Today

☐ A methodology for making fault-tolerant distributed systems.

Building a system that masks failures

Coping with *concurrency* is difficult because of the exponential increase in the number of behaviors.

Coping with *failures* has all the problems of concurrency combined with the nondeterministic occurrence of failures.

... it's an area that can benefit from a *methodology*.
The State Machine approach: Basic idea

1. Design your system so that it consists of clients that invoke commands on *deterministic state machines*.
   - The state of a state machine depends only on its initial state and the sequence of (deterministic) commands it has been given.
   - A state machine can be a client of another state machine.
   - Clients don't have to be deterministic.
The State Machine approach: Basic idea (continued)

2. Replicate the state machines, and have the client obtain its results through a \textit{voter}.

\begin{itemize}
  \item All nonfaulty state machines, being deterministic, will give the same response to a command.
\end{itemize}
Some examples of state machines

Since this is a methodology, we don't specify how state machine command invocation is done.

```
state machine memory
  word store [0..n]
  command read (0..n loc)
    send store[loc] to client
  end read
  command write (0..n loc, word value)
    store[loc] = value
  end write
end memory
```
Some examples of state machines

state machine mutex
  var client_id user = ∅
  list(client_id) waiting = ∅
  command acquire
    if (user == ∅)
      send OK to client
      user = client
    else waiting = waiting ° client
  end acquire
  command release
    if (waiting == ∅) user = ∅
    else
      user = head(waiting)
      send OK to user
      waiting = tail(waiting)
  end release
end mutex
Some examples of state machines

The state of a state machine can only depend on its initial state and the sequence of commands it executes. It can't depend on other system activity like the time.

**process monitor**

  **while (true)**
  
  ```
  val = sensor()
  \langle pc.adjust, val \rangle
  delay D
  end monitor
  ```

**state machine pc**

  **var real q**

  **command adjust (real sensor_val)**
  
  ```
  q = F(q, sensor_val)
  send q to actuator
  end adjust
  end pc
  ```
Replicating state machines
Replicating state machines

All nonfaulty state machines always have the same state.
Replicating state machines: Voting

Failstop failures: \( n > t \)
Arbitrary (aka Byzantine): \( n > 2t \)
More detailed example
More detailed example
More detailed example
More detailed example
Replicating state machines: Agreement

All nonfaulty state machines always have the same state.
Agreement: Reliable Broadcast

There is a transmitter $p_1$ and a set of receivers $p_2, p_3, p_n$. $p_1$ has an initial value $v$ and each $p_i$ computes a decision value $d_i$.

- If $p_1$ is nonfaulty, then for all nonfaulty $p_i$: $d_i = v$.
- If $p_1$ is faulty, then for all nonfaulty $p_i$ and $p_j$: $d_i = d_j$.

... 

- (Uniformity) For any two processes $p_i$ and $p_j$ that decide, $d_i = d_j$. 

Replicating state machines: Order

All nonfaulty state machines always have the same state.
Implementing order

Many ways to do this; will briefly touch on three.

Use same basic idea:

- Assign each request $r$ a unique identifier $r.UID$ from a domain that is totally ordered.
- A request is stable at $sm_i$ when no request can be delivered that is
  - from a correct client
  - with a lower unique identifier.

... A state machine replica takes the stable request with the smallest unique identifier.
Using Lamport clocks

... a deterministic and nondense variation of causal delivery.

Assume failstop and channels that are FIFO for messages and failure detection.

☐ $C(e)$ is the Lamport clock for the event in which the client $p_i$ (reliably) broadcasts $r$; $r.UID$ is $\langle i, C(e) \rangle$.

☐ Define $\langle a, b \rangle > \langle c, d \rangle \equiv (b > d) \lor ((b = d) \land (a > c))$. 

☐ Request $r$ is stable at $sm_i$ when $sm_i$ has received a request $r'$ from each client with $UID(r') \geq UID(r)$.

Using Lamport clocks (continued)

(1:1) stable at \( sm_1, sm_2 \) and \( sm_3 \)
Using real-time clocks

- $C(e)$ is the real-time clock for the event in which the client $p_i$ (reliably) broadcasts request $r$; $r.UID$ is $\langle i, C(e) \rangle$.

- Requires
  - a client does not make more than one request during any clock tick
  - If $p_i$ sends $r$, then all state machines will receive $r$ by $r.UID + \Delta$ according to the state machine's local clock.

  … If it is desired to have causal delivery order, then requires
    - all of the processors' clocks synchronized within $\delta$ seconds, and
    - it takes more than $\delta$ seconds to send a message from one processor to another.

  Otherwise it would be possible for $r' \rightarrow r$ and $r'.UID > r.UID$. 
Using real-time clocks (continued)

Request $r$ is stable at $sm_i$ when:

- for each nonfailed client, $sm_i$ has received a request $r'$ where $\text{UID}(r') \geq \text{UID}(r)$, or
- the real time clock at $sm_i$ is greater than $r.\text{UID} + \Delta$.

Using replica-generated UIDs

- A two phase protocol:
  1. State machine replicas propose candidate UIDs.
  2. One candidate UID is selected.

$cuid(sm_i, r)$ is the candidate UID proposed by $sm_i$

- $sm_i$ has seen $r$ once it has proposed $cuid(sm_i, r)$
- $sm_i$ has accepted $r$ once $sm_i$ knows r.UID.
Using replica-generated UIDs (continued)

Constrain \( cuid \) as follows:

1. \( uid(r) \geq cuid(sm_i, r) \).
2. If request \( r' \) is seen by \( sm_i \) after \( r \) has been accepted by \( sm_i \), then \( cuid(sm_i, r') > r.UID \).

Request \( r \) is stable at \( sm_i \) when

- it has been accepted by \( sm_i \) and
- there is no request \( r' \):
  - that has been seen by \( sm_i \), and
  - that has not been accepted by \( sm_i \), and
  - for which \( cuid(sm_i, r') \leq uid(r) \).

... if there were such a \( r' \), then \( r'.UID \) could be less than \( r.UID \).
Using replica-generated UIDs (continued)

- Each $sm_i$ maintains:
  - $SEEN_i$: the largest $cuid(sm_i, r)$ issued
  - $ACCEPT_i$: the largest $r$.UID assigned by $sm_i$.

Upon receipt of a new request $r$,

$$cuid(sm_i, r) = \max(SEEN_i, ACCEPT_i) + 1$$

Upon receiving the all candidate uids, $r$.UID is set to the largest $cuid(sm_i, r)$, using processor ID to break ties (as we did with logical clocks).
Using replica-generated UIDs (continued)

\[
\begin{align*}
\text{seen}_1 &= 0 \\
\text{accept}_1 &= 0 \\
\text{seen}_2 &= 0 \\
\text{accept}_2 &= 0 \\
\text{seen}_3 &= 0 \\
\text{accept}_3 &= 0
\end{align*}
\]
Using replica-generated UIDs (continued)

\[ \begin{align*}
\text{seen}_0 &= 1 \\
\text{accept}_0 &= 0
\end{align*} \]

\[ \begin{align*}
\text{seen}_1 &= 1 \\
\text{accept}_1 &= 0
\end{align*} \]

\[ \begin{align*}
\text{seen}_2 &= 1 \\
\text{accept}_2 &= 0
\end{align*} \]
Using replica-generated UIDs (continued)

\[
\begin{align*}
&\text{seen}_0 = 2 \\
&\text{accept}_0 = 0 \\
&\text{seen}_1 = 2 \\
&\text{accept}_1 = 0 \\
&\text{seen}_2 = 2 \\
&\text{accept}_2 = 0
\end{align*}
\]
Using replica-generated UIDs (continued)

seen<sub>0</sub> = 3
accept<sub>0</sub> = 0

seen<sub>1</sub> = 3
accept<sub>1</sub> = 0

seen<sub>2</sub> = 3
accept<sub>2</sub> = 0
Using replica-generated UIDs (continued)

- \( m_1:3 \) \( m_2:3 \) \( m_3:3 \)
  - \( seen_0 = 3 \)
  - \( accept_0 = 3 \)

- \( m_2:3 \) \( m_3:3 \) \( m_1:3 \)
  - \( seen_1 = 3 \)
  - \( accept_1 = 3 \)

- \( m_3:3 \) \( m_1:3 \) \( m_3:3 \) \( m_4:4 \)?
  - \( seen_2 = 4 \)
  - \( accept_2 = 3 \)
Tolerating faulty clients

Tolerating faulty clients can be approached using detection or masking.

- Masking: replicate the clients so they fail independently
  - State machines will need to vote on sets of commands
  - Nondeterminism can make such voting harder.
Tolerating faulty clients (continued)
Tolerating faulty clients (continued)
Tolerating faulty clients (continued)

Detection: usually done by *defensive programming* of the state machine.
Tolerating faulty clients (continued)

```plaintext
state machine mutex
    var client_id user = ∅
    list(client_id) waiting = ∅

command acquire
    if (user == ∅)
        send OK to client
        user = client
    else waiting = waiting ° client
    end acquire

command release
    if (waiting == ∅) user = ∅
    else
        user = head(waiting)
        send OK to user
        waiting = tail(waiting)
    end release

end mutex
```

What if the client calling \textit{release} had not previously acquired mutex?
Tolerating faulty clients (continued)

```plaintext
state machine mutex
    var client_id user = ∅
    list(client_id) waiting = ∅
    command acquire
        if (user == ∅)
            send OK to client
            user = client
        else waiting = waiting ° client
    end acquire
    command release
        if (user ≠ client) return;
        if (waiting == ∅) user = ∅
        else
            user = head(waiting)
            send OK to user
            waiting = tail(waiting)
        end release
    end mutex
```

What if the client user never calls release?
Tolerating faulty clients (continued)

```
command acquire
  if (user == ∅)
    send OK to client
    user = client
    time_granted = TIME
    schedule (mutex.timeout, time_granted) for +B
  else waiting = waiting ∪ client
end acquire

command release
  if (user ≠ client) return;
  time_granted = null
  if (waiting == ∅) user = ∅
  else
    user = head(waiting)
    time_granted = TIME
    schedule (mutex.timeout, time_granted) for +B
    send OK to user
    waiting = tail(waiting)
end release

command timeout (int when_granted)
  if (when_granted ≠ time_granted) return
  if (waiting == ∅) user = ∅
  else
    user = head(waiting)
    time_granted = TIME
    send OK to user
    waiting = tail(waiting)
    schedule (mutex.timeout, time_granted) for +B
end timeout
```
Reconfiguration

The events of adding or removing state machine replicas can also generate commands to the state machines.

Example: Suppose that we wish to have some output done by a single state machine (effectively pulling the voter into the state machine ensemble itself)

Group membership

Coordinator-cohort
Reconfiguration (continued)