Distributed Systems

- We will consider distributed systems to be collections of compute engines in a NORMA configuration. Information exchange among systems will be by message passing through some interconnecting Inter-Process Communication (IPC) mechanism
- Such configurations are motivated by:
  - resource sharing
  - enhanced performance
  - improved reliability and availability
  - modular expandability

Distributed Systems

- As with any collection of compute elements, there are many and varied ways of deploying resources at the physical level. In general the distinguishing characteristic at the system software level is the need to support the exchange of information among threads which live in different processes which exist on different physical systems
- This exchange, of course, not only requires communication channel support, but often requires synchronization

Distributed Systems

- If we assume that a distributed system must maintain any kind of global state information (a fundamental requirement of any coordinated system), then it is fair to ask how such information can be maintained in a coherent way
- We've already seen that this is the crux problem in a non-distributed system, where we've been able to control mutually exclusive access to shared resources by employing spin locks as a foundation for various synchronization primitives (event counters, semaphores, etc.)

Distributed Systems

- In non-distributed systems, however, our spin lock implementations depended on the memory interlock provided by UMA and ccNUMA (memory interlock was a minimum requirement for Peterson’s algorithm or the Bakery algorithm), and, for most contemporary systems, we could leverage machine instructions like the IA-32 XCHG which provided atomic 2 bus-cycle access
- Since distributed systems do not share a common bus, we do not have even memory interlock to help us synchronize access to shared components

Distributed Systems

- So we have two basic challenges in distributed systems engineering
  - What type of communication channels can we establish to support the inter-host IPC we need to exchange information ?
  - How can we synchronize the exchange of information to make sure that we can provide coherence semantics to shared global objects ?
- These primitive problems must be solved to provide a foundation for any type of distributed system which shares dynamically changing information structures among its subscribers

Distributed Systems

- One can envision networks of computing systems that share very little
  - UNIX systems using NFS software to share files
  - Windows systems using CIFS software to share files and printers
- Other systems (HPC, grid computing, big data) may share a lot
  - The need for distributed storage and distributed computation has been addressed by new open software stack elements like Hadoop for storage and Spark for distributed computation

Distributed Systems

- The main issues in distributed systems include:
  - Global knowledge
    - Which parts of a distributed system must be shared among the members in a consistent fashion ?
    - To what degree can system control be de-centralized ?
  - Naming
    - How are system resources identified and what name spaces can they be found in ?
    - Is resource replication a sensible de-centralization strategy ?
  - Scalability (scale-out in this context)
    - How easily can a distributed system grow ?
    - Will certain software architectures constrain growth ?

Distributed Systems

- The main issues in distributed systems (cont'd):
  - Compatibility
    - Binary level: an executable image can run on any member (JVM brings this to another level)
    - Execution level: source code compatibility, can compile/run
    - Protocol level: most interoperable in heterogeneous environments. System services accessed with common protocol framework (NFS, XDR, RPC)
  - Process synchronization
    - The mutual exclusion, critical section problem
  - Resource management
    - How are mutually exclusive system resources allocated ?
    - Mutually exclusive resources can lead to deadlock

Distributed Systems

- The main issues in distributed systems (cont'd):
  - Security
    - Authentication: who is on each end of an exchange ?
    - Authorization: do the endpoint threads have the privilege required for an exchange ?
  - Structure
    - Monolithic kernel: one size fits all member systems
    - Collective kernel: often microkernel based with a collection of daemon processes to provide specific resources in each member's environment as required by that member. In a distributed system the microkernel message passing system sits at the lowest level of the architecture (MACH)
    - Object oriented systems: resources are treated as objects which can be instantiated on member systems…. Mechanism / Policy

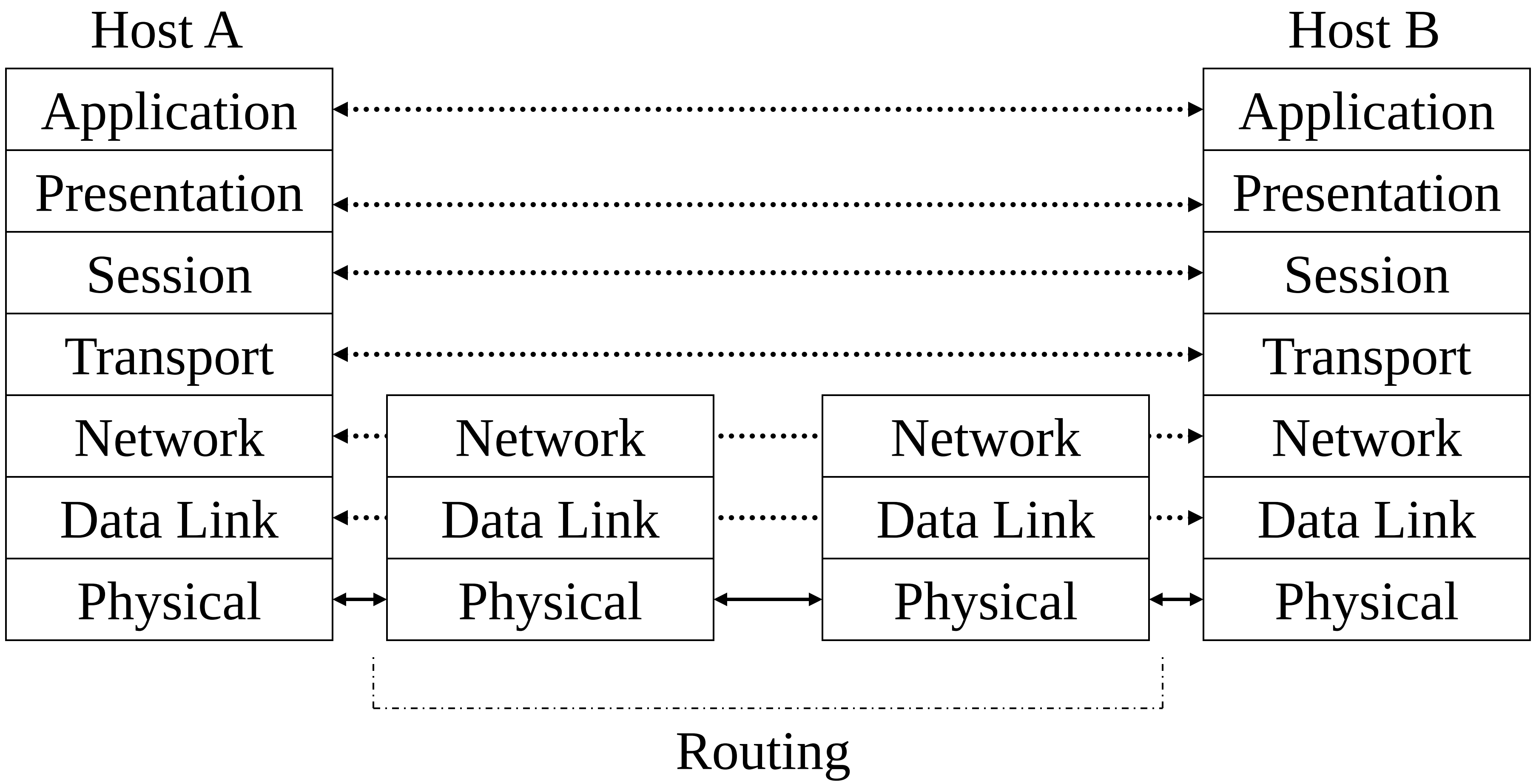
Distributed Systems

- Whatever the structure of a distributed system, the actual exchange of information from one member to another is typically based on the Client-Server computing model
  - A client requests access to information kept somewhere in the system
  - A server functionally provides the access on behalf of the requesting client
  - This embodies the Remote Procedure Call (RPC) paradigm

Distributed Systems

- Client-Server computing often takes place over a networked environment where an exchange can be decomposed into a set of layered abstractions as described in the Open System Interconnect model
  - Each layer on one side of the communication is viewed as communicating with its peer layer on the other side, and will have little or no knowledge of the layers above or beneath it
  - Exchanges, of course, occur by traveling down through the layers on the initiator’s side, across the physical network connect, and then up through the layers on the receiver’s side

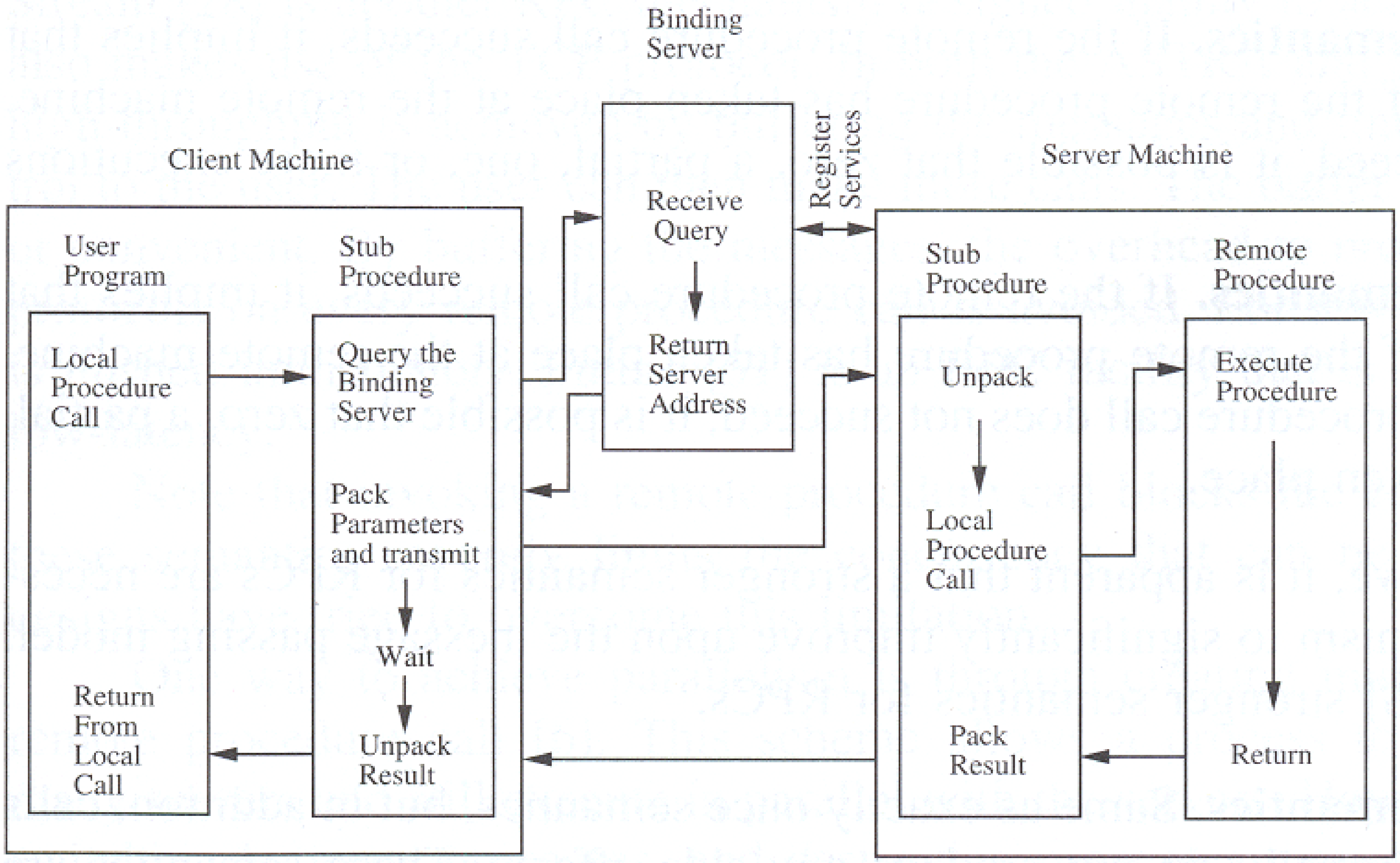
Peer Level Communication in a Routed Network



Some Examples of approximate OSI Layer Functionality

Application	NFS	FTP
Presentation	XDR	
Session	RPC	
Transport	UDP	TCP
Network	IP	
Data Link	802.3	
Physical	1000Base -T	

Remote Procedure Call Paradigm



Distributed Systems

- Design issues in RPC
  - Structure: client and server stubs (rpcgen)
  - Binding: naming and name space management (portmapper, rpcbind) … transparency
  - Parameter and result passing
    - Structured data and platform dependencies (endianness)
    - Architecturally neutral conversions (ASN.1 BER)
    - Receiver-makes-right conversions (OSF-DCE)
  - Error management and recovery
    - System failures
    - Communication failures

Distributed Systems

- Design issues in RPC (cont'd)
  - Communication semantics
    - At least once (for idempotent transactions)
    - Exactly once (non-idempotent transactions)
    - At most once (zero or one semantics)

Distributed Systems

- Inherent limitation in a distributed system
  - Absence of a global clock
    - Since each system has its own independent clock there can be no global time measure in the system
    - Systems may converge on a global time but there can never be precise enough agreement to totally order events
    - A lack of temporal ordering makes relations like “happened before” or “happened after” difficult to define
  - Absence of shared memory
    - A thread in a distributed system can get a ***coherent*** but partial view of the system, or a complete but ***incoherent*** view of the system
    - Information exchange is subject to arbitrary network delays

Distributed Systems

- If absolute temporal ordering is not possible, is there a way to achieve total ordering in a distributed system ?
  - Remember, solving problems like the multiple producer / multiple consumer require total ordering)
- Lamport's logical clocks provide such mechanism, depending on a reliable, ordered communication network (i.e. with a protocol like TCP)
- Logical clocks can implement "happened before" ($\rightarrow$) relationships between threads executing on different hosts in a distributed system

Distributed Systems

- Lamport's logical clocks can determine an order for events whose occurrence has been made known to other threads in the distributed system
- Messages may arrive with identical timestamps from different systems. Total ordering can be maintained by resolving ties arbitrarily but consistently on each system (i.e. using some unique and orderable attribute for each thread sending a message, such as a unique node-ID)

Distributed Systems

- Distributed mutual exclusion algorithms require total ordering rules to be applied for contending computations
  - If two producers want to produce an element into a ring buffer slot they must be ordered in some way to avoid the case where they each try to place their produced element into the same buffer slot
- These algorithms are implemented in two general classes
  - Non-token based
  - Token-based

Distributed Systems

- Requirements of distributed mutual exclusion algorithms
  - Freedom from deadlocks
  - Freedom from starvation (bounded waiting)
  - Fairness (usually FIFO execution)
  - Fault tolerance
- The performance of such algorithms is also very important.
  - Since distributed systems are message based and network latencies impact overall performance, limiting the number of messages necessary to achieve mutual exclusion is always an objective of any implementation

Distributed Systems

- Non-token-based implementations must exchange some number of messages to achieve total ordering of events at all sites
- As previously seen, Lamport’s logical clocks can solve the total ordering problem, provided certain conditions are met
  - A fully connected network
  - Reliable delivery of messages over the network
  - Ordered (pipelined) delivery of messages over the network

Distributed Systems

- Three nontoken-based algorithms are discussed here
  - Lamport's algorithm
    - Using request, reply and release messages
  - The Ricart and Agrawala algorithm
    - Using only request and reply messages
    - Message reduction means improved performance
  - Maekawa's algorithm
    - Further reduces messages
    - Provides lower bounds on messages exchanged for certain sized networks
    - Uses request, locked, release under light load, and may need to use failed, inquire and relinquish under heavy load

Distributed Systems

- Lamport's algorithm
  - Each node in the distributed system has a **node controller process** which is fully connected to each other peer node controller process
  - Node controllers each maintain a **Lamport Logical Clock** at their respective sites, along with one or more **request queues**
    - All messages that they exchange among one-another are always time stamped with the originator's Logical Clock value and node ID
  - When a client at a node wants to proceed with a mutually exclusive operation it requests permission to do so from its local node controller

Distributed Systems

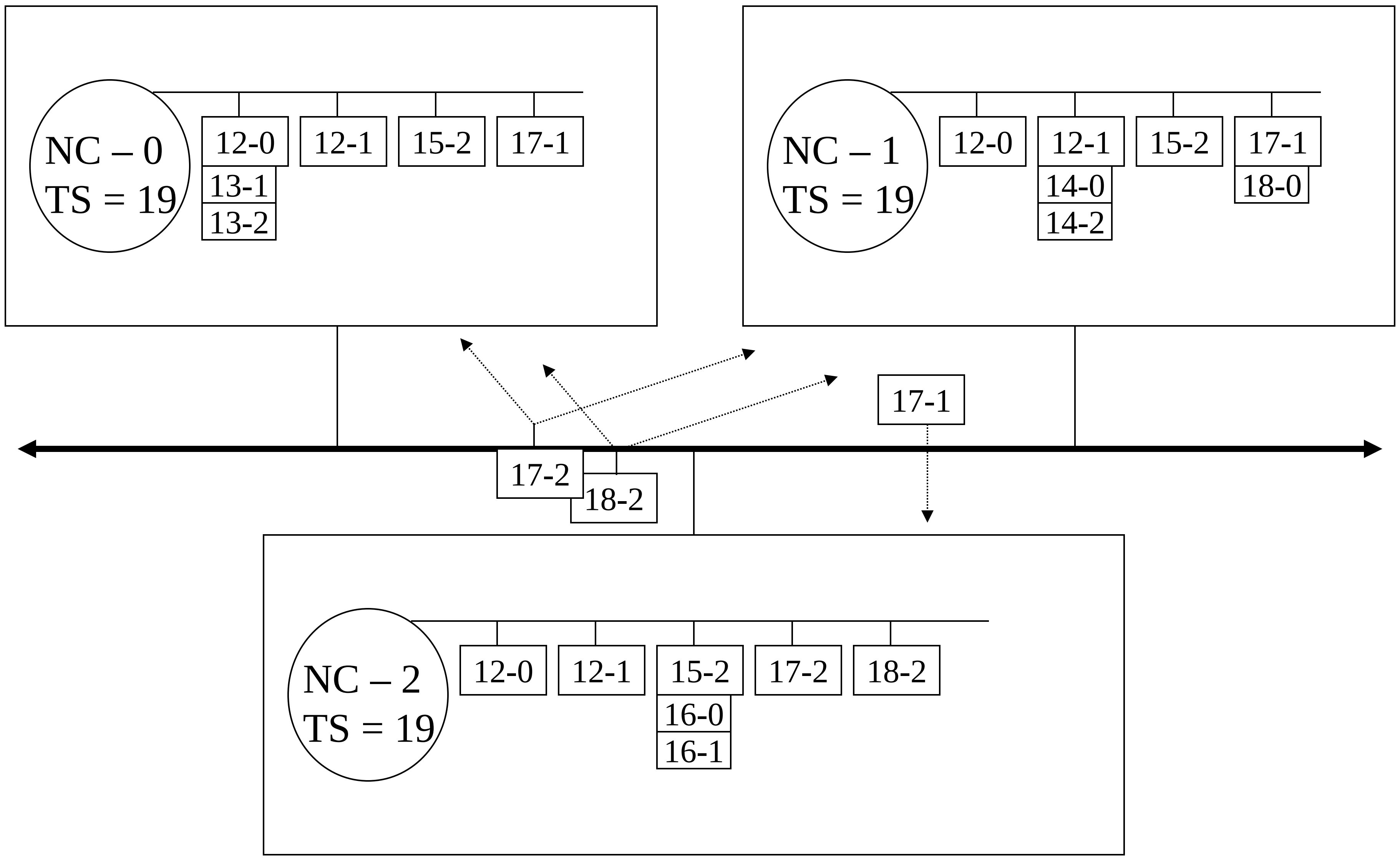
- Lamport's algorithm (cont'd)
  - A node controller sends a copy of a **time stamped REQUEST** message (stamped with a value 1 greater than any value previously used or seen) to all of its peers (N – 1 transmissions), and places its client request in its local queue of requests in **time stamp order**
  - When a node controller receives a **REQUEST** from a peer it
    - Places the request in its local queue of requests in time stamp order
    - Adjusts its Logical Clock to be 1 greater than any time stamp it has used or seen on any message previously sent or received
    - Sends a time stamped **REPLY** message to the originating node controller

Distributed Systems

- Lamport’s algorithm (cont’d)
  - When an originating node receives a **REPLY** from **each peer** (N – 1 transmissions) with a time stamp > the **REQUEST** message, and
  - When the client request has reached the **head of the originating node’s queue**, the node controller notifies the client that it may proceed
  - When the client has **completed** its mutually exclusive operation, it notifies its node controller that it’s done
  - The node controller now sends a copy of a **time stamped** **RELEASE** to each of its peers (N – 1 messages) to complete the protocol

Distributed Systems

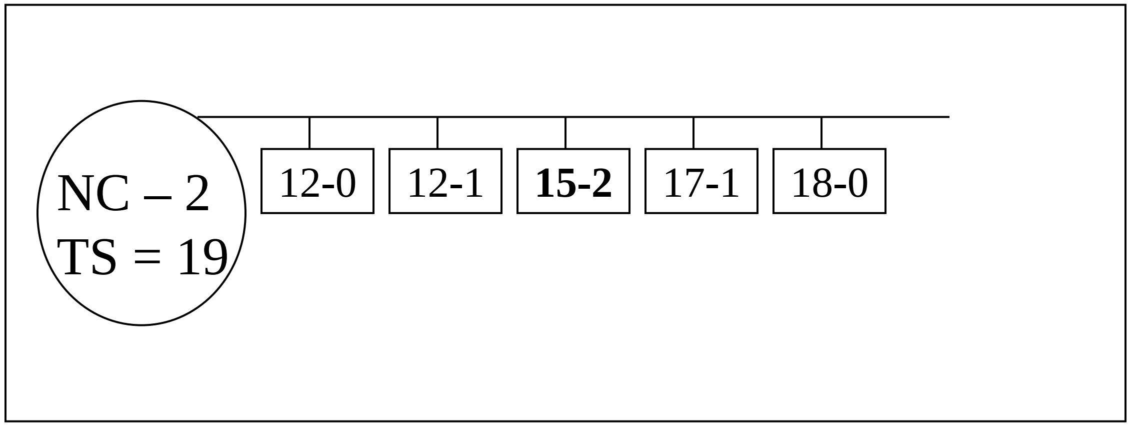
- Lamport's algorithm (cont'd)
  - When a node controller receives a **RELEASE** message it
    - Discards the request at the head of its queue (the **RELEASE** can only apply to that request), and checks to see which request will become the next head of the queue (if any)
  - Whichever node owns the next request to reach the head of the queue (all node queues should have an identical head element at any time) can then notify its client that it's safe to proceed, provided the request element has been made active by the receipt of a **REPLY** message from each peer
  - Message complexity is 3(N – 1) messages per CS


NC – 0
TS = 19
12-0
12-1
15-2
17-1
13-1
13-2
NC – 1
TS = 19
12-0
12-1
15-2
17-1
14-0
14-2
18-0
17-1
17-2
18-2
NC – 2
TS = 19
12-0
12-1
15-2
17-2
18-2
16-0
16-1

Distributed Systems

- Ricart and Agrawala algorithm
  - The same general structure as Lamport, but messages are reduced by deferring the **REPLY** message until all of a node's preceding requests have completed
  - A node controller can allow a client to proceed when a **REPLY** has been received from each peer controller with a Time Stamp > than the original **REQUEST** TS, **discarding** any elements on the queue that belong to a peer which has just sent a **REPLY**
  - **RELEASE** messages are no longer required, lowering the message complexity to 2(N – 1) messages per CS

Ricart and Agrawala algorithm



This node controller has sent a REPLY message to node 0 with a TS of 13-2 and a REPLY to node 1 with a TS of 14-2 and has sent REQUEST messages to everyone with **TS 15-2** for one of its clients, as you can see on this queue above. This node controller has also received additional REQUEST messages from node 1 and node 0, and, while these have been queued in TS order, this node controller will not send a REPLY to either until his **15-2** client completes his CS.

Distributed Systems

- Maekawa's algorithm
  - Maekawa was interested in exploring the bounds of optimality for a non-token based **Fully Distributed Mutual Exclusion** (FDME) algorithm. Fully Distributed implied equal **effort** and **responsibility** from all nodes in the network (Lamport's solution as well as Ricart and Agrawala's are non-optimal FDMEs)
  - He reasoned that a requesting node did not need permission from **every other node**, but only from **enough nodes** to ensure that no one else could have concurrently obtained permission. This is analogous to an election where a majority of votes is enough to assure victory, and a unanimous decision is not needed

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - But Maekawa realized that even a majority might be more than what’s needed in certain cases
  - Maekawa reasoned that if each node had to secure the permission of a **set of other nodes**, then finding the minimum size of such a set for a system of nodes could provide a lower bound on the number of messages required to execute a CS
  - He also reasoned that, since **Fully Distributed** implied equal work and equal responsibility, whatever set size was needed for a given network, all nodes would be assigned sets of precisely that size, and each node would have to participate in the same number of sets

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - So Maekawa established the following conditions:
    - For a network with N nodes, each node controller NC_i will be assigned a voting set of nodes Si such that:
      - For all i and j , $1 \leq i,j \leq N$, $S_i \cap S_j \neq \phi$, know as pair-wise non-null intersection
      - Equal effort requires that $|S_1| = \ldots = |S_N| = K$, where K is the number of elements in each set
      - Equal responsibility requires that every node controller NC_i , $1 \leq i \leq N$ will be in the same number D of S_j’s , $1 \leq j \leq N$
      - To minimize message transmission it is further required that an S_j , $1 \leq j \leq N$ always include NC_j as a member

Distributed Systems

- Maekawa's algorithm (cont'd)
  - From the preceding it should be clear that for **Full Distribution** K = D must always hold, and for optimality a network must contain exactly N nodes such that if I am one of the nodes:
    - N = (#of sets I participate in: D=K)

      * (set size after I remove myself from such sets: K – 1)

      + (me)
  - which transforms to **$N = K(K - 1) + 1$** since we will always have K = D
  - The algorithm is called Maekawa's square root algorithm since **$K \approx \sqrt{N}$**

Distributed Systems

- Maekawa's algorithm (cont'd)
  - For example, if voting sets include 3 members (**K = 3**) then a corresponding network must contain N nodes such that N = K(K – 1) + 1 = **3(3 – 1) + 1 = 7 nodes**
  - Clearly not all N can accommodate an optimal FDME solution
  - In fact even when N can be shown to be = K(K – 1) + 1 an optimal FDME solution may still be unavailable
  - A solution depends on finding N sets with the properties previously discussed, and this is equivalent to finding a ***finite projective plane*** of N points (also known as an FPP of ***order k*** where ***k*** = **K** – 1) for the example above this is an FPP of order **2**

Distributed Systems

- Maekawa's algorithm (cont'd)
  - An FPP of ***order k*** is known to exist if ***k*** can be shown to be an **integral power of a <u>prime number</u>** (Albert and Sandler '68) $\boldsymbol{k = p^{i}}$
  - In our example K = 3 so ***k*** = 2 and 2 = 2^1 so an FPP of order 2 with 7 points (sets) exists:
    - {1,2,**<u>3</u>**} S_1 {1,4,5} S_4 {1,6,7} S_6
      {2,4,6} S_2 {2,5,7} S_5 {**<u>3</u>**,4,7} S_7
      {**<u>3</u>**,5,6} S_3
  - Notice that all 4 conditions are met, pair-wise non-null intersection, equal effort, equal responsibility and **each node is a member of its own set**

Distributed Systems

- Maekawa's algorithm (cont'd)
  - What about N values which **can** be expressed as K(K – 1) + 1 , but whose ***k*** value **cannot** be shown to be an integral power of a prime number ? ($\boldsymbol{k} \neq p^i$)
  - A theorem by Bruck and Ryser ('49) states that an FPP of order ***k*** **<u>cannot</u>** exist if:
    - $(\boldsymbol{k} - 1)$ or $(\boldsymbol{k} - 2)$ is <u>divisible by 4</u>

      <u>AND</u>
    - $\boldsymbol{k} \neq a^2 + b^2$ (the sum of two integral squares)
  - So, for example, a system of N = 43 nodes where K = 7 and ***k*** = 6 satisfies the theorem since **(6 – 2) is divisible by 4** <u>AND</u> **6 <u>cannot</u> be shown to be the sum of 2 perfect integral squares** (i.e. no FPP exists for K=7)

Distributed Systems

- Maekawa's algorithm (cont'd)
  - But what about a system of N = 111 nodes where K = 11 and ***k*** = 10 ?
  - Since ***k*** cannot be shown to be an integral power of a prime we **cannot** apply the Albert and Sandler Theorem so there is no guarantee of an FPP and
  - Although we see that (10 – 2) is divisible by 4, we now also see that $10 = 1^2 + 3^2$, and since ***k*** can, in this case, be expressed as the sum of two integral squares we **cannot** apply the Bruck and Ryser Theorem, leaving us with no conclusion … this is an open question

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - For the particular system of K =11, ***k*** =10, with 111 (11*10+1) nodes there is additional information, although not in the form of a specific theorem
  - It has been shown by **exhaustive search** that **no FPP exists of order k=10**
  - **Clement Lam** developed the proof with distributed help from colleagues around the world for many years
  - The “proof” took several **years** of computer search (the equivalent of 2000 hours on a Cray-1). It is still known to be one of the most time-intensive computer assisted proofs ever reported. The final steps were ready in January 1989. See the URL at:

    http://www.maa.org/mathland/mathtrek_10_04_04.html

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - If we can find an FPP for a given set size K (of order ***k***) then we can implement Maekawa’s algorithm for the N nodes of such a network **(N = K (K – 1) + 1)**
  - The algorithm consists of 3 basic messages
    - **REQUEST**
    - **LOCKED**
    - **RELEASE**
  - And 3 additional messages to handle circular wait (deadlock) problems
    - **INQUIRE**
    - **RELINQUISH**
    - **FAILED**

Distributed Systems

- Maekawa's algorithm (cont'd)
  - When a node has a client which wants to execute its CS, the node controller sends each member of its voting set a time stamped **REQUEST** message
  - When a **REQUEST** message arrives at a peer node controller, that controller will return a **LOCKED** message to the requesting controller **provided that it has not already locked for another requestor** and that the timestamp on this request precedes any other requests it may have in hand
  - When a requesting node controller receives a **LOCKED** message from each peer in its set (K – 1 messages) and can **LOCK** its own node, <u>**the client can do its CS**</u>

Distributed Systems

- Maekawa's algorithm (cont'd)
  - When the client completes its CS, its node controller sends all peers in its set a **RELEASE** message, which allows any of these node controllers to then give their **LOCKED** message to some other requestor
  - When there is little or no contention for CS execution in the network, these 3 messages are generally all that is needed per CS, for a message cost of $\approx \mathbf{3 \sqrt{N}}$
  - This algorithm is prone to **<u>circular wait</u>** problems under heavy load, however, and additional messages are requires for a **deadlock free implementation**

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - If a **REQUEST** message arrives at a node controller which has already sent a **LOCKED** message to someone else, the controller checks the time stamp on the request, and sends the requesting node controller a **FAILED** message if the time stamp is newer (larger) than the **LOCKED** request or any other time stamped request in the node controller’s queue of waiting requests.
  - The node controller also places this request in **time stamped order** in its queue of waiting requests so that sometime in the future it can send a **LOCKED** message to this requesting node controller

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - On the other hand, if a **REQUEST** message arrives at a node controller which has already sent a **LOCKED** message to someone else, and the arriving request has an older (smaller) time stamp than the **LOCKED** request (and no previous **INQUIRE** message is currently outstanding), then an **INQUIRE** message is sent to the locked node controller.
  - The **INQUIRE** message attempts to recover the **LOCKED** message, and will succeed in doing so if the node the **INQUIRE** was sent to has received <u>**any**</u> **FAILED** messages from other members of its set. Such a node will send a **RELINQUISH** message in response to the inquiry, to free the **LOCKED** message

Distributed Systems

- Maekawa's algorithm (cont'd)
  - The inquiring node controller can now return the **LOCKED** message in response to the the older (smaller) **REQUEST** message, while it puts the **relinquishing node on its waiting queue in time stamp order**
  - When a **RELEASE** message shows up to a controller after one of his client's has finished a CS
    - A node controller will update its waiting queue and send its **LOCKED** message to the oldest (smallest) requestor in the queue if there are any left
  - Under heavy load then, it may take as many as **$\approx 5 \sqrt{N}$** messages per CS execution in the form of a **REQUEST**, **INQUIRE**, **RELINQUISH**, **LOCKED** and **RELEASE** sequence

Distributed Systems

- Maekawa's algorithm (cont'd)
  - Maekawa's algorithm can only provide an optimal FDME solution for **certain N** (N = K (K – 1) + 1), as we've seen
  - For networks with node counts which **cannot** be expressed by K (K – 1) + 1 for any **integral K**, Maekawa suggests a near-optimal, though no longer Fully Distributed solution, obtained by finding the N sets for the nearest **N > the required node count**, and then eliminating unused sets and editing the remaining sets to only include surviving nodes

Distributed Systems

- Maekawa’s algorithm (cont’d)
  - For example, consider the need to support a **10** node network. The nearest N = K (K – 1) + 1 is for a **K = 4 with that N = 13**. So we **<u>drop 3</u>** of the sets, and **edit the remaining sets** so that any occurrences of node controllers 11, 12 or 13 are systematically replaced by in range nodes
  - Notice that the mappings for 11, 12 and 13 must be **unique and consistent** in all remaining sets, but they can be arbitrarily selected from among the surviving nodes **(e.g. 11 ➔ 2, 12 ➔ 5, and 13 ➔ 3** would be valid mappings)

Distributed Systems

- Maekawa's algorithm (cont'd)
  - Consider the 13 node system:

{1,2,3,4}	**S1**	{1,5,6,7}	**S5**	{1,8,9,10}	**S8**	{1,11,12,13}	**S11**
{2,5,8,11}	**S2**	{2,6,9,12}	**S6**	{2,7,10,13}	**S7**		
{3,5,10,12}	**S10**	{3,6,8,13}	**S3**	{3,7,9,11}	**S9**		
{4,5,9,13}	**S13**	{4,6,10,11}	**S4**	{4,7,8,12}	**S12**		

  - Now delete and remap: **11 ➔ 2, 12 ➔ 5, and 13 ➔ 3**

{1,2,3,4}	**S1**	{1,5,6,7}	**S5**	{1,8,9,10}	**S8**
{***2***,5,8}	**S2**	{2,6,9,***5***}	**S6**	{2,7,10,***3***}	**S7**
{3,***5***,10,}	**S10**	{***3***,6,8,}	**S3**	{3,7,9,***2***}	**S9**
{4,6,10,***2***}	**S4**				

Distributed Systems

- Maekawa's algorithm (cont'd)
  - Notice that the system of 10 sets formed this way maintains the **pair-wise non-null intersection** requirement necessary to ensure mutual exclusion, **but the equal effort and equal responsibility requirements** of a fully distributed algorithm are no longer met, since the sets **are not all the same size**, and **some members are in more sets than others**
  - Nevertheless, the number of messages required is still bounded by the K value of the nearest N (here 4), and, for large networks, this provides the best solution available in terms of message complexity

Distributed Systems

- Token based algorithms
  - A **unique token** is shared by all members of a distributed system
  - Algorithms differ principally in the way that the search is carried out for the token
  - Token based algorithms use **sequence numbers** instead of time stamps
  - Every request for the token contains a sequence number, and the **sequence numbers advance independently at each site**
  - A site advances its sequence number each time it makes a request for the token

Distributed Systems

- Token based algorithms (cont’d)
  - Enforcing mutual exclusion with tokens is trivial since **only the token holder can execute CS code**
  - The central issues in such algorithms are:
    - Freedom from deadlock
    - Freedom from starvation
    - Performance and message complexity
  - Three algorithms are discussed here
    - The Suzuki-Kasami Broadcast Algorithm
    - Singhal’s Heuristic Algorithm
    - Raymond’s Tree-Based Algorithm

Distributed Systems

- The **Suzuki-Kasami** Broadcast Algorithm
  - A node controller sends a **sequenced REQUEST** message to **every other site** when it has a client which wants to do CS code
  - When the site which currently holds the token receives a REQUEST message it **forwards the token to the requesting node controller** if it is not doing CS code itself. A holding site is allowed to do its CS code as many times as it wants as long as it has the token. The main issues here are:
    - Identifying outdated REQUESTS from current REQUESTS
    - Understanding which site has an outstanding REQUEST

Distributed Systems

- The Suzuki-Kasami Broadcast Algorithm (cont'd)
  - A REQUEST from node controller j has the form **(j, n)**, where n is the sequence number
  - Each site keeps an array **RN_j of N elements** (where N is the number of sites), and **RN_j [i]** contains the largest sequence number seen to date by node controller j from node controller i
  - A **REQUEST** message (i, n) received at site j is **outdated if RN_j [i] $\geq$ n** and can be discarded (this may seem to imply that messages can be delivered out of order, but this is not the case)
  - The token itself also contains an array with one element per site called LN, such that **LN[i]** will always contain the sequence number of the last CS execution at site i

Distributed Systems

- The Suzuki-Kasami Broadcast Algorithm (cont'd)
  - The token also maintains a **queue of outstanding requestors** for the token
  - After the execution of CS code at a site j, the **LN[j]** element is updated with j's current sequence number, and j can then **compare** the rest of the LN array with its RN array
    - If j finds any RN[i] element that has a sequence number greater than the corresponding LN[i] element it **adds** site i to the token's queue of outstanding requestors; if the opposite is true it **updates its RN[i]** value from the token (discard info)
    - When j **completes the update** it removes the element from the head of the token's queue and sends the token to that site, or just holds the token if the queue is empty at that time (parking)

Token Arrives at Site #4

RN_4

Node	Seq #
1	7
2	13
3	4
4	3
5	15
6	21
7	13
8	6
9	9
10	17
11	2

LN

Node	Seq #
1	7
2	**12**
3	4
4	**2**
5	15
6	**20**
7	13
8	**7**
9	9
10	17
11	2

QH ➔2

Token Leaves Site #4 to Site #2

RN_4

Node	Seq #
1	7
2	13
3	4
4	3
5	15
6	21
7	13
8	**7**
9	9
10	17
11	2

LN

Node	Seq #
1	7
2	**12**
3	4
4	**3**
5	15
6	**20**
7	13
8	**7**
9	9
10	17
11	2

QH ➔2➔6

Distributed Systems

- The Suzuki-Kasami Broadcast Algorithm (cont'd)
  - The algorithm requires **0 or N** messages to be sent per CS execution, so worst case message complexity is **N** messages, less than both Lamport and Ricart & Argawala, but **not less** than Maekawa as N gets large
  - Since the token's queue is updated after each CS, requesting sites are queued in **approximated FIFO** order, and the algorithm avoids starvation (note that true FIFO is not achievable due to a lack of sortable timestamp, and the approximated FIFO is **less precise here than with the non-token based algorithms** previously discussed that employ a logical clock)

Distributed Systems

- Singhal's Heuristic Algorithm
  - In an effort to reduce the message complexity of algorithms like the Suzuki-Kasami, Singhal devised a token based implementation which requires a requestor to send a sequenced numbered REQUEST, on average, to **just half** of the N node controllers in the system
  - Of course the algorithm must assure that the token will be found among the node controllers receiving the REQUESTs, or, at the very least, that one of them will take possession of the token soon
  - The implementation requires **two N element arrays** to be kept at each site, and a token also with two arrays

Distributed Systems

- Singhal's Heuristic Algorithm (cont'd)
  - Each node controller at a site j keeps an N element array which maintains a **state value** for each site called **Sv_j[i]**, where $1 \leq i \leq N$. The element entries can have one of four state values:
    - $\mathcal{H}$ - the corresponding node is holding an idle token
    - $\mathcal{R}$ - the corresponding node is requesting the token
    - $\mathcal{E}$ - the corresponding node is executing CS code
    - $\mathcal{N}$ - none of the above
  - Each node controller at a site j keeps an N element array which has the most recent **sequence number** known for each site called **Sn_j[i]**, where $1 \leq i \leq N$

Distributed Systems

- Singhal's Heuristic Algorithm (cont'd)
  - The token has similar arrays, **TSv[i]**, and **TSn[i]**, where $1 \leq i \leq N$, and the elements of the TSv array maintain the most current state known by the token for each node, and those of the TSn array maintain the **most current sequence numbers known by the token** for each node
  - Whenever the site holding the token completes CS code, it **mutually updates** its and the token's arrays to reflect the most current available information, and then the site looks for an entry in its Sv array with state $\mathcal{R}$ (if such an entry exists) and sends the token to that site

Distributed Systems

- Singhal's Heuristic Algorithm (cont'd)
  - Algorithm steps:
    - Initialization: for each site j where $1 \leq j \leq N$, the site's Sv array elements are set such that **$Sv_j[i] = \mathcal{N}$** for i values from N to i and, **$Sv_j[i] = \mathcal{R}$** for i values from (i – 1) to 1, so for a 10 node network, node 5 will have an Sv array which looks like:

$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

    - The Sn array elements are all set to 0 at each site, since no sequence numbers have been seen at this point
    - Site 1 takes initial possession of the token by setting his own **$Sv_1[1] = \mathcal{H}$**, while all TSv elements = $\mathcal{N}$, and TSn's = 0

Distributed Systems

- Singhal's Heuristic Algorithm (cont'd)
  1. A requesting site i which does not already have the token, sets its own **$Sv_i[i] = \mathcal{R}$**, and increments its own **$Sn_i[i]$** sequence number by 1. It then checks its Sv array and, using its newly incremented sequence number, sends a sequence numbered REQUEST to **each element** in the array with **state $\mathcal{R}$**
  2. A site j which receives a request (i, sn_i) checks its **$Sn_j[i]$** element to see if this REQUEST is outdated. If it is outdated, then it is **discarded**, otherwise the receiving site **updates** its Sn[i] to this larger sn, and determines how to handle this REQUEST as follows:

Distributed Systems

- Singhal’s Heuristic Algorithm (cont’d)
  - If your own Sv entry is $\mathcal{N}$, then the Sv entry for the arriving REQUEST is set to $\mathcal{R}$ (it already could be $\mathcal{R}$)
  - If your own Sv entry is $\mathcal{R}$, and the Sv entry for the arriving REQUEST is not already $\mathcal{R}$, set your Sv entry for the arriving REQUEST to $\mathcal{R}$, and send a sequence numbered REQUEST of your own back to the requesting node (to tell him that you’re requesting) … otherwise, (after updating the Sv entry for the arriving REQUEST) do nothing (if your Sv entry was not $\mathcal{R}$)
  - If your own Sv entry is $\mathcal{E}$, then the Sv entry for the arriving REQUEST is set to $\mathcal{R}$ (it already could be $\mathcal{R}$)

Distributed Systems

- Singhal's Heuristic Algorithm (cont'd)
  - If your own Sv entry is $\mathcal{H}$, then the Sv entry for the arriving REQUEST from node i is set to $\mathcal{R}$, the token's TSv[i] element is set to $\mathcal{R}$, while your own Sv entry is changed to $\mathcal{N}$, and the token is sent to site i

3. When site i gets the token it sets its own Sv entry to $\mathcal{E}$ and does its CS code
4. When site i finishes its CS it sets its own Sv entry to $\mathcal{N}$, updates the token's TSv[i] entry to $\mathcal{N}$, and token's TSn[i] entry to its own Sn[i] value, and then begins the mutual update of its and the token's arrays as:

Distributed Systems

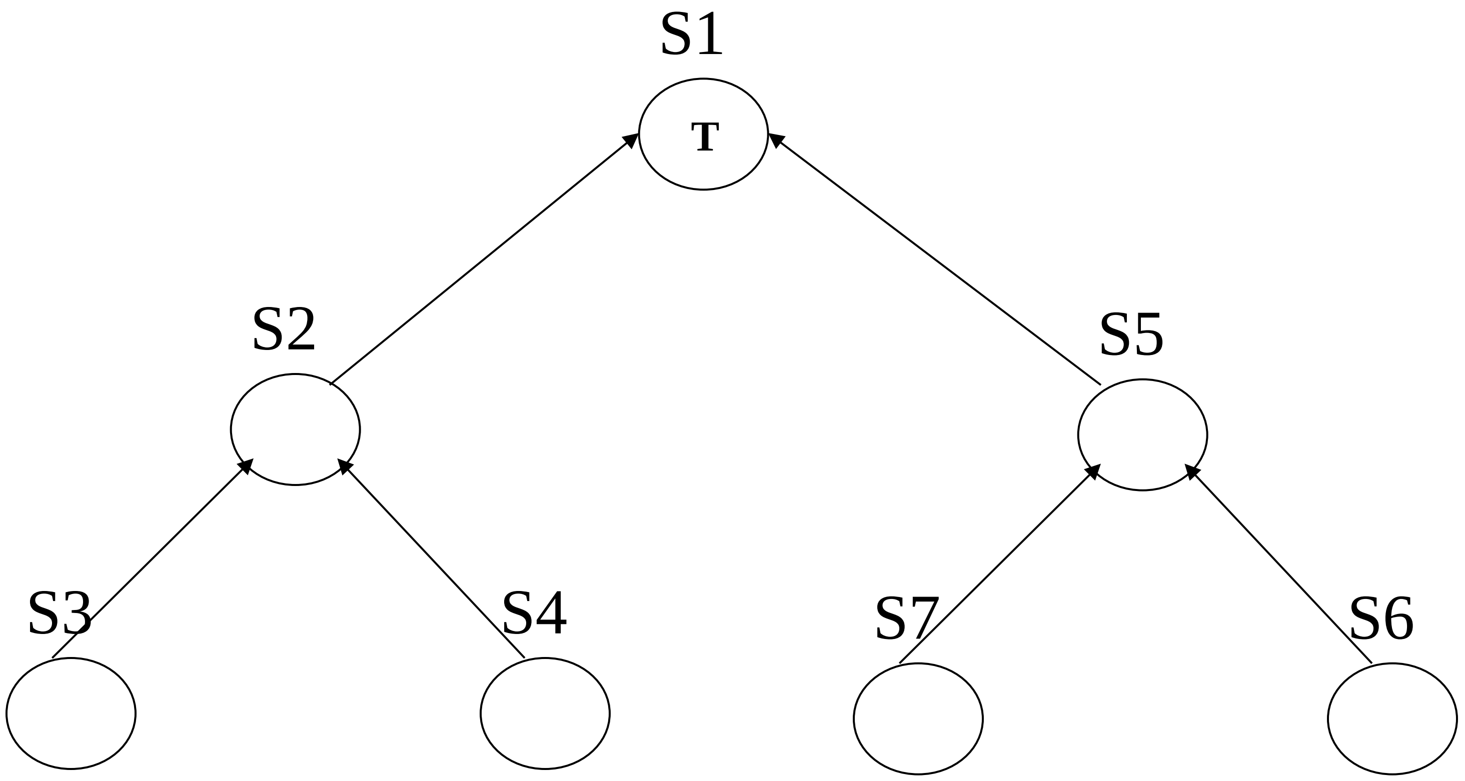
- Singhal's Heuristic Algorithm (cont'd)
  - For all elements in the local Sv array and the token's TSv array, update to the values corresponding to the largest sequence numbers
    - For example, if Sv[6] had state $\mathcal{R}$ and Sn[6] was 43 while TSv[6] was $\mathcal{N}$ and TSn[6] was 44, then the local nodes Sv and Sn arrays should be updated from the token information, but if Sv[6] had state $\mathcal{R}$ and Sn[6] was 44 while TSv[6] was $\mathcal{N}$ and TSn[6] was 43, then the token arrays TSv should be updated from the local node information (i.e. TSv[6] ➔ $\mathcal{R}$)

5. Finally, if there are no $\mathcal{R}$ states in any Sv elements, mark your own Sv entry $\mathcal{H}$, otherwise send the token to some site j such that Sv[j] and TSv[j] shows state $\mathcal{R}$

Distributed Systems

- Raymond's Tree-Based Algorithm
  - In an effort to further reduce message traffic, Raymond's algorithm is implemented by requiring each site to maintain a variable called ***holder,*** which points at the node which this site sent the token to the last time this site had the token (an initial configuration is required).
  - Sites can logically be viewed as a tree configuration with this implementation, where the root of the tree is the site which is currently in possession of the token, and the remaining sites are connected in by ***holder*** pointers as nodes in the tree

Raymond’s Tree-Based Algorithm



Distributed Systems

- Raymond's Tree-Based Algorithm (cont'd)
  1. Each node keeps a queue of requests, and when a node wants to do CS code it places its request in its queue and sends a REQUEST message to the node that its local ***holder*** variable points to, provided that its request queue was empty before this local CS request (if its request queue was not empty before this local request then this node has <u>**already sent**</u> a REQUEST message to the node that its local ***holder*** variable points to

Distributed Systems

- Raymond's Tree-Based Algorithm (cont'd)
  2. When a site receives a REQUEST message from another node it places the message in its queue and forwards a REQUEST message to the node that its local ***holder*** variable points to, provided that it has not already done so on behalf of a preceding message
  3. When the **root** receives a REQUEST message from another node it adds the request to its queue, and when done with the token, sends the token to the requesting node at the top of its queue and redirects its ***holder*** variable to that node. If its request queue is not empty then it also sends a REQUEST to its ***holder*** node.

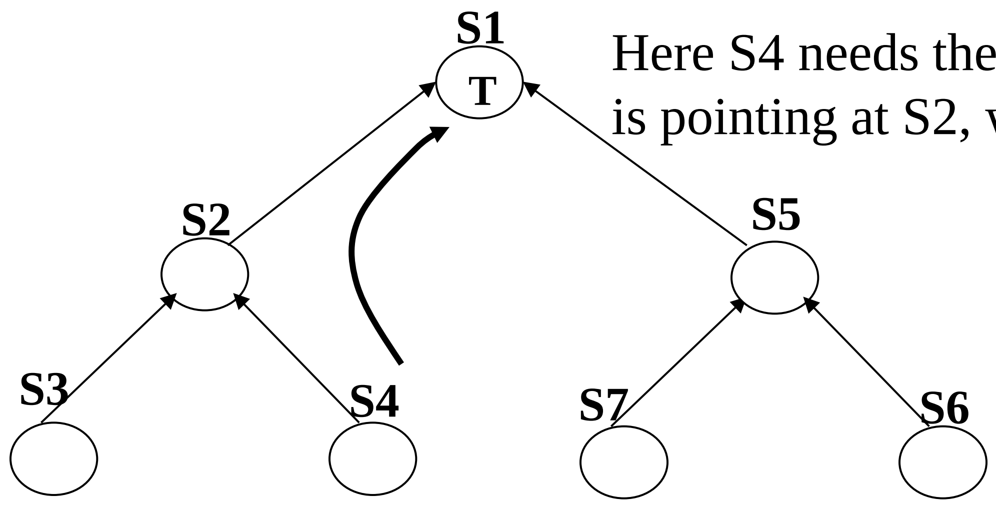
Distributed Systems

- Raymond's Tree-Based Algorithm (cont'd)

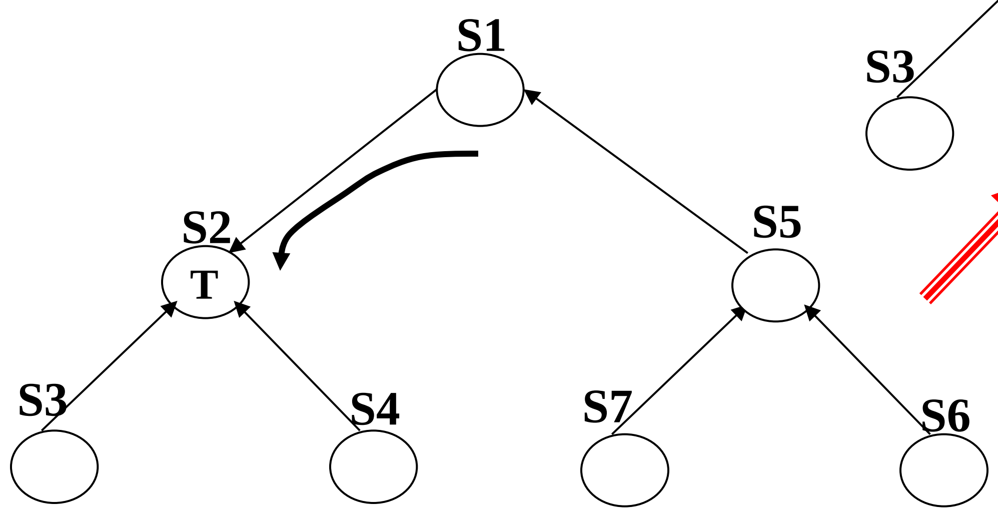
4. When a site receives the token it removes the top request from its request queue and if it's a local request does CS code, otherwise it sends the token to the requesting node. If its request queue is not empty at that point it also sends the requesting node a REQUEST message
5. A completion of CS code is followed by step 3 again

– Raymond's algorithm has a low average message complexity of log(N), but it can have lengthy response times and is subject to fairness problems

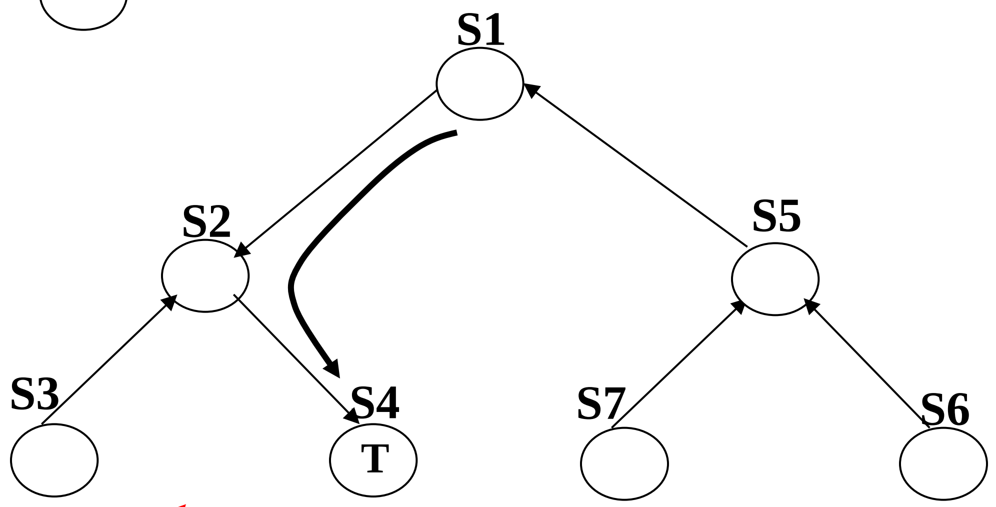
Here S4 needs the token and its ***holder*** is pointing at S2, whose ***holder*** points to S1



S1 returns the token to S2



Finally, S2 returns the Token to S4


S1
S2
S5
S3
S4
T
S7
S6