Homework #2

Nathan Balon
CIS 578
October 31, 2004

1 – Answer the following questions about the snapshot algorithm:

A) What is it used for?

It used for capturing the global state of a distributed system. The global state consists of the state of all nodes and communication channels. A global state is constructed by send a marker message. When a site receives a marker it records its state and if the site has all ready recorded its state it record the messages it received from the channel since it recorded its state. From the individual states of the system a global state can be established. The snap algorithm can be used for purposes such as detecting deadlocks and for performing garbage collection.

B) How does it differ from Lamport's timestamps algorithm in terms of purpose?

Lamport 's timestamps are used to establish a happens before relationship. Lamport's timestamps are used to logically order the message. When a message is received by a node, it will time stamp the receive event, by computing the maximum value of the time stamp it received from another node and the last time stamp the node used and adding one to the value. Lamport's timestamps algorithm is used to partially order messages. On the other hand, the snap algorithm is used to record the global state of the system. One major advantage of the snapshot algorithm over the timestamps algorithm is it records the state of the communication channel. Lamport's timestamps algorithm just records the order in which messages are sent and received between processes it doesn't account for the state of the system.

C) Identify a scenario in which the Snapshot algorithm can be used and one scenario in which Lamport's Timestamps can be used.

Lamport's timestamps algorithm could be used when total ordering is not required. For instance they could be used for a chat server program to order the messages that are received at the server. For a program such as this it only matters what order the server receives the messages for the chat room. The snapshot algorithm can be used for various purposes such as deadlock detection, detecting when there are no more references to an object so it can be garbage collected and debugging a program in a distributed environment. One case where the Snapshot algorithm can be used is in the middleware assignment in class for detecting when it is ok to perform garbage collection on an object. The benefit of algorithm Snapshot algorithm is if the only reference to an object was in the communication channel. It would be unacceptable to garbage collect the object since there is still a reference to the object in the communication channel. Both of these algorithms have there uses when constructing a distributed system.

D) Is the Snapshot algorithm fault-tolerant?

No, the assumption of the algorithm is that there are no failures in either the channel or a process. It is unrealistic that there will be no failures in the system. Lamport's paper on the snapshot algorithm gives no explanation of what to do in the case of a node failing when running the algorithm. If a node failed it would be possible to take

another snapshot of the system. Another way where Lamport algorithm is not fault tolerant is it assumes that communication channels are reliable. So the Snapshot algorithm would only work with certain transport level protocols. If UDP was used and messages were not received the algorithm would fail. The algorithm could be made fault tolerant by combining additional algorithms with it, this is probably what Lamport had in mind when he proposed the algorithm.

2 - Read Maekawa's voting algorithm for mutual exclusion, page 429 of Coulouris' book.
A) How does it differ from the other distributed mutual exclusion algorithms examined in class (in terms of main approach and functionality)?

The other mutual exclusion algorithms covered are either centralized or distributed. In the centralized approach the processes ask a coordinator permission to enter a critical section, which has the problem of the coordinator failing. In the distributed mutual exclusion algorithm all processes must give a single process permission to enter a critical section, which has the problem instead of one point of failure there are multiple points of failure in the system. Maekawa's voting algorithm takes a different approach to the mutual exclusion problem. Instead of have a centralized algorithm or relying on all process to determine if it is alright to enter a critical section a process only needs to receive permission from a subset of processes. The Algorithm can be viewed as having a matrix structure. There is a voting set that determines if processes should be granted for a process to enter a critical section. The voting set is the intersection of the processes row and column in the matrix. The process that then wants to enter the critical section send a message to all the other processes in the voting set. So the number of processes that must give permission to another process to enter a critical section is greatly reduced.

B) Describe its performance in terms of bandwidth, client delay and effect on throughput.

The centralized algorithms will achieve better performance than Maekawa's algorithm but in the case of the coordinator failing it is catastrophic to the system. In terms of the other decentralized mutual exclusion algorithms Maekawa's algorithm achieves better performance. To enter a critical section it take 2 * square_root(N) of messages. To exit the critical section square_root(N) messages needs to be sent. The effective throughput of the algorithm is 3 * square_root(N) messages. In comparison, Ricart and Agrawala's algorithm uses a total of 2(N – 1) messages. So if there were 16 nodes in the system Maekawa's algorithm would send 12 messages where as Ricart and Agrawala's algorithm would require 30 messages. For Maekawa's algorithm the client delay is one round trip time and the synchronization delay is also one round trip time.

3 – Ricart and Agrawala's algorithm has the problem that if a process has crashed and does not reply to a request from another process to enter a critical region, the lack of response will be interpreted as denial of permission. How would you address this problem? What mechanism would you suggest to differentiate between a denial of permission and a crashed process?

Ricart and Agrawala's mutual exclusion algorithm is based on sending a message to all processes and asking permission to enter a critical section. Which ever process has the

lowest time stamp will be able to enter the critical section. If a process receives response from all other processes this indicates it has the lowest timestamp and is able to enter the critical section. If a response is not received from a process, this is taken to mean that the process is in the critical section so it must wait to enter till all other processes are finished. The problem that can arises is if the process that is waiting to enter the critical section never receives the "ok" to enter the critical section it will wait indefinitely to enter. If the processes are multi-threaded and can receive and respond to messages when it is in a critical section the process waiting to enter the critical section could send out a message asking if the process it is waiting on is alive. If the process responds it would be known that is in the critical section, so the other process should continue to wait. For this approach to be feasible the process waiting on the critical section would need knowledge of all the process in the system so it could determine which process to send messages to, to determine if they are still alive. One way to do this is by sending an "are you alive" message to all processes that have not sent it a response indicating it is alright to enter the critical section. If no replies are received from a process after a certain period of time it would be assumed that the process it is waiting on is dead. It could then enter the critical section.

4 – In the bully algorithm, suppose two processes detect the demise of the coordinator simultaneously and both decide to hold an election using the bully algorithm. What happens?

The bully algorithm is an election algorithm that is used to determine a coordinator for a group of processes in the system. When messages are sent to the coordinator and after a set period of time if no response is generated than a timeout will occur and it can be established that the coordinator has failed. When this happens the bully algorithm is run to determine a new coordinator for the group. If two processes simultaneously detect that the coordinator has failed, they will both begin the bully algorithm by sending out an election message. The algorithm assumes that all processes know the ids of other processes, so a message will be sent to all of the processes in the system that has a higher id then themselves. The process that sent the election message will then set a timeout period and wait to see if a message is returned in that period. Whichever of the processes that have a higher identifier will send a response to those with a lower identifier and will begin its own election algorithm in the case of two process simultaneously starting the election process lowest id of the two processes that started the algorithm will receive a response indicating they are out of the running of becoming the new coordinator. So each node that receives a election message and has a higher identifier will runs its own bully algorithm. Eventually the process with the highest id will become the coordinator and send out a message to the other processes indicating it is the new coordinator. If eventually the process that fails come back up it will then hold a new election. The basic idea behind the bully algorithm is when the algorithm ends the node with the highest identifier will be chosen as the new coordinator. So in the case of two processes starting the bully algorithm simultaneously, if one of the nodes has the highest identifier of all nodes in the system then it will be chosen as the new coordinator, if not then other nodes with higher identifiers will run the algorithm.

5 – A) What is the "uncertainty" period in the Two-phase commit protocol?

The two-phase commit protocol is used to ensure that when a transaction completes it is either all carried out or it is aborted. The two-phase commit protocol takes place when a node asks the coordinator to commit a transaction. In the first phase of the two-phase commit protocol the coordinator sends out requests asking participating nodes if it can commit the transaction. The coordinator then receives responses from participating nodes if any node responds with a no, then the transaction is aborted and if the answer is yes, from all nodes then phase two of the protocol takes place. In phase two if all votes were to commit the transaction then the coordinator sends out a "do commit" message or else it aborts the transaction. Next, the nodes receive either a "do commit" or "do abort" message from the coordinator. Then after a "do commit" message was sent to a node in the transaction they send back a have committed message and this completes the protocol. The problem with the two-phase commit protocol is there is an uncertainty period. The uncertainty period occurs after a node sends the coordinator its response to commit or abort a transaction. At this time they are uncertain what will be the outcome and must wait on a message of aborting or committing the transaction. The problem is a node can not continue till it gets a response and in the event that the coordinator has crashed it must wait on a recovery method to be used. So what the problem amounts to is that the two-phase commit protocol is a blocking protocol and nodes will wait indefinitely for a response from the coordinator.

B) Explain how the three-phase commit protocol avoids delay to participants during their `uncertain' period due to the failure of the coordinator or other participants.

The three-phase commit protocol avoids the problem of the two-phase commit protocol by adding an extra round of messages and making it non-blocking. In the first phase the coordinator sends out a commit request message to all nodes and they respond with an abort or agreed message. In the case of a node failing to respond to a request from the coordinator, the coordinator will timeout and send an abort message to all. In the second phase the coordinator sends out prepare messages to all nodes cooperating in the transaction. The nodes then respond to with an acknowledgment message. If the acknowledgment is received from all nodes it moves on to phase three or else it aborts the transaction in the case it doesn't receive an acknowledgment. In phase three the coordinator sends out a commit message. If the coordinator fails after sending a commit message the transaction will be committed on recovery. The benefit of this protocol is it allows a transaction to be aborted if another node fails by timing out the transaction. By using a timeout it avoids the problem of blocking indefinitely.

C) In the three-phase commit protocol, what happens if the coordinator fails just after sending the "Pre-Commit" (or "prepare") message to all participants?

If the coordinator were to fail after sending either a pre-commit or prepare message the transaction would be aborted. The nodes would send a response to either the pre-commit or prepare message but the coordinator would be unable to respond to this message since it has failed. Eventually, the transaction would timeout when messages from the coordinator in the next phase are not received and the transaction would be aborted.

D) In this scenario, what if a process *p* already received a *pre-commit* message and sends an acknowledgment, but another process *q* did not receive the "pre-commit" message, can *p* commit?

No, the coordinator will not receive acknowledgments for all the pre-commit messages. In the case where the coordinator has not received acknowledgments from all participants it will abort the transaction. Process p will receive an abort message from the coordinator because the coordinator did not receive all the acknowledgements.

6 – A) What is meant by "at most once" and "at least once" RPC semantics?

At least once semantics guarantees that a RPC request will be executed at least 1 time and possible more. The problem with at least once semantics is if the operation is not idempotent. For example, it would unacceptable to execute an operation that withdrew money from a bank account twice. At most once semantics guarantees that RPC is executed at most on time and possibly is not executed at all. If an RPC request can not be satisfied, then a failure is reported.

B) How can the underlying RPC protocol provide "at least once" semantics?

At least once semantics can be implemented by the requester continuously sending request until they can be satisfied. For instance if a server crashed it could keep trying to send the request until the server came back online. As mentioned before this has the problem that a procedure may be executed numerous times.

C) How can it provide "at most once" semantics?

A solution to providing at most once semantics is to use timestamps. Each message would contain a connection identifier and a timestamp and the server keeps a table of most recent timestamps. If a message is received that has a lower timestamp it is rejected. If a method such as this is used it will guarantee that a procedure is executed only one time but the problem is a procedure may never be executed.

7- A) ***Vector clocks*** were developed to overcome one of the shortcomings of Lamport's clocks. What shortcoming is it?

The problem with Lamport's timestamps is if a logical time stamp of event A is less then the logical time stamp of event B it can not be concluded that event A happened before event B by examining the timestamps. Lamport's timestamps only create a partial ordering of events. Two processes that have events that don't send or receive messages to each other are not ordered. When using Lamport's timestamps the events that can not be ordered are said to be concurrent events. By using vector clocks you can determine which events are casually related and which one happened first. Vector clocks over come the problem of
L(e) < L(e')
that you can not conclude e -> e'.

B) Briefly describe how this approach works, highlighting the difference between the usual Lamport clocks covered in class

The main difference between the two approaches is for Lamport's timestamps one value is used for the time stamp for all logical clocks in the system and when a message is sent or received the time stamp is incremented to largest value of between the two processes involved and incremented by one. In the case of vector clocks there is a vector the size of all processes in the system. Each process has a vector clock that it timestamps that it uses to record local events in their spot in the vector. So as local events happen the process will set the vector Vi[i] = Vi[i] + 1. When a process sends a message it adds its vector clock to the message. Next, when a process receives a message it will set it vector to the maximum values of value for each position of the vector of the timestamp vector received and the vector for the process. It will basically take the maximum value for each position in the two vectors. So it is possible to compare vectors. It then possible to determine if V(a) < V(b) then a -> b. This was not possible to determine using Lamport's timestamps. It is also possible to tell if two events are concurrent.

8 – Research questions: (20 points)

From all the issues we discussed in class, i.e., synchronization (causality, logical clocks, global state), garbage collection, mutual exclusion, elections, deadlocks, transactions, (concurrency control), how are they (if they are) implemented/addressed by existing technologies, like CORBA, Java RMI, Jini, DCOM, Amoeba, etc..?

Choose a subset (at least 2) of the topics and at least two existing technologies. Compare the two in terms of how the topics you chose are implemented or addressed.

You do not need to hand in any extensive paper. A brief discussion consisting of a maximum of one page for each topic selected will suffice.

CORBA and RMI are two competing middleware technologies with similar functionality. Although the functionality is similar they take different approaches in there implementations. There are many similarities and differences between the two middleware's. CORBA and RMI both implement garbage collection and transactions differently.

Garbage collection is performed differently in the two protocols. First, in CORBA there is no automatic garbage collection, on the other hand RMI has built in garbage collection. In CORBA it is the programmer's responsibility to delete an object similar to C++. There are a few add-on packages which were developed by different vendors that are available for garbage collection, but the standard says nothing on how garbage collection should be implemented. Java RMI on the other hand has garbage collection built into the protocol. RMI uses reference counting to perform garbage collection. The JVM has a garbage collector that automatically deletes an object when there are no more references to it. RMI uses the Distributed Garbage Collector that works by having an external server keep track of all the references that external clients have to an object. When an external process gets a reference to an object it increments the objects reference count by one. When an external client does not refer an object any more then Distributed Garbage Collector decrements the reference count by one. If the distributed objects reference count has dropped to zero it becomes a weak reference. If there are

no local objects referring to the weak reference it will be marked as a for garbage collection and when the garbage collector runs it will free the resource. So, for a resource to be freed by the Distributed Garbage Collector there must not be any local or remote references to it. This is the basic approach that RMI uses for distributed garbage collection.

The two middleware products also vary in there support for transactions. Java RMI has no built in support for transactions. If a transaction algorithm is need it is up to the programmer to implement it themselves. Two of Sun Microsystems other distributed technologies Jini and EJB both provide support for transaction but at the current time there is transaction service that is build into RMI. As with the case for distributed garbage collection for CORBA there are third parties that sell transaction services for RMI. CORBA on the other hand has the transaction service specification. CORBA provides the Object Transaction Service (OTS) which implements the two-phase commit protocol. The OTS provides a transaction coordination framework to achieve the ACID property.

CORBA and RMI take different approaches to both transactions and garbage collection. RMI provides support for garbage collection while CORBA does not. In the case of transactions, CORBA provides the OTS where in RMI controlling transactions it is left up to the programmer.