School of Computer Science, McGill University

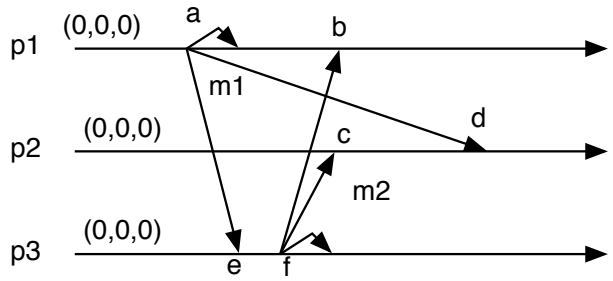
COMP-512 Distributed Systems, Midterm Fall 2015

29-October-2015 – Duration: 75 min

1 Minis (30 points)

1. Consider the causal multicast protocol based on vector timestamps discussed in class (and copied below). The figure below shows how the multicast layer multicasts messages to the other sites and how it receives such messages from the underlying communication system. (It doesn't show how the application submits the request or receives the message).

   For the timepoints a to f in the figure, indicate the value of the vector clocks of the corresponding processes after the event is completed.



   For the timepoints a to f in the figure, indicate the value of the vector clocks of the corresponding processes.

   - a: (1,0,0),
   - b: (1,0,1) (after delivering m2)
   - c: (0,0,0)
   - d: (1,0,1) (after delivering m1 and m2)
   - e: (1,0,0) after delivering m1
   - f: (1,0,1) after delivering m2

   I accepted other values if proper explanation was given

2. Assume the following schedule involving three transactions

   $r_1(b), r_2(a), w_3(b), c_3, w_1(a), c_1, w_2(a), c_2$.

   Let's have a look at the schedule in a centralized system. Assume that the sequence of read/write operations given in the schedule is the sequence in which operations are submitted. Assume the DBS uses strict 2PL.

   Show how and when internal locks are requested and released, and whether/when transactions commit/abort similar to the examples given in class. If an operation is blocked because of a lock conflict all following operations of the same transaction are also blocked. Give a conflict equivalent serial schedule (omitting any aborted transactions).

   $S_1(b), r_1(b), S_2(a), r_2(a), X_3(b)(blocked)X_1(a)(blocked), X_2(a)w_2(a), c_2, U(a), w_1(a), c_1, U(a,b), w_3(b), c_3$
   equivalent schedule: T2, T1, T3

3. Look at the following two variants of 2PC for a system with n nodes considering only the default case of successful execution with commit decision and no failures.

   1. In the 2PC protocol discussed in class (one coordinator and $n-1$ participants), the coordinator sends vote requests to all participants, the participants respond to the coordinator each with a yes vote, the coordinator decides to commit and sends a commit message to each participant who then commits upon reception of the message.
   2. An alternative is as follows (one initiator and $n-1$ participants). The initiator sends its yes vote to all participants. Upon receiving this vote, each participant sends its vote to all (coordinator and other participants). For each node (coordinator or participant), once it has received the votes of all, it decides.

   For each of the two variants, indicate the total number of messages (for the commit case without failures), and the number of message rounds until completion of the protocol at all nodes (i.e., the number of messages that are sent sequentially).

   1. number of messages: 2PC: 3*(n-1), alternative: n*(n-1)
   2. message rounds: 2PC: 3, alternative: 2

2 Algorithm (35 Points)

Assume a distributed database system as depicted in Figure 1. There are n data stores and one central scheduler. The data is distributed across the data nodes (e.g., x resides on d_1, y on d_2 and z on d_3). A client can connect to any of the data stores and then submits all requests (beginTxn, read, write, commitTxn) to this one data store. The following can be assumed (it simplifies the concurrency control tasks).

- Each transaction reads a data item at most once.
- Each transaction reads only data items that are local to the data store to which it connects. However, it might write data items of any of the data stores.
- A write operation on data item x writes the full data item. That is, it does not need to read the data item before writing it.

Develop an algorithm that provides a snapshot isolation concurrency control mechanism where the central scheduler validates all transactions upon commit time. Your algorithm should depict (i)

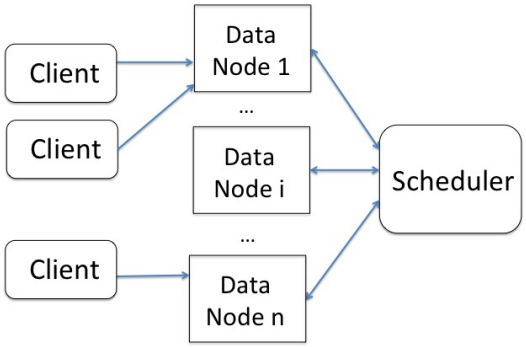


Figure 1: Distributed Data Architecture

initialization of any data structures, counters, etc. that you might need on any data store d_i and the central scheduler, (ii) the actions that are performed when a client submits a $beginTxn(T_i)$, a $read(x)$ of T_i, a $write(x)$ of T_i and $commitTxn(T_i)$ request to a data node d_j. The algorithm should indicate whenever one node requests something from a different node, and how this node reacts when receiving such a request.

- Init at d_j:
  - $CT_j := 0$
- Init at *scheduler*:
  - $CT := 0$
- $beginTxn(T_i)$ submitted to d_j
  - $StartTC := CT_j$
  - $WS(T_i) = \emptyset$
  - return ok
- upon read(x) of T_i to d_j // guaranteed to be on local data item
  - return x_{ts} such that $ts = max(t)$ where x_t is an existing version of x and $t \leq StartTC(T_i)$
- upon write(x) of T_i to d_j // store them locally for now
  // this is possible because the full data item is written, so no previous version is needed
  // of course, it was also ok to send the write operation to the remote site responsible for it
  - $WS(T_i) := WS(T_i) \cup \{x\}$
  - return ok
- Upon $commitTxn(T_i)$ submitted to d_j
  - Send $(T_i, WS(T_i), StartTC(T_i))$ to the central scheduler
- Upon central scheduler receiving $(T_i, WS(T_i), StartTC(T_i))$ from d_j (atomic action)
  - If $\exists j : StartTC(T_i) < j \leq CT$, such $CommitTC(T_k) = j$ and $WS(T_k) \cap WS(T_i) \neq \emptyset$
    - return $abort(T_i)$ to d_j
  - else
    - $CT := CT + 1$
    - $CommitTC(T_i) := CT$
    - for each $1 < k \leq n$, let $WS_k(T_i) := \{x \in WS(T_i) | x \in d_k\}$ (x managed by d_k)
      - send $(T_i, WS_k(T_i), CT)$ to d_k

- Upon d_j receiving $abort(T_i)$ from central scheduler
  - abort locally (discard writeset etc. from T_i)
  - return abort to client
- Upon d_j receiving $(T_i, WS_j(T_i), CT)$ from scheduler
  - for each $x \in WS_j(T_i)$ create version x_{CT} and append to data store
  - $CT_j := CT$
  - if d_j local node of T_i
    - return commit to client

Obviously, many alternatives are possible. As I mentioned in class, many of you maintained the commit counter only at the central scheduler and got the current value at the start of a transaction. The problem is that the scheduler increases the counter before sending the writesets to the data nodes. Thus, it is possible to get a start timestamp but the writeset of a transaction with a smaller commit timestamp has not yet been applied. These data versions would be missed.

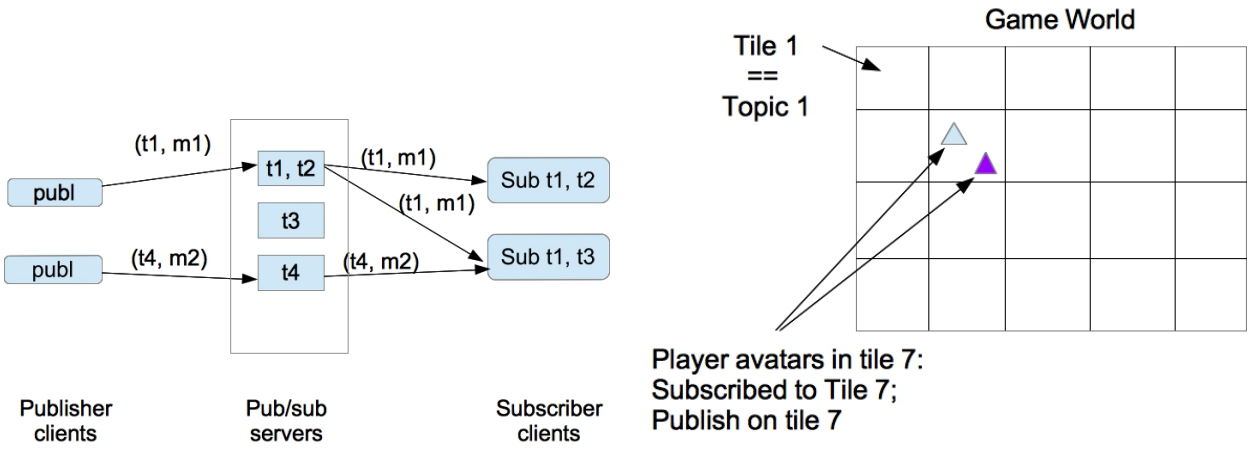
In principle, the flow is the same. But now messages are sent, and every sent message must trigger an action at the recipient. With this, the protocol becomes distributed....

3 Performance Analysis (35 Points)

Figure 2(a) shows a scalable publish-subscribe architecture. A basic pub/sub system works as follows. Publisher clients send publications. Each publication is tagged with a *topic*. Subscriber clients subscribe to topics. When the pub/sub server receives a publication for topic t it sends that message to all clients that have subscribed to t. In the figure, the top subscriber has subscribed to topics $t1$ and $t2$ and the bottom subscriber to $t1$ and $t3$. The pub/sub server keeps track of these subscriptions. When now the top publisher client sends a publication on topic $t1$ (which includes the topic name and the message body), the pub/sub server forwards the publication to both subscribers. Note that a client can be both publisher and subscriber (to the same or different topics). If there are too many messages, a single server might become overloaded. In this case, a scalable system can distribute topics among many server nodes. The figure shows a system with three servers, where the top server is responsible for $t1$ and $t2$, the middle server for $t3$ and the bottom server for $t4$. That is, publishers send publications on $t1$ and $t2$ to the top server, etc.

We look at two scalable pub/sub systems, referred to as *Dynamoth Load Balancer* (DLB) and *Consistent Hashing* (CH). DLB uses two types of load-balancing events/actions: First, when it detects that a server is overloaded it tries to move some of its topics to other servers (for instance, if the top pub/sub server becomes overloaded it could move topic $t2$ to one of the other servers). Second, when it detects that all servers are too loaded, it automatically adds a new server to the system and moves topics from the existing servers to this new server. CH has only one mechanism: when it detects that a server is too loaded, it activates a new server and moves half of the load of the overloaded server to the new one.

Both approaches are evaluated with a gaming application (Figure 2(b)): the game world is split into rectangular tiles (the game world has more tiles than shown in the figure). Each player should only see the actions of players that reside in the same tile as the player itself. Thus, each tile is modelled as a topic. Each player subscribes to the tile in which its player figure resides. When a player moves it sends a publication on the topic of its current tile; the move action itself is in the message body. With this all other players in the same tile will receive the move action and can render it on their game screen.



(a) Scalable Pub/Sub Architecture (b) Gaming Application

Figure 2: .

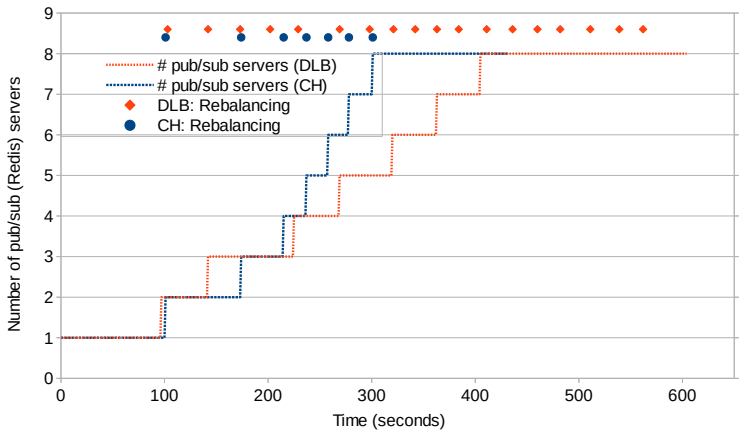


Figure 3: Number of Server Nodes

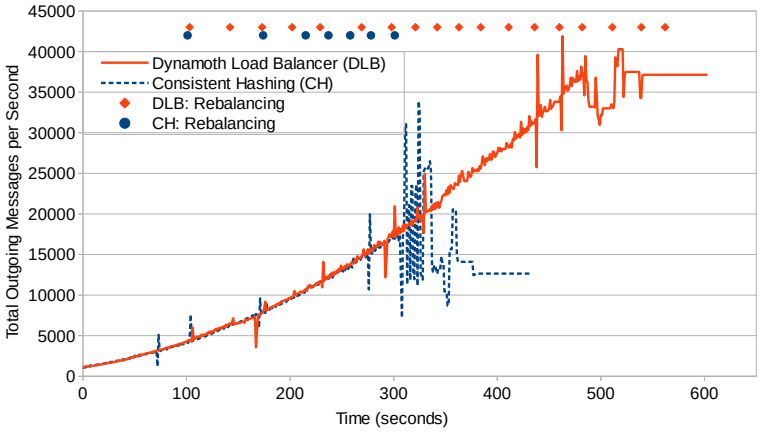


Figure 4: Number of Messages that the system can send

The evaluation is performed as follows. A game is started and as time progresses, more and more players join the game in a liner fashion. More precisely, at time 0, there are 150 players. From then on, in every 100 second interval, 150 new players join the game (100 sec = 300 players, 200 sec = 450 players, etc.) until 1100 players are in the the game shortly after 600 seconds. They are all randomly distributed across the game world and start moving when they join. As there are more and more players, more publications are sent, and each topic has more subscribers. The number of topics remains the same and is equal to the number of tiles of the game world.

Figures 3 and 4 show the behaviour of the system as time progresses and the load to be handled increases. The dots at the top of both figures show the time points where a load balancing event takes place and changes the configuration. The y-axis of Figure 3 shows the number of servers that the system allocates to handle the load. At start the system uses one server and a total of 8 servers are available. The y-axis of Figure 4 shows the number of messages the pub/sub servers actually send to the subscribers.

1. *For each of the figures describe all the observations that you can make. For performance differences, you have to decide whether they are significant or "noise".*

   (a) Number of load-balancing events. DLB has more load-balancing events at the beginning also adding quicker the 2nd and 3rd server than CH. Then CH has more events until all 8 servers are used at time 300. After that no more load balancing occurs for CH. DLB has regular load-balancing events after that.

   (b) Number of servers vs. load-balancing events. At each load-balancing event CH adds a new server until all 8 servers are used. DLB has a mix of load-balancing events that add new servers and load-balancing events that only shuffle the load. CH uses all 8 servers at time point 300 while DLB adds new servers later and has the full server capacity only after 400 seconds.

   (c) Number of messages. The number of messages sent increases with time and is the same for both systems. Shortly after 300 ms, CH does not further increase the number of outgoing messages but observes bursts and irregularities around 15,000 msg/sec. That occurs at a time when all 8 servers are provisioned. After 400ms no measurements are taken anymore because the system can obviously not handle the load. In contrast, for DLB the number increases until around 500 ms (quite a bit after all 8 servers are installed), and then stabilizes at around 35,000 msg/sec with a somewhat irregular behavior until the experiment finishes at 600 sec.

   (d) Both approaches have some irregularity in the throughput time. The spikes usually coincide with a load balancing event.

2. *Attempt to find explanations for the behavior and any type of performance difference.*

   (a) Number of load-balancing effects. At one server, overload of the server triggers the addition of a new server for both systems (no other option). Then, except of the move from 2 to 3 servers (which is probably noise), CH adds quicker a new server than DLB. The reason is that CH cannot redistribute load among existing servers but can only add a new server to share the load with the overloaded server. Due to these limits in load redistribution it has to use all 8 servers quicker. It's likely that the 8 servers are not equally loaded. However, as the load cannot be redistributed, once one or more servers are overloaded they cannot handle the amount of outgoing messages and performance

is compromised. In contrast, DLB has several load-balancing events where it simply reshuffles the load, trying to distribute it as equal as possible among the servers; only when all are loaded it adds a new server. Even with 8 servers, it is able to continuously readjust the load to avoid overload of any server.

(b) The number of messages reflects the throughput of the system. Thus, it is the same for both systems as long as they are not overloaded. At the overload point the system does not deterioriate a lot (in the sense of sending much less messages than before). It's likely that incoming messages are simply dropped to not further overload the system.

(c) The irregularities at load-balancing effects likely occur because topics are moved from one server to another leading to delay for some messages (leading to decreased throughput in that interval) or less delay for a certain time period (higher throughput). In fact, for DLB one can observe several times, especially with higher load that there is first a short decrease in throughput (messages are delayed in the system due to transfer to a different server) with an immediate increase in the throughput (the delayed message are all delivered in the next time interval).

As I mentioned in class, throughput increases exponentially with the number of players and not linear. The reason is that every player update in a tile will lead to an update message to all other players in the tile. Thus, if there are n players in a tile and each moves every x seconds, then there are $n * n$ messages to be sent in that time unit.