, if it meets a Case 0 process it decides 0.



In this scenario, *r* and *s* split from *q* and *p,* and immediately merge with *t.*

The scenario for messages *m1, m2, m3,* and *m4* is identical to the previous example.

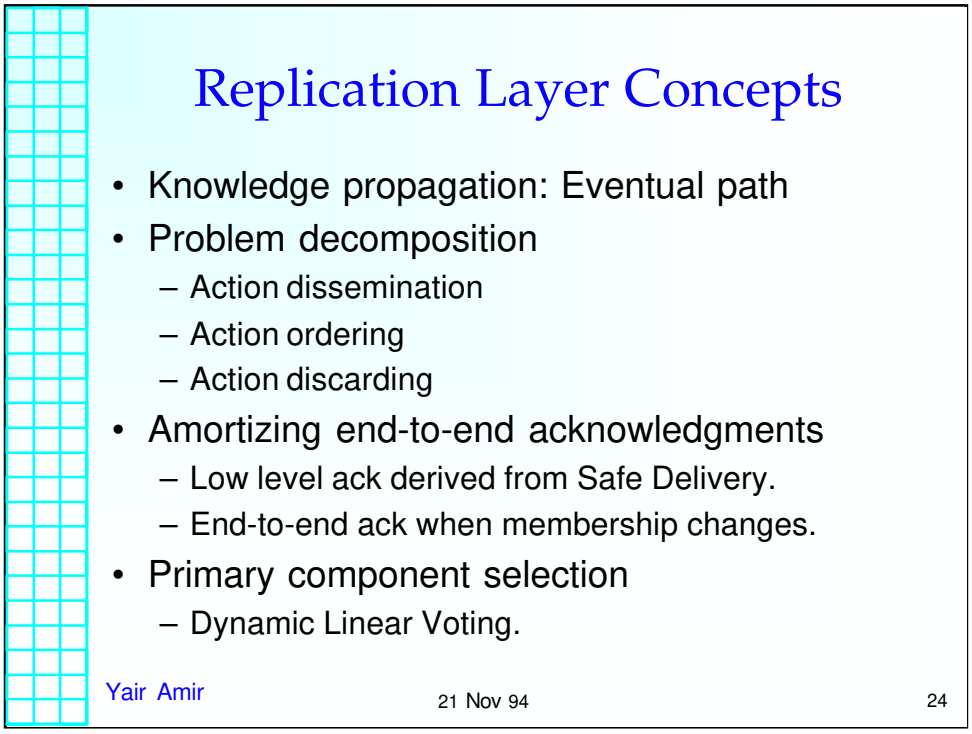
It is worth noting that:

• The Transitional configuration at *t* is equal to the former regular configuration. In the case of a merge, the Transitional configuration will always be empty because any message from the former regular configuration meets the safe criterion.

• The Transitional configuration in *s* and *r* is {*s*, *r*}.

• The relative order of messages that originated in a regular configuration is similar at all the processes that deliver them.

• Processes that belong to two consecutive regular configurations, deliver the same set of messages at the transitional configuration in between.



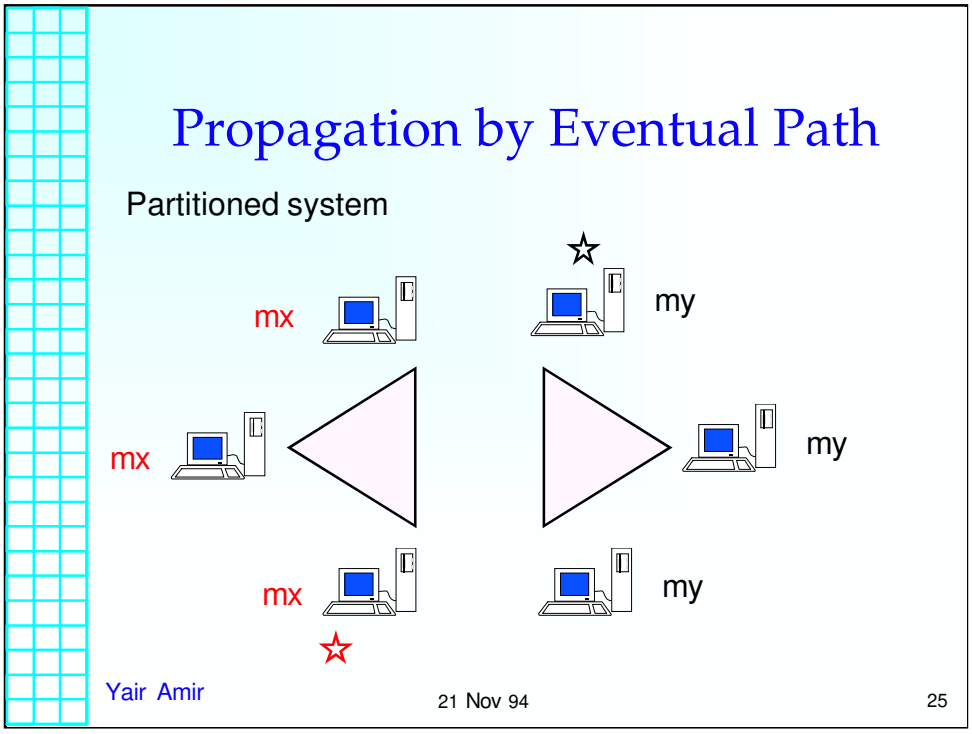
The replication layer is based on four main ideas:

1. We developed an efficient method, called "Propagation by Eventual Path", to propagate knowledge among the replication servers.

2. We decomposed the problem of maintaining consistent replicas to three disjoint problems: a) how to propagate the actions, b) how to consistently order the actions subject to the failure model, and c) how to know that all the servers already ordered and applied certain actions, so the messages that contain these actions can be discarded. Traditional replication protocols initiate, commit, and discard actions, only while in a primary component of the network. i.e. they gather all the acks in Two Phase Commit, or they gather a majority in Three Phase Commit, or they are connected with the primary site in a Primary-Backup approach. Otherwise updates can not be initiated and sometimes queries can not be consistently answered. In our method, actions are initiated anytime, and are ordered when they reach a primary component.

3. Our ordering algorithm guarantees consistency subject to the failure model. It avoids the need for end-to-end ack on a per action basis. Therefore, the latency for an update to be committed while in a primary component is equal to the latency for Safe delivery in this network! i.e. no need to wait for synchronous disk writes.

4. We use Dynamic Linear Voting to select the primary component of the network. DLV is considered to be the best method is this research area today.



In most systems, processes exchange information only as long as they have a direct and continuous connection. In contrast, we propagate knowledge by means of eventual path.

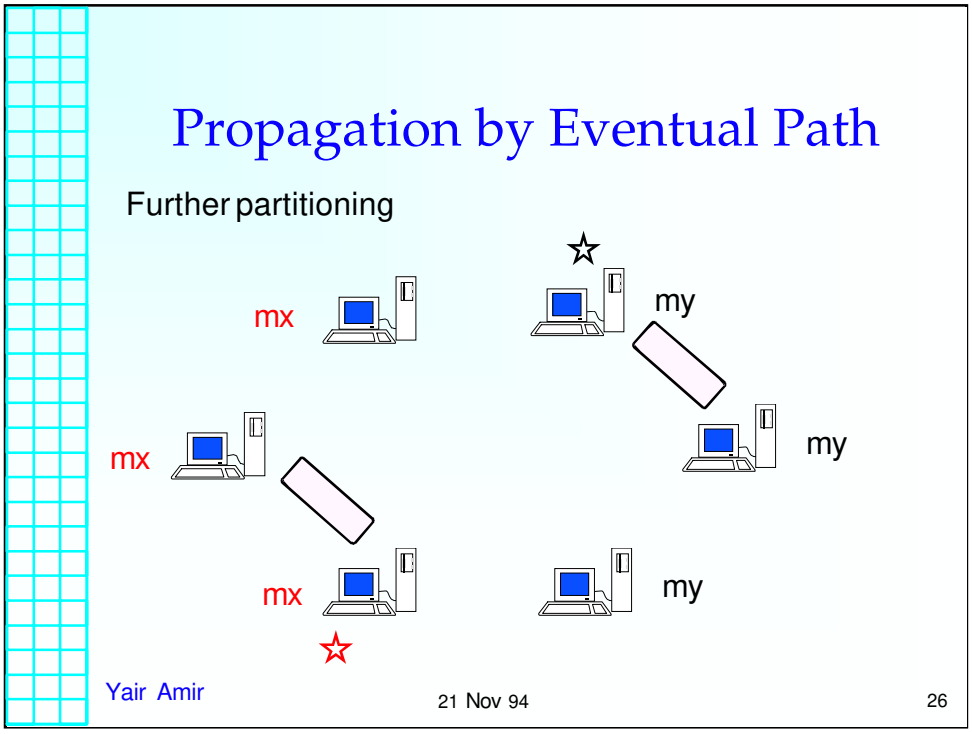
An *Eventual Path* from server *p* to server *q* up to and including action *a* is a communication path from *p* to *q* such that there exist pairs of servers along the path and intervals during which they are connected so the the message containing action *a*, and all prior messages received by *p* are eventually received by *q*.

Here we give a scenario to explain the eventual path. We show how using eventual path propagation affect the progress of the system.

In the example the network is partitioned to two components.

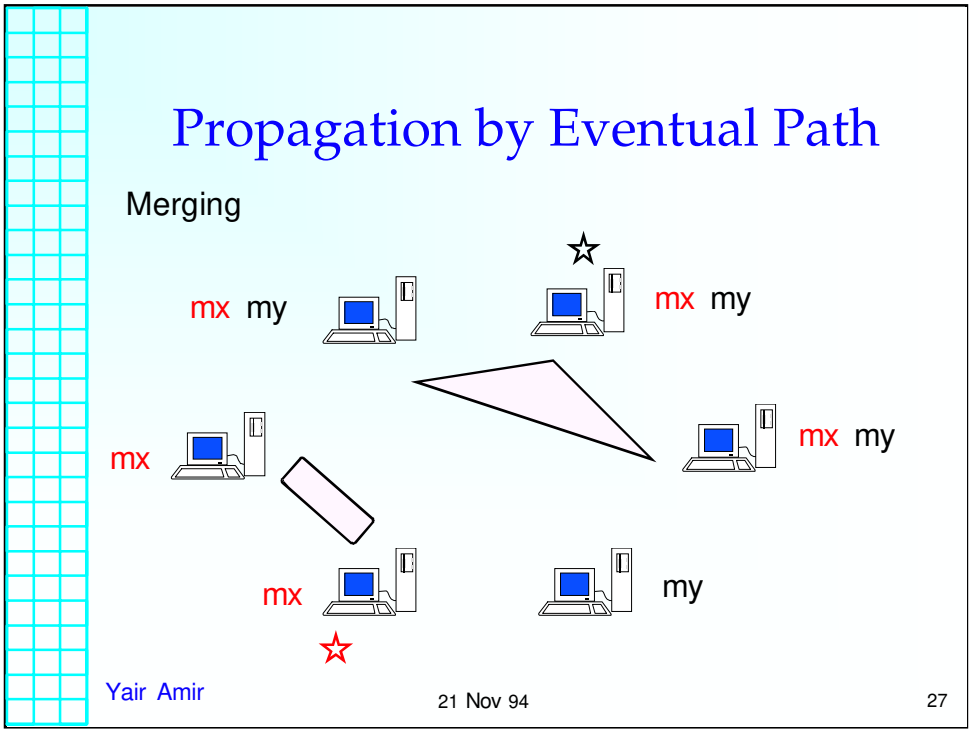
In each of the components, the stared processor initiates an action in a message (message *mx* in the left component and message *my* in the right component).

Each processor knows about the action initiated in its component and knows that the other processors in its component know the action. However, each processor can not know about the action which was initiated in the other component.

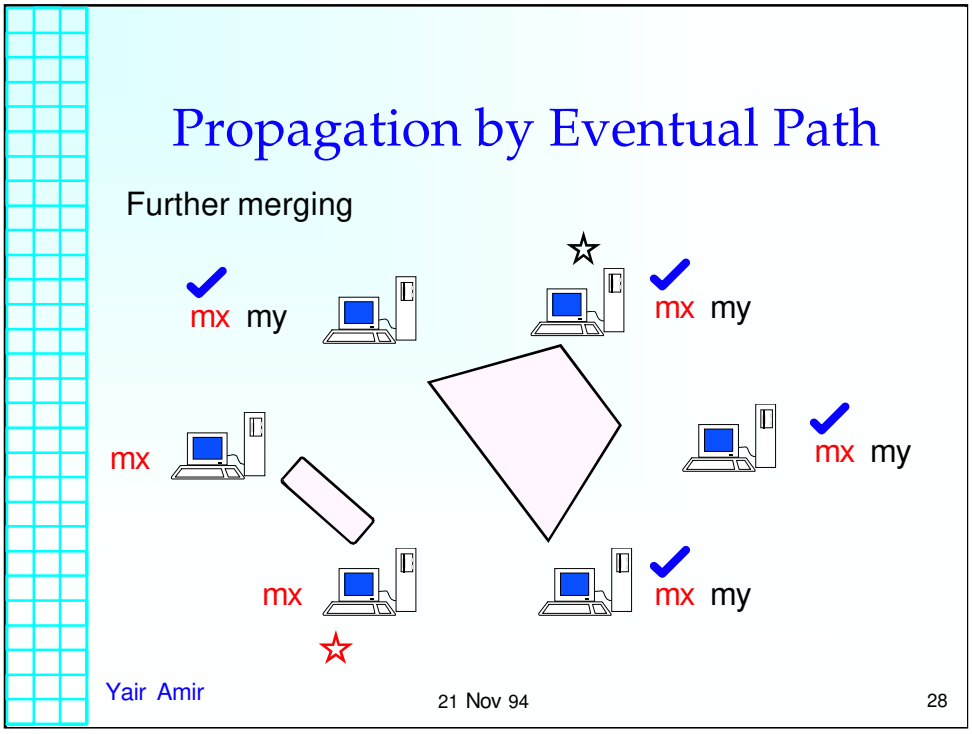


The network further partitioned. Now we have four components.

No new knowledge is gained by any of the processors.



Here, two components merge. The three processors in the upper component exchange messages so that they all have *mx* and *my*. Moreover, the fact that the two lower component processors know *mx* is also shared.



Finally, the bottom right processor merges with the upper component.

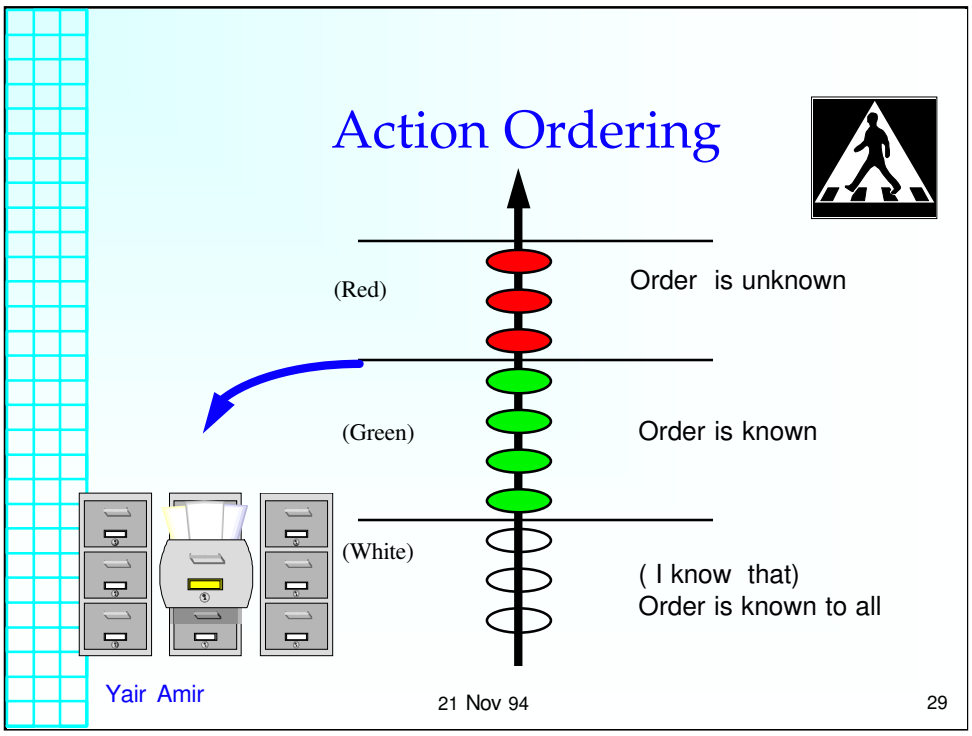
Again, after exchanging the necessary messages, all the processors belonging to the upper component have *mx* and *my*.

Notice that *mx* was initiated by a processor which was never connected to processors that have *mx*. Moreover, some of the processors even know that **all** the processes in the system have *mx*.

Propagation by means of eventual path is a generalization of gossip methods. Instead of sharing information pair-wise, we use the group communication mechanisms and do it group-wise.

Propagation by means of eventual path, although superior to gossip methods, is not popular in systems today. I believe that this is due to the expensive bookkeeping needed when point to point communication is used. Group communication simplifies this bookkeeping and makes it an appealing tool for knowledge dissemination and sharing.

We use the eventual path propagation to disseminate actions and to learn that actions were already ordered (and applied) by all processors, and therefore can be discarded.



The replication servers implement a symmetric distributed algorithm to determine the order of actions to be applied to the database. Each server builds its own knowledge about the order of actions in the system. Each server marks the actions delivered to it with one of the following colors:

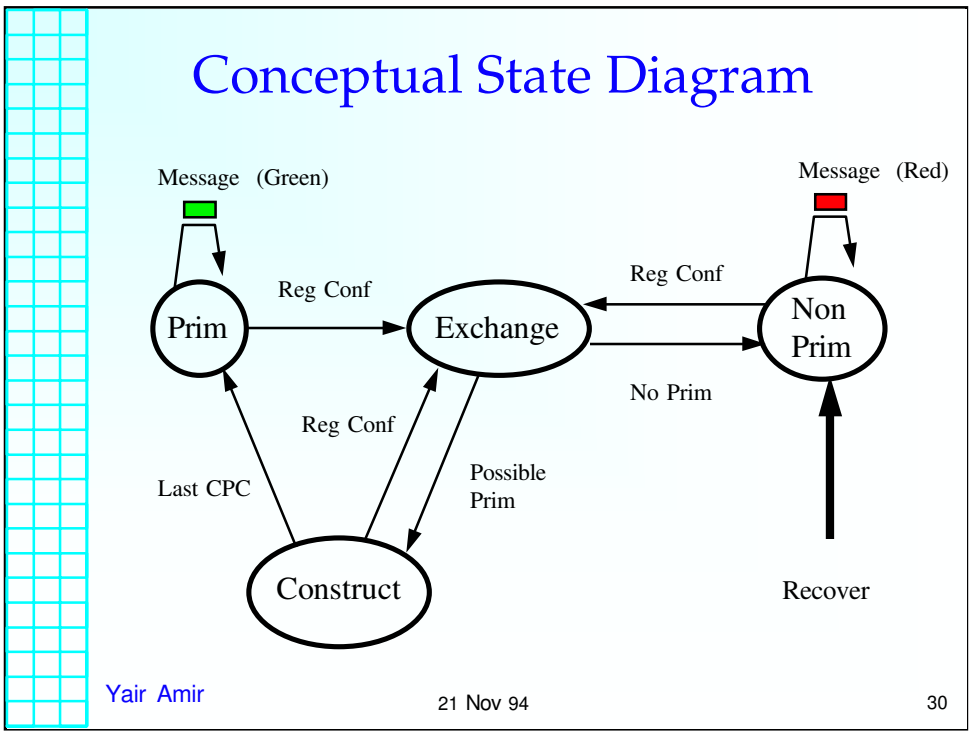
• **Red Action:** An action that has been ordered within the local component by the group communication layer but for which the server cannot, as yet, determine the global order.

• **Green Action:** An action for which the server has determined the global order and which, therefore, can be applied to the database.

• **White Action:** An action for which the server knows that **all** of the servers have already marked as green. Thus, the server can discard a white action because no other server will need that action subsequently.

The replication layer identifies at most a single component of the server group as a *primary component*; The other components of a partitioned group are *non-primary components.* We use a Dynamic Linear Voting technique to determine the primary component.

In the primary component, actions are marked green on delivery by the group communication and, hence, are immediately applied to the database. In a non-primary component, actions are marked red.



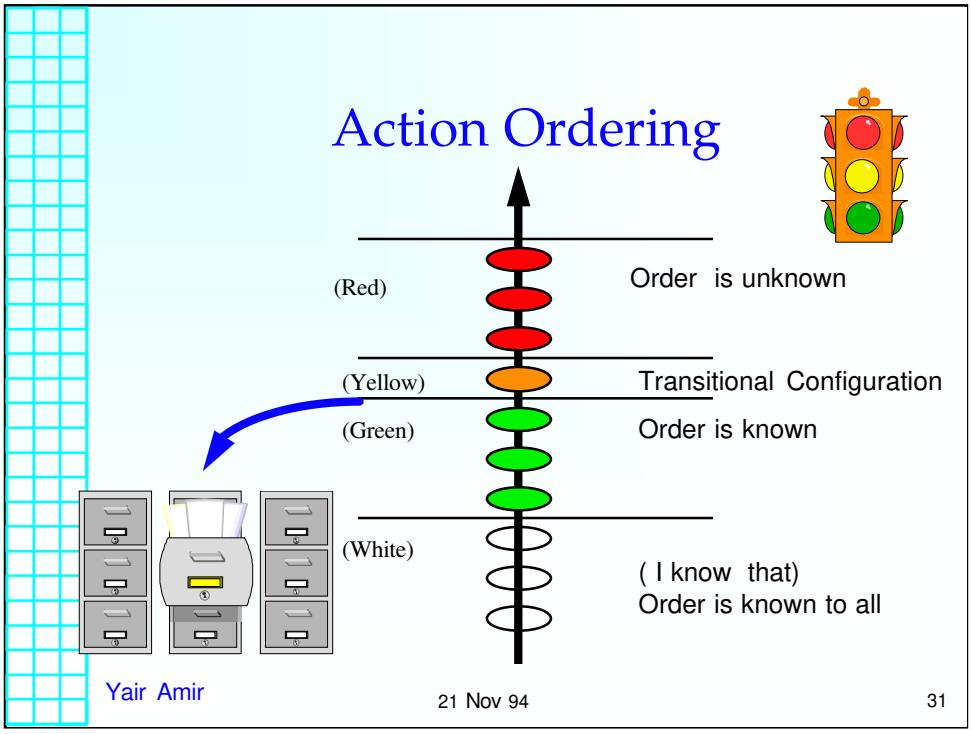
A simple state machine describes the concept of the replication server :

• **Prim** - The server currently belongs to the primary component. When a message containing an action is delivered by the group communication layer, the action is immediately marked green and is applied to the database.

• **Non Prim** - The server belongs to a non-primary component. When a message containing an action is delivered by the group communication layer, the action is marked red.

• **Exchange** - The server shifts to Exchange when a new (regular) configuration is formed. The servers belonging to the new configuration exchange information to define the set of actions that are known to some, but not to all, of them. After these actions have been exchanged and the green actions have been applied to the database, the servers checks if this configuration can become the primary component. If so, they shift to Construct; otherwise, they shift to Non Prim and form a non-primary component. This check is done locally at each server without the need for additional messages.

• **Construct** - In this state, all of the servers in the component have identical knowledge about the configurations. After writing the data to stable storage, each of the servers multicasts a *Create Primary Component* (CPC) message. On receiving a CPC message from each of the servers, the server shifts to the Prim state. If a configuration change occurs before all the CPC messages have been received, the server shifts to Exchange.



Due to the asynchronous nature of the system model, we cannot reach complete knowledge about which messages were received by which processes just before a network partition or a process crash occurs. Instead we rely on the semantics of extended virtual synchrony for safe delivery. The lack of complete knowledge is evident when:

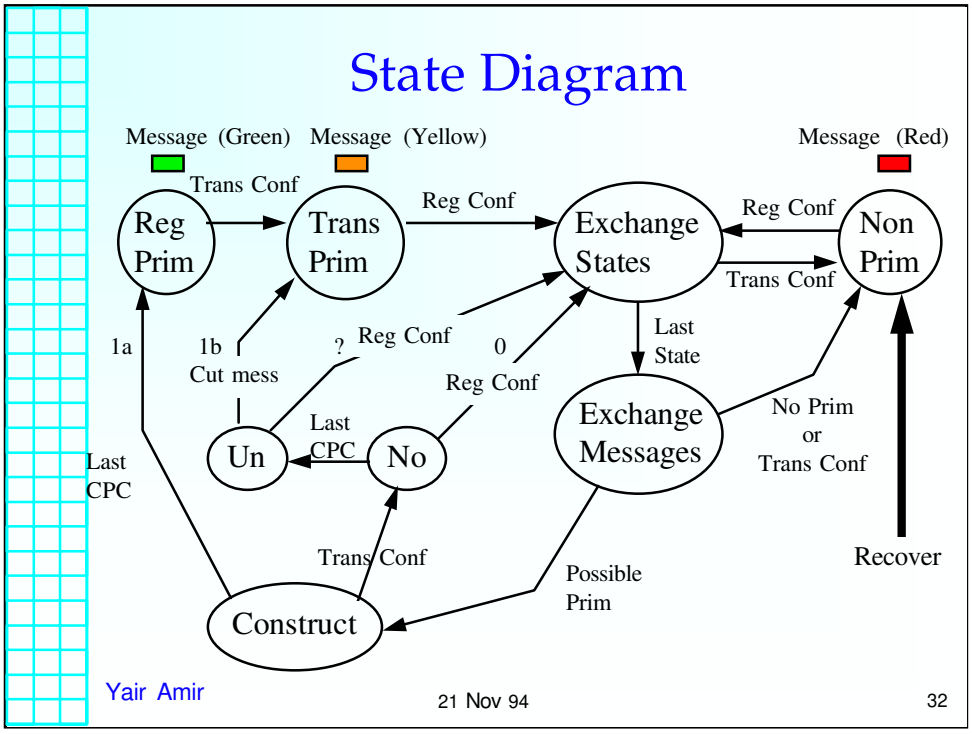
a) A server is in the Prim state and a partition occurs. In this case, the server cannot always tell whether the last messages were received by all the members of the primary component (including itself).

b) A server is in the Construct state and a partition occurs. In this case, the server cannot always tell whether all the servers initiated the CPC message, or whether some of them delivered all the CPC messages in the previous regular configuration, and therefore, installed a new primary component.

Extended virtual synchrony provides the notion of transitional configuration. to handle these cases. We add an intermediate color to our colors model:

• **Yellow Action:** An action that was received in a transitional configuration of a primary component.

This action could have been marked green by another server of the primary component, or, alternatively, could have been missed by another server, but both cases cannot simultaneously exist. Yellow actions cannot be applied to the database. However, They will be the first actions to be marked green when a new primary component is formed.



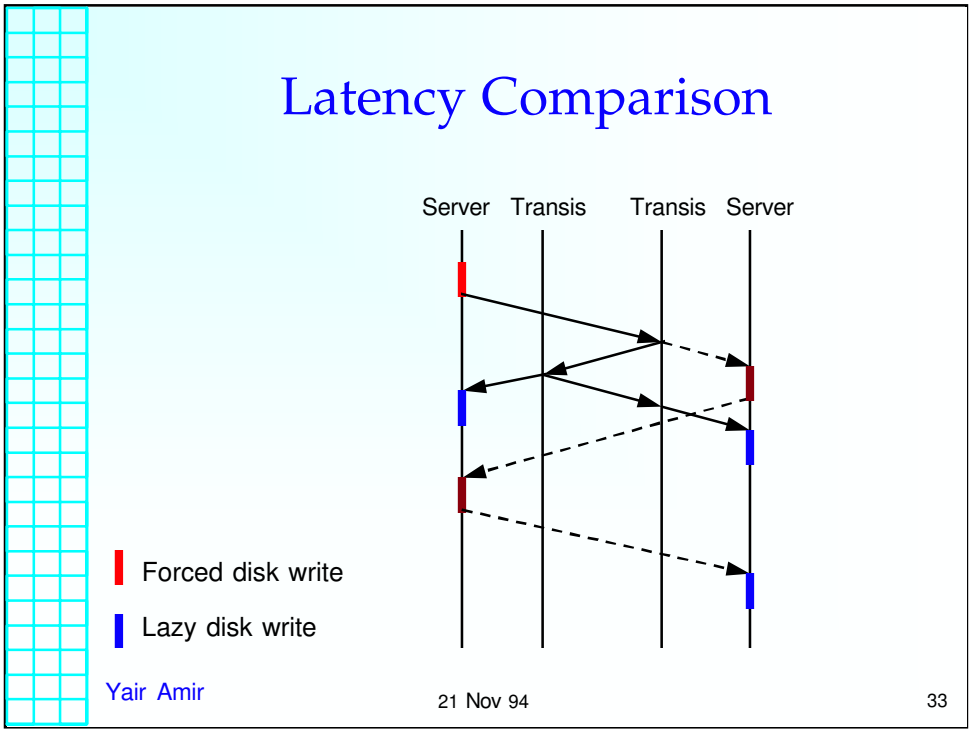
This slide presents the full state diagram that solves the consistency problem subject to our failure model. There are 3 refinements compared to the conceptual state diagram:

1. The Exchange state is broken into Exchange States and Exchange Messages. This is done for clarification reasons.

2. The Prim state is broken into Reg Prim and Trans Prim states. This is done to address the lack of knowledge explained in the previous slide regarding yellow actions. Extended virtual synchrony guarantees that if a safe message (containing an action) is delivered to the replication server in a primary component, this message will be delivered to all other members of that component, unless they crash. Some may receive it in a transitional configuration and mark it yellow, if a configuration change has just occurred.

3. The Construct state is broken into Construct, Un, and No states. It is guaranteed that if a server receives all the CPC messages and goes to Reg Prim or Trans Prim (case 1a or 1b), then no other server goes to Exchange States through the No state (case 0). This way we guarantee that if even one server installs the new primary component, all the others, that do not crash**,** receive all the CPC messages, and therefore, either install (if they go through case 1a or 1b) or protect this installation (if they go through case ?).

For a more detailed explanation and correctness of this slide, please refer to the paper "Robust and Efficient Replication Using Group Communication".

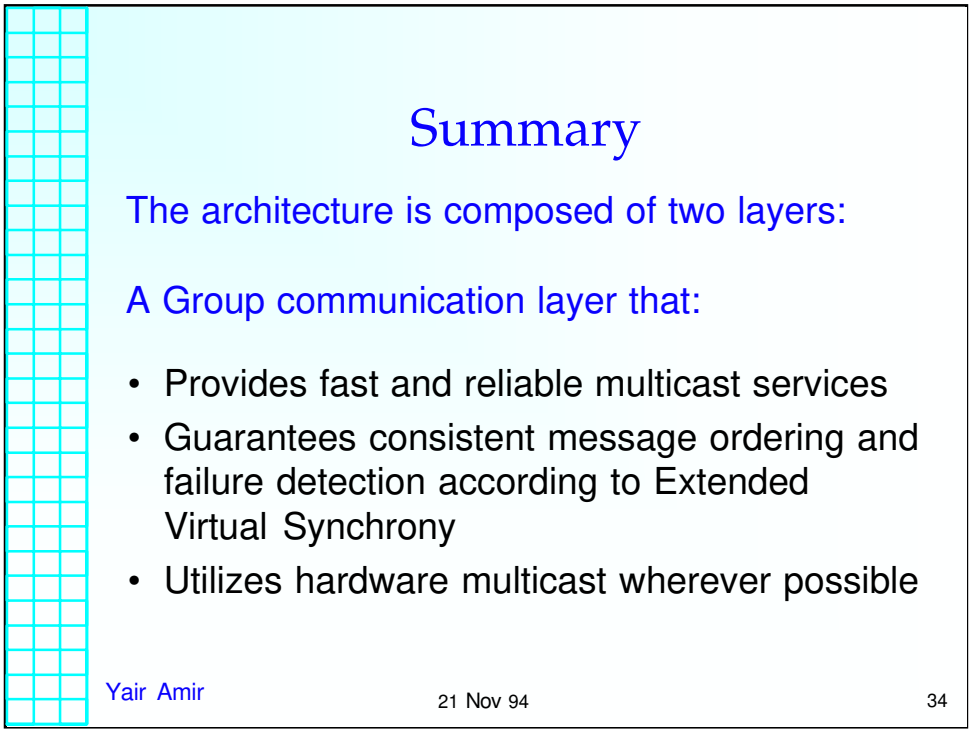


In a primary component the latency of actions is determined by the safe delivery latency of the group communication layer plus the processing time within the replication layer. In contrast, the latency for existing database systems is determined by end-to-end acknowledgment mechanisms such as two-phase commit, and by forced disk writes. The approach presented here can, therefore, provide lower latency than existing replication methods, while preserving strict consistency.

Furthermore, using our group communication techniques, performance can go up to several hundreds of actions per second. The fact that forced disk writes are not used on a per action basis allows the operating system to optimize disk operations. Hence, performance is not limited by the disk seek time as in other replication methods.

A question which is not considered in this talk, but is addressed in the paper, is what happens to clients that belong to a non-primary component. Traditional replication methods either block non-primary components or do not allow to update the database while in a non-primary component.

Our method allows clients to always initiate new updates. Furthermore, *dirty* and *weak* queries are answered immediately while in a non-primary component. *Consistent* queries can be answered only when the action containing this query is marked green. For exact definitions of these services and query semantics, please refer to the paper.



Further information about the group communication results described in this talk can be found in the Transis www and ftp sites (see first slide), and were published in:

• Y. Amir, D. Dolev, S. Kramer, D. Malki
"Transis: A Communication Sub-System for High Availability".
In the *proceedings of the 22nd Fault-Tolerant Computing Symposium*, Boston MA. (July 1992) 76-84

• Y. Amir, L. E. Moser, P.M. Melliar-Smith, D. A. Agarwal and P. Ciarfella
"Fast message ordering using a logical token-passing ring".
In the *proceedings of the 13th International Conference on Distributed Computing Systems,* Pittsburgh PA. (May 1993). 551-560. IEEE.

• L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal
"Extended Virtual Synchrony"
In the *proceedings of the 14th International Conference on Distributed Computing Systems,* Poznan Poland. (June 1994). 56-65. IEEE.

Summary (continue)

A Replication layer that:

- Automatically recovers from crashes
- Consistently orders Actions
- Optimally disseminates Actions
- Utilizes low level acknowledgments
- Needs end-to-end acknowledgment only upon membership change

Further information about the replication layer described in this talk can be found in the Transis www and ftp sites (see first slide), and were published in:

• O. Amir, Y. Amir and D. Dolev
"A Highly Available Application in the Transis Environment"
In the *Proceedings of the Hardware and Software Architectures for Fault Tolerant Workshop*, Le Mont Saint-Michel, France (June 1993). LNCS 774.

• 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, Israel.