Outline for February 13, 2001

1. Greetings and felicitations!
2. Suzuki-Kasami’s broadcast protocol
   a. token-based
   b. uses sequence numbers, not clocks
   c. token has sequence numbers, associated queue
   d. how to handle stale requests? token’s sequence number too high
3. Raymond’s tree-based protocol
   a. token-based
   b. think of token as at root of tree, root moves around
4. Distributed Agreement Protocols: system model
   a. synchronous vs. asynchronous
   b. different types of failure (crash, omission, malicious)
   c. authentication
5. Classification: agreement (on value), validity (the right value)
   a. Byzantine problem (all agree, initial value of source); review Byzantine Generals’ problem
   b. consensus problem (all agree, if initial value of nodes is same, the final value is that value)
   c. interactive consistency problem (all agree on same vector, if $i$th processor non-faulty, $i$th element of vector is the value of that node)
   d. relationship
6. Solution to Byzantine Problem
   a. Can show: if $3m$+1 processors, at most $m$ can be faulty or agreement cannot be reached.
   b. Demonstration with 3 processors.
   c. Lamport-Shostak-Pease algorithm
7. Application: clock synchronization in the face of faults
   a. interactive convergence algorithm
   b. interactive consistency algorithm
Suzuki-Kasami Broadcast Protocol

Introduction

This is a token-based protocol. Unlike non-token-based ones, it uses the token’s being possessed by a site to provide ordering of requests. Clocks and virtual time do not play a role; but order of arrival does.

Notation

- \( n \) processes
- \( p_i \) process
- \( R_i[j] \) largest sequence number \( p_i \) has received in a REQUEST message from \( p_j \)
- \( L[i] \) sequence number of request that \( p_i \) has most recently executed
- \( Q \) queue (sequence) of sites requesting token
- \( T = (Q, L) \) token

Protocol

1. To request entry, if \( p_i \) does not have the token, it increments its sequence number \( R_i[i] \) and then sends REQUEST\((i, s)\), \( s = R_i[i] \), to all other sites.
2. When \( p_i \) receives REQUEST\((i, s)\) from \( p_j \), \( p_i \) sets \( R_i[j] = \max(R_i[j], s) \). If \( p_i \) has the token and \( R_i[j] = L[j] + 1 \), it sends the token to \( p_j \).
3. If \( p_i \) is requesting entry and it has or receives the token, it enters the critical section.
4. When \( p_i \) finishes executing the critical section:
   - it sets \( L[i] = R_i[i] \);
   - for every \( j \) not in \( Q \) and for which \( R_i[j] = L[j] + 1 \), \( p_i \) appends \( j \) to \( Q \); and
   - if \( Q \) is not empty, \( p_i \) deletes the first element \( f \) of \( Q \) and sends the token to \( p_f \).

Example

There are three processes, \( p_1, p_2, \) and \( p_3 \). \( p_1 \) and \( p_3 \) seek mutually exclusive access to a shared resource.

Initially: the token is at \( p_2 \) and the token’s state is \( L = [0, 0, 0] \) and \( Q \) empty;
- \( p_1 \)’s state is \( C1 = 0, R1 = [0, 0, 0] \); \( p_3 \)’s state is \( C1 = 0, R2 = [0, 0, 0] \); and \( p_3 \)’s state is \( C3 = 0, R3 = [0, 0, 0] \)
- \( p_1 \) sends \( R(1, 1) \) to \( p_2 \) and \( p_3 \); \( p_1 \)’s state is \( C1 = 1, R1 = [1, 0, 0] \)
- \( p_3 \) sends \( R(3, 1) \) to \( p_1 \) and \( p_2 \); \( p_3 \)’s state is \( C3 = 1, R3 = [0, 0, 1] \)
- \( p_2 \) receives \( R(1, 1) \) from \( p_1 \); \( p_2 \)’s state is \( C2 = 1, R2 = [1, 0, 0] \), holding token
- \( p_2 \) sends the token to \( p_1 \)
- \( p_1 \) receives \( R(3, 1) \) from \( p_3 \); \( p_1 \)’s state is \( C1 = 1, R1 = [1, 0, 1] \)
- \( p_3 \) receives \( R(1, 1) \) from \( p_1 \); \( p_3 \)’s state is \( C3 = 1, R3 = [1, 0, 1] \)
- \( p_1 \) receives the token from \( p_2 \)

\textit{p1 enters the critical section}

\textit{p1 exits the critical section} and sets the token’s state to \( L = [1, 0, 0] \) and \( Q = (3) \)
- \( p_1 \) sends the token to \( p_3 \); \( p_1 \)’s state is \( C1 = 2, R1 = [1, 0, 1] \), holding token, token’s state is \( L = [1, 0, 0] \) and \( Q \) empty
- \( p_3 \) receives the token from \( p_1 \); \( p_3 \)’s state is \( C3 = 1, R3 = [1, 0, 1] \), holding token

\textit{p3 enters the critical section}

\textit{p3 exits the critical section} and sets the token’s state to \( L = [1, 0, 1] \) and \( Q \) empty
Raymond’s Tree-Based Protocol

Introduction
This is a token-based protocol. The nodes are arranged in a binary tree, and one acquires the token by going up the tree. The token is always kept at the root, so the tree needs to rearrange itself as the token floats from site to site.

Notation
- \( n \) processes
- \( p_i \) process
- \( Q_i \) request queue (sequence) of sites associated with process \( p_i \)
- \( H_i \) holder variable associated with process \( p_i \)
- \( T \) token

Protocol
1. To request entry, if \( p_i \) does not have the token, it sends a REQUEST\((i)\) message to the node named in \( H_i \) unless \( Q_i \) is not empty (because then it has already sent a REQUEST\((i)\) but has not yet received the token). It adds the request to \( Q_i \).
2. When \( p_i \) receives REQUEST\((j)\) from \( p_j \):
   a. if \( p_i \) does not have the token, it places the REQUEST\((j)\) on \( Q_i \) and sends a REQUEST\((i)\) message along (provided that it is not waiting for a response to an earlier REQUEST\((i)\)).
   b. if \( p_i \) has the token, it sends the token to \( p_j \) and sets \( H_i \) to \( j \).
3. If \( p_i \) is requesting entry and receives the token:
   a. if \( i \) is not the first entry in \( Q_i \), it deletes the first entry \( j \) from \( Q_i \) and forwards the token to \( p_j \). It then sets \( H_i \) to \( j \). If \( Q_i \) is not empty, \( p_i \) sends REQUEST\((i)\) to \( p_j \).
   b. if \( i \) is the first entry in \( Q_i \), \( p_i \) deletes \( i \) from \( Q_i \) and enters the critical section.
4. When \( p_i \) finishes executing the critical section:
   a. if \( Q_i \) is not empty, it deletes the first entry \( j \) from \( Q_i \), sends the token to \( p_j \), and sets \( H_i \) to \( j \)
   b. if after step a \( Q_i \) is not empty, \( p_i \) sends REQUEST\((i)\) to \( p_j \).
Example

There are six processes, \( p_1 \) through \( p_6 \). \( p_1 \) and \( p_5 \) seek mutually exclusive access to a shared resource, and later \( p_3 \) will request it.

Initially: \( p_4 \) has the token;

- \( p_1 \)'s state is \( C_1 = 0 \), HOLDER2 = \( p_3 \), Q1 empty
- \( p_2 \)'s state is \( C_2 = 0 \), HOLDER2 = \( p_3 \), Q2 empty
- \( p_3 \)'s state is \( C_3 = 0 \), HOLDER3 = \( p_4 \), Q3 empty
- \( p_4 \)'s state is \( C_4 = 0 \), HOLDER4 = \( p_4 \), Q4 empty, holding token
- \( p_5 \)'s state is \( C_5 = 0 \), HOLDER5 = \( p_4 \), Q5 empty
- \( p_6 \)'s state is \( C_6 = 0 \), HOLDER6 = \( p_5 \), Q6 empty

\( p_1 \) sends \( Q(1) \) to \( p_3 \); \( p_1 \)'s state is \( C_1 = 1 \), HOLDER2 = \( p_3 \), Q1 = \( Q(1) \).

\( p_5 \) sends \( Q(5) \) to \( p_4 \); \( p_5 \)'s state is \( C_5 = 1 \), HOLDER5 = \( p_4 \), Q5 = \( Q(5) \).

\( p_3 \) receives \( Q(1) \) from \( p_1 \); \( p_3 \)'s state is \( C_3 = 0 \), HOLDER3 = \( p_4 \), Q3 empty.

\( p_3 \) sends \( Q(3) \) to \( p_4 \); \( p_3 \)'s state is \( C_3 = 1 \), HOLDER3 = \( p_4 \), Q3 empty.

\( p_4 \) receives \( Q(5) \) from \( p_5 \); \( p_4 \)'s state is \( C_4 = 0 \), HOLDER4 = \( p_4 \), Q4 empty, holding token.

\( p_4 \) sends token to \( p_5 \); \( p_4 \)'s state is \( C_4 = 1 \), HOLDER4 = \( p_5 \), Q4 empty.

\( p_4 \) receives \( Q(3) \) from \( p_3 \); \( p_4 \)'s state is \( C_4 = 1 \), HOLDER4 = \( p_5 \), Q4 empty.

\( p_4 \) sends \( Q(4) \) to \( p_5 \); \( p_4 \)'s state is \( C_4 = 2 \), HOLDER4 = \( p_5 \), Q4 = \( Q(3) \).

\( p_5 \) receives token from \( p_4 \); \( p_5 \)'s state is \( C_5 = 1 \), HOLDER5 = \( p_4 \), Q5 = \( Q(5) \).

\( p_5 \) resets state to \( C_5 = 1 \), HOLDER5 = \( p_4 \), Q5 empty, holding token.

**\( p_5 \) enters the critical section**

**\( p_5 \) leaves the critical section**

\( p_5 \) receives \( Q(4) \) from \( p_4 \); \( p_5 \)'s state is \( C_5 = 1 \), HOLDER5 = \( p_4 \), Q5 empty, holding token.

\( p_5 \) sends token to \( p_4 \); \( p_5 \)'s state is \( C_5 = 2 \), HOLDER5 = \( p_4 \), Q5 empty.

\( p_3 \) receives \( Q(3) \) from \( p_3 \); \( p_3 \)'s state is \( C_3 = 2 \), HOLDER3 = \( p_4 \), Q3 = \( Q(1) \).

\( p_4 \) receives \( Q(3) \) from \( p_3 \); \( p_4 \)'s state is \( C_4 = 2 \), HOLDER4 = \( p_5 \), Q4 = \( Q(3) \).

\( p_4 \)'s state is \( C_4 = 3 \), HOLDER4 = \( p_5 \), Q4 = \( Q(3) \)\( Q(3) \) [it sends nothing as it is waiting for a response]

\( p_4 \) receives token from \( p_5 \); \( p_4 \)'s state is \( C_4 = 3 \), HOLDER4 = \( p_5 \), Q4 = \( Q(3) \)\( Q(3) \), holding token.

\( p_4 \) sends token to \( p_3 \); \( p_4 \)'s state is \( C_4 = 3 \), HOLDER4 = \( p_3 \), Q4 = \( Q(3) \).

\( p_4 \) sends \( Q(4) \) to \( p_3 \); \( p_4 \)'s state is \( C_4 = 3 \), HOLDER4 = \( p_3 \), Q4 = \( Q(3) \).

\( p_3 \) receives token from \( p_4 \); \( p_3 \)'s state is \( C_3 = 2 \), HOLDER3 = \( p_4 \), Q3 = \( Q(1) \)\( Q(3) \), holding token.

\( p_3 \) sends token to \( p_1 \); \( p_3 \)'s state is \( C_3 = 3 \), HOLDER3 = \( p_1 \), Q3 = \( Q(3) \).

\( p_3 \) sends \( Q(3) \) to \( p_1 \); \( p_3 \)'s state is \( C_3 = 4 \), HOLDER3 = \( p_1 \), Q3 = \( Q(3) \).

\( p_1 \) receives token from \( p_3 \); \( p_1 \)'s state is \( C_1 = 1 \), HOLDER1 = \( p_3 \), Q1 = \( Q(1) \), holding token.

\( p_1 \) resets state to \( C_1 = 1 \), HOLDER1 = \( p_3 \), Q1 empty, holding token.

**\( p_1 \) enters the critical section**

**\( p_1 \) leaves the critical section**

\( p_1 \) receives \( Q(3) \) from \( p_4 \); \( p_1 \)'s state is \( C_1 = 1 \), HOLDER1 = \( p_3 \), Q1 empty, holding token.

\( p_1 \) sends token to \( p_3 \); \( p_1 \)'s state is \( C_1 = 2 \), HOLDER1 = \( p_3 \), Q1 empty.

\( p_3 \) receives token from \( p_1 \); \( p_3 \)'s state is \( C_3 = 4 \), HOLDER3 = \( p_1 \), Q3 = \( Q(3) \), holding token.

\( p_3 \) receives \( Q(4) \) from \( p_4 \); \( p_3 \)'s state is \( C_3 = 4 \), HOLDER3 = \( p_1 \), Q3 = \( Q(3) \)\( Q(4) \).

\( p_3 \) resets state to \( C_3 = 4 \), HOLDER3 = \( p_1 \), Q3 = \( Q(4) \).

**\( p_3 \) enters the critical section**

**\( p_3 \) leaves the critical section**

\( p_3 \) sends token to \( p_4 \); \( p_3 \)'s state is \( C_3 = 5 \), HOLDER3 = \( p_4 \), Q3 empty, holding token.

\( p_4 \) receives token from \( p_3 \); \( p_4 \)'s state is \( C_4 = 3 \), HOLDER4 = \( p_3 \), Q4 = \( Q(3) \).

\( p_4 \) sends token to \( p_3 \); \( p_4 \)'s state is \( C_4 = 4 \), HOLDER4 = \( p_3 \), Q4 empty.

\( p_3 \) receives token from \( p_4 \); \( p_3 \)'s state is \( C_3 = 5 \), HOLDER5 = \( p_4 \), Q5 empty.
Lamport-Shostak-Pease Algorithm

Introduction

This is a recursive protocol. It requires $3m+1$ processors where at most $m$ are faulty. It consists of two protocols, the base protocol and the inductive protocol. To run it, determine $m$ from $n$ and invoke OM($m$).

Notation

- $n$ processes
- $p_i$ process

Protocol OM(0)

1. The source process sends its value to all processes.
2. Each process uses the value it receives from the source. If it receives no value, it uses a value of 0.

Protocol OM($m$), $m > 0$

1. The source process sends its value to all processes.
2. Let $v_i$ be the value process $p_i$ receives from the source. (If it receives no value, then take $v_i = 0$.) Process $p_i$ initiates OM($m-1$) with itself as the source and the other $n-2$ processes as the recipients.
3. Process $p_i$ uses the value majority($v_1, \ldots, v_{n-1}$), where $v_i$ is the value received in step 2 from the source process and the others are the values received from OM($m-1$).

Example

There are four processes, $p_0$ through $p_3$. They wish to agree on a value 0 or 1. Let $p_0$ be the initiator, and it has value 1. Assume all processes are non-faulty.

$p_0$ invokes OM(1)

- $p_0$ sends 1 to $p_1$, $p_2$, and $p_3$.
- $p_1$ receives 1 from $p_0$ and invokes OM(0).
  - $p_1$ sends 1 to $p_2$ and $p_3$.
  - $p_2$ receives value 1.
  - $p_3$ receives value 1.
- $p_2$ receives 1 from $p_0$ and invokes OM(0).
  - $p_2$ sends 1 to $p_1$ and $p_3$.
  - $p_1$ receives value 1.
  - $p_3$ receives value 1.
- $p_3$ receives 1 from $p_0$ and invokes OM(0).
  - $p_3$ sends 1 to $p_1$ and $p_2$.
  - $p_1$ receives value 1.
  - $p_2$ receives value 1.

$p_1$ computes majority (1, 1, 1) and takes the value at the source to be 1.
$p_2$ computes majority (1, 1, 1) and takes the value at the source to be 1.
$p_3$ computes majority (1, 1, 1) and takes the value at the source to be 1.
Now assume p2 is faulty and will send a bogus value.
p0 invokes OM(1)
  p0 sends 1 to p1, p2, and p3.
  p1 receives 1 from p0 and invokes OM(0).
    p1 sends 1 to p2 and p3.
    p2 receives value 1.
    p3 receives value 1.
  p2 receives 1 from p0 and invokes OM(0).
    p2 sends 0 to p1 and p3.
    p1 receives value 0.
    p3 receives value 0.
  p3 receives 1 from p0 and invokes OM(0).
    p3 sends 1 to p1 and p2.
    p1 receives value 1.
    p2 receives value 1.

p1 computes majority (1, 0, 1) and takes the value at the source to be 1.
p2 computes majority (1, 1, 1) and takes the value at the source to be 1.
p3 computes majority (1, 0, 1) and takes the value at the source to be 1.

Now assume p0 is faulty and will send a random value.
p0 invokes OM(1)
  p0 sends 1 to p1 and 0 to p2 and p3.
  p1 receives 1 from p0 and invokes OM(0).
    p1 sends 1 to p2 and p3.
    p2 receives value 1.
    p3 receives value 1.
  p2 receives 0 from p0 and invokes OM(0).
    p2 sends 0 to p1 and p3.
    p1 receives value 0.
    p3 receives value 0.
  p3 receives 0 from p0 and invokes OM(0).
    p3 sends 0 to p1 and p2.
    p1 receives value 0.
    p2 receives value 0.

p1 computes majority (1, 0, 0) and takes the value at the source to be 0.
p2 computes majority (1, 0, 0) and takes the value at the source to be 0.
p3 computes majority (1, 0, 0) and takes the value at the source to be 0.
In this case agreement is reached, but as the source is faulty the result is not valid.
Fault-Tolerant Clock Synchronization

Introduction
The goal is to synchronize the time of clocks on different systems. The protocol includes both faulty and non-faulty clocks. The assumptions are that initially all clocks are synchronized to within some small value $\delta$, that non-faulty clocks run at the correct rate (that is, one tick per second), and a nonfaulty process can read a non-faulty clock with an error of at most $\varepsilon$. In what follows, we shall assume $\varepsilon = 0$.

Notation
- $n$ processes
- $p_i$ process

Interactive Convergence Protocol
This assumes that no two non-faulty clocks differ by more than $\delta$. All processes execute this protocol simultaneously.
1. $p_i$ obtains the value of the other processes’ clocks (for example, by using the OM($m$) protocol). Call these values $v_1, \ldots, v_n$.
2. For all $j < n$, if $|v_j - v_i| > \delta$, set $v_j' = v_i$. Otherwise, $v_j' = v_j$.
3. Set $p_i$’s clock to $(\sum_j v_j')/n$.

Example
Suppose $p_0, p_1, p_2,$ and $p_3$ wish to synchronize their clocks. Take $\delta = 10$, $C_0 = 2$, $C_1 = 5$, $C_2 = 8$, and $C_3 = 10$. Then: after this protocol is used, all the clocks are set to $(2 + 5 + 8 + 10)/4 = 25/4 = 6$.
Now suppose $p_2$’s clock is faulty and drifts to $C_3 = 25$. Then:
- $C_0 = (2 + 5 + 8 + 2)/4 = 17/4 = 4$  
- $C_1 = (2 + 5 + 8 + 5)/4 = 20/4 = 5$  
- $C_2 = (2 + 5 + 8 + 8)/4 = 23/4 = 6$
After the next round, assuming $p_3$ reports any value $\delta$ away from $C_0$, $C_1$, and $C_2$:
- $C_0 = (4 + 5 + 6 + 4)/4 = 19/4 = 5$  
- $C_1 = (4 + 5 + 6 + 5)/4 = 20/4 = 5$  
- $C_2 = (4 + 5 + 6 + 6)/4 = 21/4 = 5$
Now assume $C_3$ is a two-faced clock. The danger is that $p_3$ will report a value within $\delta$ of $C_1$ to $p_1$, and not within $\delta$ of $C_0$ and $C_2$. So, begin with the same values as above, except that $p_3$ reports $C_3 = 1$ to $p_1$ and $C_3 = 25$ to $p_0$ and $p_2$:
- $C_0 = (2 + 5 + 8 + 2)/4 = 17/4 = 4$  
- $C_1 = (2 + 5 + 8 + 1)/4 = 16/4 = 4$  
- $C_2 = (2 + 5 + 8 + 8)/4 = 23/4 = 6$
At the next round, $p_3$ reports $C_3 = 15$ to $p_2$ and $C_3 = 0$ to $p_0$ and $p_1$.
- $C_0 = (4 + 4 + 6 + 0)/4 = 14/4 = 4$  
- $C_1 = (4 + 4 + 6 + 0)/4 = 14/4 = 4$  
- $C_3 = (4 + 4 + 6 + 15)/4 = 29/4 = 7$
By continuing in this fashion, $p_3$ can prevent the value of the clocks of the non-faulty processors from converging.
Interactive Consistency Protocol
This assumes that no two non-faulty clocks differ by more than $\delta$. All processes execute this protocol simultaneously.

1. $p_i$ obtains the value of the other processes’ clocks (for example, by using the OM($m$) protocol). Call these values $v_1, \ldots, v_n$.
2. Set $p_i$’s clock to the median of $v_1, \ldots, v_n$.

Example
Suppose $p_0$, $p_1$, $p_2$, and $p_3$ wish to synchronize their clocks. Take $\delta = 10$, $C_0 = 2$, $C_1 = 5$, $C_2 = 8$, and $C_3 = 10$. Then: after this protocol is used, all the clocks are set to $\text{median}(2, 5, 8, 10) = (5 + 8)/2 = 6$.

Now suppose $p_3$’s clock is faulty and drifts to $C_3 = 25$. Then:
- $C_0 = \text{median}(2, 5, 8, 25) = (5 + 8)/2 = 6$
- $C_1 = \text{median}(2, 5, 8, 25) = (5 + 8)/2 = 6$
- $C_2 = \text{median}(2, 5, 8, 25) = (5 + 8)/2 = 6$

Now assume $C_3$ is a two-faced clock. Begin with the same values as above, except that $p_3$ reports $C_3 = 1$ to $p_1$ and $C_3 = 25$ to $p_0$ and $p_2$. All apply an agreement protocol:

$p_3$ invokes OM(1)
- $p_3$ sends 1 to $p_1$ and 25 to $p_0$ and $p_2$.
- $p_0$ receives 25 from $p_3$ and invokes OM(0).
  - $p_0$ sends 25 to $p_1$ and $p_2$.
  - $p_1$ receives value 25.
  - $p_2$ receives value 25.
- $p_1$ receives 1 from $p_3$ and invokes OM(0).
  - $p_1$ sends 1 to $p_0$ and $p_2$.
  - $p_0$ receives value 1.
  - $p_2$ receives value 1.
- $p_2$ receives 25 from $p_3$ and invokes OM(0).
  - $p_2$ sends 25 to $p_0$ and $p_1$.
  - $p_0$ receives value 25.
  - $p_1$ receives value 25.
$p_0$ computes majority (25, 1, 25) and takes the value at the source to be 25.
- $C_0 = \text{median}(2, 5, 8, 25) = (5 + 8)/2 = 6$
- $C_1 = \text{median}(2, 5, 8, 25) = (5 + 8)/2 = 6$
- $C_2 = \text{median}(2, 5, 8, 25) = (5 + 8)/2 = 6$

Notice that all arrive at the same value.