School of Computer Science, McGill University

COMP-512 Distributed Systems, Midterm Fall 2013

24-October-2013 – Duration: 75 min

1 Total Order Multicast (40 points)

In class, we discussed that group communication systems provide reliable message delivery and ordering guarantees. We discussed two reliability levels: reliable delivery and uniform reliable delivery.

Uniform reliable delivery guarantees that whenever the AL of any process receives the message, then the AL of all correct processes receive it. (page 13, lower slide of GC lecture). We also indicated shortly, that in order to achieve it, the CL of a process may only deliver a message to the AL once it knows that the CL of all other processes have received it. That is, they might not yet have delivered it to their AL, but they have already received it from the underlying network layer. (page 16 of GC lectures). We also indicated in class that this can be achieved with the following. The CL of each process, upon receiving a message m, multicasts an acknowledgement to all other processes. Once the CL has received an acknowledgement for a message m from all other processes uniform reliability is achieved and it can deliver m to the AL (in the proper order). These acknowledgements can create a lot of extra messages. An alternative is to piggyback such information on other messages.

Let's now have a look at one of the sequencer approaches discussed in class (Handout about GC, page 9 upper slide). In this algorithm, all nodes send their messages to the sequencer. The sequencer multicasts its own messages and the messages from the others using FIFO-multicast. The other nodes deliver the messages they receive from the sequencer in FIFO order (which builds a total order because the sequencer is the only one multicasting).

If this algorithm wants to support uniform reliable delivery, it has to be extended. A possibility is to use a piggyback mechanism where nodes piggyback with their application messages they send acknowledgments of messages they have received.

Provide an algorithmic description of the sequencer based algorithm described above that provides uniform-reliable delivery. All acknowledgement information should be piggybacked on application messages. Assume that each node sends messages on a regular basis so that delay remains reasonable. Try to find a scheme that uses compact data structures that implicitly acknowledge several messages. Use a proper message identifier / message sequence number scheme. Make sure that any process only delivers a message to the application once it has acknowledgements from all nodes for that message. You can assume that you have a reliable FIFO-multicast that can be used as underlying layer.

Solution Outline 1: Each process keeps track of what other processes have received; info forwarded through sequencer

- At any process p_i:
  - On init:
    - $Received[i] = 0, 1 \leq i \leq n$
    - delivered $= 0$
    - $InQueue = \{\}$

- Upon TO-multicast(g,m) from AL
  - send (g,m, Received[i]) to sequencer // piggyback information of what p_i has seen
- Upon $FIFO-receive(g, m, R, p_j)$ from sequencer // piggyback information of what sender of message has received so far
  - Append m to $InQueue$ // store in receive order as this will be delivery order (FIFO-mc)
  - if $j \neq i$, then Received[j] = R // keep track of which messages the others have received
  - Received[i]++ // p_i has now received one message more
  - minR = $min(Received[k], 1 \leq k \leq n)$ // deliver messages all have received in FIFO order
  - If $minR > delivered$
    - for i = delivered+1 to minR
      - remove first message m from $InQueue$ and deliver
    - delivered = minR
- Additionally at sequencer p_i:
  - Upon receive (g,m,R) from process p_j // sequencer simply forwards piggybacked info
    - $FIFO - multicast(g, m, R, p_j)$ to all

Solution Outline 1: Sequencer keeps track of what other processes have received

- At any process p_i:
  - On init:
    - $Received = 0$
    - $Delivered = 0$
    - $InQueue = \{\}$
  - Upon TO-multicast(g,m) from AL
    - send (g,m, Received) to sequencer
  - Upon $FIFO - receive(g, m, confirmed)$ from sequencer
    - Received = $m.seq$ // could also simply do Received++
    - Append m to $InQueue$
    - for i = Delivered+1 to confirmed
      - remove m with $m.seq = i$ from $InQueue$ and deliver // could also simply remove confirmed-delivered first messages and deliver as they are in proper order
    - Delivered = confirmed
- Additionally at sequencer p_i:
  - On init:
    - $ReceivedVector[j] = 0, 1 \leq j \leq n$
  - Upon receive (g,m,R) from process p_j
    - $ReceivedVector[j] = R$ // keep track of what each process has received
    - confirmed = $min(ReceivedVector[k], 1 \leq k \leq n)$ // take minimum over all
    - $FIFO - multicast(g, m, confirmed)$ to all

2 Performance Evaluation (40 Points)

We discussed in class that one can handle increasing load to a service if one installs several instances of the service. Assume a service implemented as a remote object. In order to achieve scalability, there can be several instances of the remote object. A load-balancer is located in front of the system that knows about all currently existing remote object instances and distributes the current load equally across these instances. Assume for simplicity, that the service is some compute intensive job that does not need to maintain a lot of mutable state. All mutable state is maintained in a central storage, and if an object instance wants to change the state or access this state it has to make a call to this central storage. That is, each object instance can serve requests independently and does not need to communicate with the others. Figure 1 below illustrates the architecture.

As the amount of load submitted to the system can change, such a replicated system is ideally *elastic*. At a steady workload, sufficient object instances exist to handle that workload. When the load increases, new object instances are dynamically added, while when the workload decreases, unnecessary object instances are removed (so that the nodes can be used for other things).

The "goodness" of such an elasticity approach is measured as *agility*. At a given time, an agility value of 0 means that the system has exactly the right amount of object instances as to handle the current load given a QoS requirement: response time must remain below a certain value. An agility value above 0 means that it either has too few instances (the instances are overloaded and the response time is too high), or too many instances (the nodes holding the instances are underloaded and could handle more load without violating response time requirements).

In the following you have to analyze the agility of three different systems.

- The first, referred to as *Over provisioning* does not provide elasticity. Instead, it simply knows the maximum load that will ever be submitted and has always the same number of object instances, namely the minimum number of instances that is required to handle that maximum load without violating response time requirements.
- The second, referred to as *CloudWatch* provides dynamic provisioning, that is it monitors the CPU and memory of the nodes with object instances and creates new object instances when the utilization goes above an upper threshold as well as removes object instances when the utilization goes below a lower threshold.
- Finally, *ElasticRMI* also provides dynamic provisioning but does not only take CPU and memory into account when deciding on the number of object instances but also takes application-specific performance metrics into account (e.g., how long an object instance waits on average to access the central storage...).

Figure 2 below shows a periodic load pattern over time, that is it shows how many requests per minute were submitted over a time interval of 500 minutes. Figure 3 shows the agility values of the compute nodes of the three systems over time for this periodic load. Note that you can assume that the load balancer and the storage are not overloaded, their agility values are not considered.

- *Describe the observations you can see in the agility Figure 3.*
- *Provide an explanation for the behaviour.*

- Observations:
  - The agility value of *over provisioning* oscillates with the workload changes: when workload increases, agility goes down; when the workload decreases, agility goes up. It is 0 with the highest workload, and the highest (around 40) with the lowest load.
  - CloudWatch and ElasticRMI have much lower agility values except of when the workload is the highest. CloudWatch has always slight higher values than ElasticRMI, with a maximum of 5 compared to ElasticRMI whose maximum seems to be 2 or 3.
  - CloudWatch also oscillates a bit but at much lower values than overprovisioning. There is also no clear correlation between workload changes and agility values. In contrast ElasticRMI is always very low so it is difficult to observe whether there is any oscillation.
- Reasons
  - Overprovisioning has exactly the resources to handle the maximum value. Therefore, when the load is the highest, the agility value is the lowest (and mostly lower than the other two approaches) because at this time, all resources are optimally utilized but the system is not overloaded. When the load decreases then the system is more and more underloaded, leading to higher agility values as the system is not optimal utilized. Accordingly the agility values go again down when the load increases because the replicas become again more utilized.
  - CloudWatch observes the resource utilization and provide dynamic provisioning according to the utilization but reacts late. So when the load changes the agility values go up. When the workload increases, the system is shortly overloaded leading to higher agility values. It then reacts on this CPU/memory overload (likely adding more resources) and the agility value goes down. Similar, when the load goes down from the highest value, agility goes shortly up, likely because the CPU/memory of the replicas are underutilized, until CloudWatch reacts and removes replicas, and brings the agility value down again. Thus, in contrast to overprovisioning, higher agility values are due to overload and underload depending on the workload tendency.
  - ElasticRMI seems to react always very fast and, thus can keep the agility value always down. Adding some application-awareness additionally to resource utilization in order to determine the right amount of instances, makes it faster to react and never get into any significant overload/underload situation.

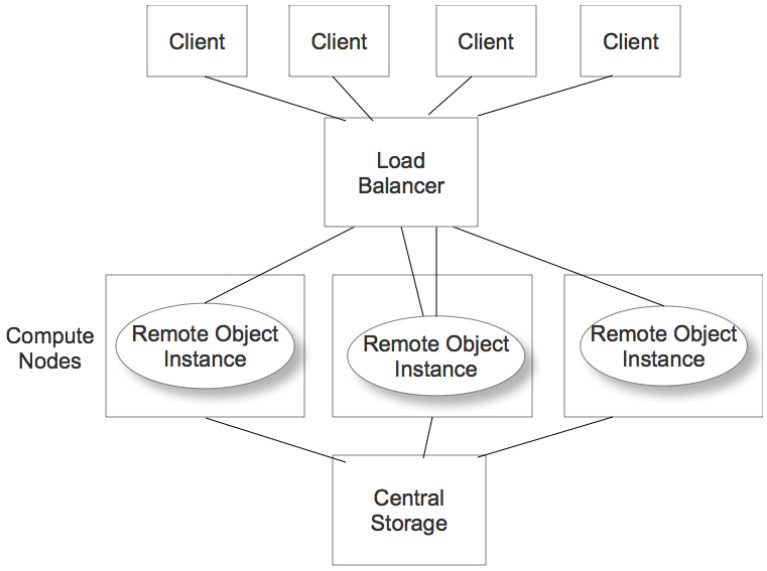

Figure 1: Architecture

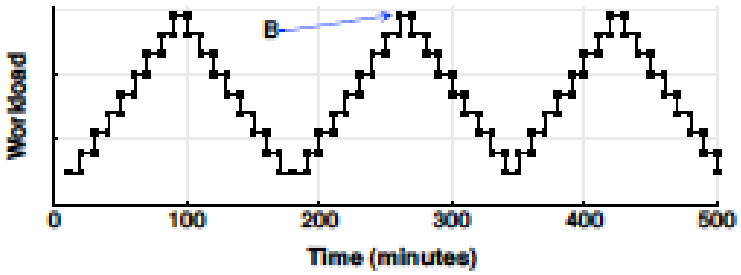

Figure 2: Periodic load pattern

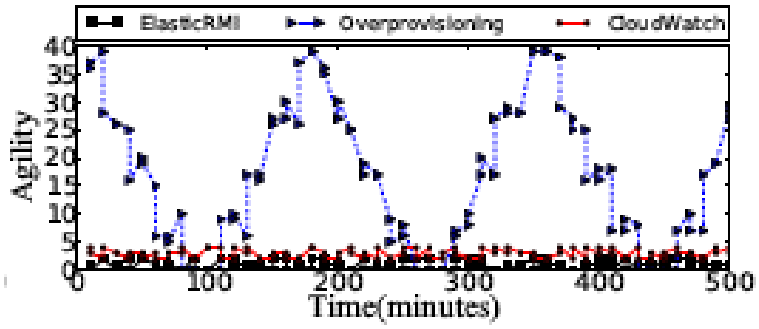

Figure 3: Agility

3 Distributed Transaction Management (20 Points)

Assume a distributed database with three nodes A, B, and C, where A holds data item a, B holds data item b, and C holds data item c. Assume all requests of all clients go through A who redirects requests to the nodes with the corresponding data items (including itself). Only A has a lock manager installed and acquires the necessary locks before forwarding/executing operations. Now assume a client submits the following transaction:

$r(b), r(c), w(a), w(b), w(c)$

Show the execution as time proceeds (similar to the figures developed in class and in the homework). Indicate the messages exchanged, when locks are requested/granted and when and where operations are executed.

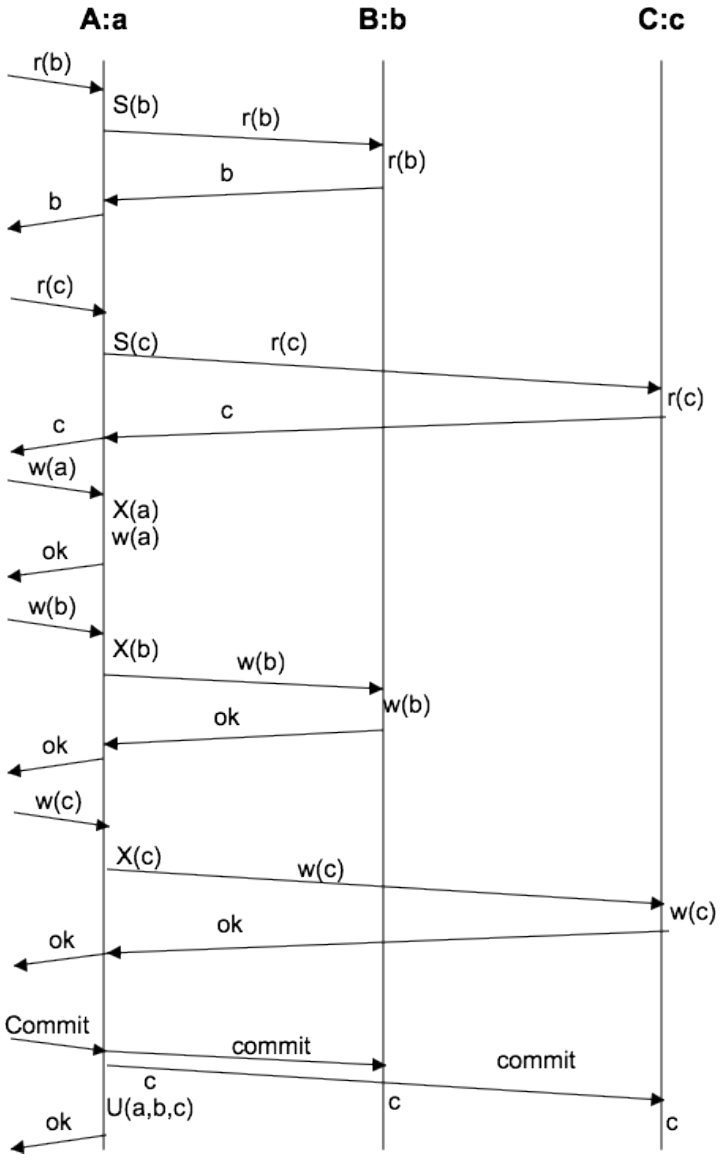

Figure 4: Transaction Execution