6.852: Distributed Algorithms
Fall, 2009

Class 10
Today’s plan

• Simulating synchronous algorithms in asynchronous networks
• Synchronizers
• Lower bound for global synchronization
• Reading: Chapter 16
• Next:
  – Logical time
  – Reading: Chapter 18, [Lamport time, clocks…], [Mattern]
Minimum spanning tree, revisited

- In GHS, complications arise because different parts of the network can be at very different levels at the same time.

- Alternative, more synchronized approach:
  - Keep levels of nearby nodes close, by restricting the asynchrony.
  - Each process uses a level variable to keep track of the level of its current component (according to its local knowledge).
  - Process at level $k$ delays all “interesting” processing until it hears that all its neighbors have reached level $\geq k$.
    - Looks (to each process) like global synchronization, but easier to achieve.
    - Each node inform its neighbors whenever it changes level.

- Resulting algorithm is simpler than GHS.

- Complexity:
  - Time: $O(n \log n)$, like GHS.
  - Messages: $O(|E| \log n)$, somewhat worse than GHS.
Strategy for designing asynchronous distributed algorithms

- Assume undirected graph $G = (V,E)$.
- Design a synchronous algorithm for $G$, transform it into an asynchronous algorithm using local synchronization.
- Synchronize at every round (not every “level” as above).
- Method works only for non-fault-tolerant algorithms.
  - In fact, no general transformation can work for fault-tolerant algorithms.
  - E.g., ordinary stopping agreement is solvable in synchronous networks, but unsolvable in asynchronous networks [FLP].
- Present a general strategy, some special implementations.
  - Describe in terms of sub-algorithms, modeled as abstract services.
  - [Raynal book], [Awerbuch papers]
- Then a lower bound on the time for global synchronization.
  - Larger than upper bounds for local synchronization.
Synchronous model, reformulated in terms of automata

- Global synchronizer automaton
- User process automata:
  - Processes of an algorithm that uses the synchronizer.
  - May have other inputs/outputs, for interacting with other programs.
- Interactions between user process i and synchronizer:
  - user-send($T,r)_i$
    - $T$ = set of (message, destination) pairs, destinations are neighbors of i.
    - $T$ = empty set $\emptyset$, if no messages sent by i at round $r$.
    - $r$ = round number
  - user-rcv($T,r)_i$
    - $T$ = set of (message, source) pairs, source a neighbor of i.
    - $r$ = round number
Behavior of GlobSynch

• Manages global synchronization of rounds:
  – Users send packages of all their round 1 messages, using `user-send(T,r)` actions.
  – GlobSynch waits for all round 1 messages, sorts them, then delivers to users, using `user-rcv(T,r)` actions.
  – Users send round 2 messages, etc.

• Not exactly the synchronous model:
  – GlobSynch can receive round 2 messages from i before it finishes delivering all the round 1 messages.
  – But it doesn’t do anything with these until it’s finished round 1 deliveries.
  – So, essentially the same.

• GlobSynch synchronizes globally between each pair of rounds.
Requirements on each $U_i$

- **Well-formed:**
  - $U_i$ sends the right kinds of messages, in the right order, at the right times.

- **Liveness:**
  - After receiving the messages for any round $r$, $U_i$ eventually submits messages for round $r+1$.

- **See code for GlobSynch in [book, p. 534].**
  - State consists of:
    - A *tray* of messages for each (destination, round).
    - Some Boolean flags to keep track of which sends and rcvs have happened.
  - Transitions obvious.
  - Liveness expressed by tasks, one for each (destination, round).
Synchronizers
The Synchronizer Problem

- Design an automaton $A$ that “implements” GlobSynch in the sense that it “looks the same” to each $U_i$:
  - Has the right interface.
  - Exhibits the right behavior:
    - $\forall$ fair execution $\alpha$ of the $U_i$s and $A$,
    - $\exists$ fair execution $\alpha'$ of the $U_i$s and GlobSynch, such that
      - $\forall i$, $\alpha$ is indistinguishable by $U_i$ from $\alpha'$,
        $\alpha \sim_{U_i} \alpha'$.
- $A$ “behaves like” GlobSynch, as far as any individual $U_i$ can tell.
- Allows global reordering of events at different $U_i$. 
Local Synchronizer, LocSynch

- Enforces local synchronization rather than global, still looks the same locally.

- Only one difference from GlobSynch:
  - Precondition for $\text{usr-rcv}(T,r)_i$.
  - Now, to deliver round $r$ messages to user $i$, check only that $i$’s neighbors have sent round $r$ messages.
  - Don’t wait for all nodes to get this far.

- **Lemma 1**: For every fair execution $\alpha$ of the $U_i$s and LocSynch, there is a fair execution $\alpha'$ of the $U_i$s and GlobSynch, such that for each $U_i$, $\alpha \sim U_i \alpha'$.

- **Proof**:
  - Can’t use a simulation relation, since global order of external events need not be the same, and simulation relations preserve external order.
  - So consider partial order of events and dependencies:
Proof sketch for Lemma 1

- Consider partial order of events and dependencies:
  - Each $U_i$ event depends on previous $U_i$ events.
  - $user-rcv(*,r)_i$ event depends on $user-send(*,r)_j$ for every neighbor $j$ of $i$.
  - Take transitive closure.

- **Claim:** If you start with a (fair) execution of LocSynch system and reorder the events while preserving these dependencies, the result is still a (fair) execution of the LocSynch system.

- So, obtain $\alpha'$ by reordering the events of $\alpha$ so that:
  - These dependencies are preserved, and
  - Events associated with any round $r$ precede those of round $r+1$.
- Can do this because round $r+1$ events never depend on round $r$ events.
- This reordering preserves the view of each $U_i$.
- Also, yields the extra $user-rcv$ precondition needed by GlobSynch.
Trivial distributed algorithm to implement LocSynch

- Processes, point-to-point channels.
- SimpleSynch algorithm, process $i$:
  - After $\text{user-send}(T,r)_i$, send message to each neighbor $j$ containing round number $r$ and any basic algorithm messages $i$ has for $j$.
  - Send $(\emptyset,r)$ message if $i$ has no basic algorithm messages for $j$.
  - Wait to receive round $r$ messages from all neighbors.
  - Output $\text{user-rcv()}$.

- **Lemma 2:**
  - For every fair execution $\alpha$ of $U_is$ and SimpleSynch, there is a fair execution $\alpha'$ of $U_is$ and LocSynch, such that for each $U_i$, $\alpha \sim U_i \alpha'$.

- Here, indistinguishable by all the $U_is$ together---preserves external order.
SimpleSynch, cont’d

• **Proof of Lemma 2:**
  – No reordering needed, preserves order of external events.
  – Could use simulation relation.

• **Corollary:** For every fair execution $\alpha$ of $U_i$s and SimpleSynch, there is a fair execution $\alpha'$ of $U_i$s and GlobSynch, such that for each $U_i$, $\alpha \sim_{U_i} \alpha'$.

• **Proof:** Combine Lemmas 1 and 2.

• **Complexity:**
  – Messages: $\leq 2 |E|$ per simulated round.
  – Time:
    • Assume user always sends ASAP.
    • $l$, upper bound on task time for each task of each process.
    • $d$, upper bound on time for first message in channel to be delivered
    • Then $r$ rounds completed within time $r (d + O(l))$. 
Reducing the communication

- General Safe Synchronizer strategy [Awerbuch].
- If there’s no message \( U_i \rightarrow U_j \) at round \( r \) of underlying synchronous algorithm, try to avoid sending such messages in the simulating asynchronous algorithm.
- Can’t just omit them, since each process must determine, for each round \( r \), when it has received all of its round \( r \) messages.
- **Approach**: Separate the functions of:
  - Sending the actual messages, and
  - Determining when the round is over.
  - Algorithm decomposes into:
    - Front Ends + channels + SafeSynch
      - For the actual messages
      - For deciding when finished
Safe Synchronizers

• FE:
  – Sends, receives actual messages for each round r.
  – Sends acks for received messages.
  – Waits to receive acks for its own messages.

• Notes:
  – Sends messages only for actual messages of the underlying algorithm, no dummies.
  – Acks double the messages, but can still be a win.

• FE, cont’d:
  – When FE receives acks for all its round r messages, it’s safe: it knows that all its messages have been received by its neighbors.
  – Then sends OK for round r to SafeSynch.
  – Before responding to user, must know that it has received all its neighbors’ messages for round r.
  – Suffices to know that all its neighbors are safe, that is, that they know that their messages have been received.

• SafeSynch:
  – Tells each FE when its neighbors are safe!
  – After it has received OK from i and all its neighbors, sends GO to i.
Correctness of SafeSynch

• **Lemma 3:** For every fair execution $\alpha$ of SafeSynch system, there is a fair execution $\alpha'$ of LocSynch system, such that for each $U_i$, $\alpha \sim_{U_i} \alpha'$.
  
  (Actually, indistinguishable to all the $U_i$s together.)

• **Corollary:** For every fair execution $\alpha$ of SafeSynch system, there is a fair execution $\alpha'$ of GlobSynch system, such that for each $U_i$, $\alpha \sim_{U_i} \alpha'$.

• Must still implement SafeSynch with a distributed algorithm…

• We now give three SafeSynch implementations, Synchronizers $A$, $B$, and $\Gamma$ [Awerbuch].

• All implement SafeSynch, in the sense that the resulting systems are indistinguishable to each $U_i$ (in fact, to all the $U_i$s together).
SafeSynch Implementations

- **SafeSynch’s job:** After receiving **OK** for round \( r \) from \( i \) and all its neighbors, send **GO** for round \( r \) to \( i \).

- **Synchronizer \( A \):**
  - When process \( i \) receives **OK** \( i \), sends to neighbors.
  - When process \( i \) hears that it and all its neighbors have received OKs, outputs **GO** \( i \).

- **Obviously implements SafeSynch.**

- **Complexity:** To emulate \( r \) rounds:
  - **Messages:** \( \leq 2m + 2r|E| \), if synch alg sends \( m \) actual messages in \( r \) rounds.
  - **Time:** \( \leq r(3d + O(1)) \)
Comparisons

• To emulate \( r \) rounds:
  – SafeSynch system with Synchronizer \( \Lambda \)
    • Messages: \( 2m + 2 \, r \, |E| \)
    • Time: \( r \, (3d + O(l)) \)
  – Simple Synch
    • Messages: \( 2 \, r \, |E| \)
    • Time: \( r \, (d + O(l)) \)

• So Synchronizer \( \Lambda \) hasn’t improved anything.

• Next, Synchronizer \( \beta \), with lower message complexity, higher time complexity.

• Then Synchronizer \( \Gamma \), does well in terms of both messages and time, in an important subclass of networks (those with a “cluster” structure).
Synchronizer B

- Assumes rooted spanning tree of graph, height h.
- Algorithm:
  - All processes convergecast OK to root, using spanning tree edges.
  - Root then bcasts permission to GO, again using the spanning tree.
- Obviously implements SafeSynch (overkill).
- Complexity: To emulate r rounds, in which synch algorithm sends m messages:
  - Messages: $2m + 2rn$
  - Beats A: $2m + 2r|E|$
  - Time: $\leq r(2d + O(l) + 2h(d + O(l)))$

Messages and acks by FEs

Messages within B

FEs

B, convergecast and broadcast
Synchronizer $\Gamma$

- Hybrid of A and B.
- In “clustered” (almost partitionable) graphs, can get performance advantages of both:
  - Time like A, communication like B.
- Assume spanning forest of rooted trees, each tree spanning a “cluster” of nodes.
- Example:
  - Clusters = triangles
  - All edges between adjacent triangles.
  - Spanning forest:
- Use B within each cluster, A among clusters.
Decomposition of $\Gamma$

- **ClusterSynch:**
  - After receiving OKs from everyone in the cluster, sends cluster-OK to ForestSynch.
  - After receiving cluster-GO from ForestSynch, sends GO to everyone in the cluster.
  - Similar to B.

- **ForestSynch:**
  - Essentially, a safe synchronizer for the “Cluster Graph” $G'$:
    - Nodes of $G'$ are the clusters.
    - Edge between two clusters iff they contain nodes that are adjacent in $G$.

- **Lemma:** $\Gamma$ Implements SafeSynch
- **Proof idea:**
  - Must show: If GO($r_i$) occurs, then there must be a previous OK($r_i$), and also previous OK($r_j$) for every neighbor $j$ of $i$. 
Γ Implements SafeSynch

- **Show:** If $GO(r)_i$ occurs, then there must be a previous $OK(r)_i$, and also previous $OK(r)_j$ for every neighbor $j$ of $i$.
- **Must be a previous $OK(r)_i$:**
  - $GO(r)_i$ preceded by $cluster-GO(r)$ for $i$’s cluster (ClusterSynch),
  - Which is preceded by $cluster-OK(r)$ for $i$’s cluster (ForestSynch),
  - Which is preceded by $OK(r)_i$ (ClusterSynch).
- **Must be previous $OK(r)_j$ for neighbor $j$ in the same cluster as $i$.**
  - $GO(r)_i$ preceded by $cluster-GO(r)$ for $i$’s cluster (ClusterSynch),
  - Which is preceded by $cluster-OK(r)$ for $i$’s cluster (ForestSynch),
  - Which is preceded by $OK(r)_j$ (ClusterSynch).
- **Must be previous $OK(r)_j$ for neighbor $j$ in a different cluster.**
  - Then the two clusters are neighboring clusters in the cluster graph $G'$, because $i$ and $j$ are neighbors in $G$.
  - $GO(r)_i$ preceded by $cluster-GO(r)$ for $i$’s cluster (ClusterSynch),
  - Which is preceded by $cluster-OK(r)$ for $j$’s cluster (ForestSynch),
  - Which is preceded by $OK(r)_j$ (ClusterSynch).
Implementing ClusterSynch and ForestSynch

• Still need distributed algorithms for these…

• ClusterSynch:
  – Use variant of Synchronizer B on cluster tree:
    • Convergecast OKs to root on the cluster tree,
    • root outputs cluster-OK, receives cluster-GO,
    • root broadcasts GO on the cluster tree.

• ForestSynch:
  – Clusters run Synchronizer A.
    • But clusters can’t actually run anything…
    • So cluster roots run A.
    • Simulate communication channels between neighboring clusters by indirect communication paths between the roots.
    • These paths must exist: Run through the trees and across edges that join the clusters.

• cluster-OK and cluster-GO are internal actions of the cluster root processes.
Putting the pieces together

• In $\Gamma$, real process $i$ emulates $\text{FrontEnd}_i$, process $i$ in ClusterSynch algorithm, and process $i$ in ForestSynch algorithm.
  – Composition of three automata.
• Real channel $C_{ij}$ emulates channel from $\text{FrontEnd}_i$ to $\text{FrontEnd}_j$, channel from $i$ to $j$ in ClusterSynch algorithm, and channel from $i$ to $j$ in ForestSynch algorithm.

• Orthogonal decompositions of $\Gamma$:
  – Physical: Nodes and channels.
  – Logical: FEs, ClusterSynch, and ForestSynch
  – Same system, 2 views.
  – Works because composition of automata is associative, commutative.

• Such decompositions are common for complex distributed algorithms:
  – Each node runs pieces of algorithms at several layers.

• Theorem 1: For every fair execution $\alpha$ of $\Gamma$ system (or $\Lambda$, or $\beta$), there is a fair execution $\alpha'$ of GlobSynch system, such that for each $U_i$, $\alpha \sim_{U_i} \alpha'$. 
Complexity of $\Gamma$

- Consider $r$ rounds, in which the synchronous algorithm sends $m$ messages.
- Let:
  - $h = \text{max height of a cluster tree}$
  - $e' = \text{total number of edges on shortest paths between roots of neighboring clusters}$.
- Messages: $2m + O(r(n + e'))$
- Time: $O(rh(d + l))$
- If $n + e' \ll |E|$, then $\Gamma$’s message complexity is much better than $\Lambda$’s.
- If $h \ll \text{height of spanning tree of entire network}$, then $\Gamma$’s time complexity is much better than $\ Beta$’s.
- Both of these are true for “nicely clustered” networks.
Comparison of Costs

- $r$ rounds
- $m$ messages sent by synchronous algorithm
- $d$, message delay
- Ignore local processing time $l$.
- $e' = \text{total length of paths between roots of neighboring clusters}$
- $h = \text{height of global spanning tree}$
- $h' = \text{max height of cluster tree}$

<table>
<thead>
<tr>
<th></th>
<th>Messages</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$2m + 2r</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>$2m + 2rn$</td>
<td>$O(rhd)$</td>
</tr>
<tr>
<td>Γ</td>
<td>$2m + O(r(n + e'))$</td>
<td>$O(rh'd)$</td>
</tr>
</tbody>
</table>
Example

- $p \times p$ grid of complete $k$-graphs, with all nodes of neighboring $k$-graphs connected.
- Clusters = $k$-graphs
- $h = O(p)$
- $h' = O(1)$

<table>
<thead>
<tr>
<th></th>
<th>Messages</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$2m + O(rp^2k^2)$</td>
<td>$O(rd)$</td>
</tr>
<tr>
<td>B</td>
<td>$2m + O(rp^2k)$</td>
<td>$O(rp^d)$</td>
</tr>
<tr>
<td>C</td>
<td>$2m + O(rp^2k)$</td>
<td>$O(rd)$</td>
</tr>
</tbody>
</table>
Application 1: Breadth-first search

- Recap:
  - SynchBFS:
    - Constructs BFS tree
    - O( |E| ) messages, O( diam ) rounds
  - When run in asynchronous network:
    - Constructs a spanning tree, but not necessarily BFS
  - Modified version, with corrections:
    - Constructs BFS tree
    - O( n |E| ) messages, O( diam n d ) time (counting pileups)

- BFS using synchronizer:
  - Runs more like SynchBFS, avoids corrections, pileups
  - With Synchronizer A:
    - O( diam |E| ) messages, O( diam d ) time
  - With Synchronizer B:
    - Better communication, but costs time.
  - With Synchronizer Γ:
    - Better overall, in clustered graphs.
Application 2: Broadcast and ack

- Use synchronizer to simulate synchronous broadcast-ack combination.
- Assume known leader, but no spanning tree.
- Recap:
  - Synchronous Bcast-ack:
    - Constructs spanning tree while broadcasting
    - $O(|E|)$ messages, $O(\text{diam})$ rounds
  - Asynchronous Bcast-ack:
    - Timing anomaly: Construct non-min-hop paths, on which acks travel.
    - $O(|E|)$ messages, $O(n d)$ time
- Using (e.g.) Synchronizer A:
  - Avoids timing anomaly.
  - Broadcast travels on min-hop paths, so acks follow min-hop paths.
  - $O(\text{diam} |E|)$ messages, $O(\text{diam} d)$ time
Application 3: Shortest paths

- Assume weights on edges.
- Without termination detection.
- Recap:
  - Synchronous Bellman-Ford:
    - Allows some corrections, due to low-cost high-hop-count paths.
    - $O(n |E|)$ messages, $O(n)$ rounds
  - Asynch Bellman-Ford
    - Many corrections possible (exponential), due to message delays.
    - Message complexity exponential in $n$.
    - Time complexity exponential in $n$, counting message pileups.
- Using (e.g.) Synchronizer A:
  - Behaves like Synchronous Bellman-Ford.
  - Avoids corrections due to message delays.
  - Still has corrections due to low-cost high-hop-count paths.
  - $O(n |E|)$ messages, $O(n d)$ time
  - Big improvement.
Further work

• To read more:
  – See Awerbuch’s extensive work on
    • Applications of synchronizers.
    • Distributed algorithms for clustered networks.
  – Also work by Peleg

• Q: This work used a strategy of purposely slowing down portions of a system in order to improve overall performance. In which situations is this strategy a win?
Lower Bound on Time for Synchronization
Lower bound on time

- A, B, Γ emulate synchronous algorithms only in a local sense:
  - Looks the same to individual users,
  - Not to the combination of all users---can reorder events at different users.
- Good enough for many applications (e.g., data management).
- Not for others (e.g., embedded systems).

- Now show that global synchronization is inherently more costly than local synchronization, in terms of time complexity.
- Approach:
  - Define a particular global synchronization problem, the k-Session Problem.
  - Show this problem has a fast synchronous algorithm, that is, a fast algorithm using GlobSynch.
    - Time $O(kd)$, assuming GlobSynch takes steps ASAP.
  - Prove that all asynchronous distributed algorithms for this problem are slow.
    - Time $\Omega(k \text{diam } d)$.
  - Implies GlobSynch has no fast distributed implementation.
- Contrast:
  - A, SimpleSynch are fast distributed implementations of LocSynch.
k-Session Problem

• **Session:**
  – Any sequence of flash events containing at least one $\text{flash}_i$ event for each location $i$.

• **k-Session problem:**
  – Perform at least $k$ separate sessions (in every fair execution), and eventually halt.

• **Original motivation:**
  – Synchronization needed to perform parallel matrix computations that require enough interleaving of process steps, but tolerate extra steps.
Example: Boolean matrix computation

• \( n = m^3 \) processes compute the transitive closure of \( m \times m \) Boolean matrix \( M \).
• \( p_{i,j,k} \) repeatedly does:
  – read \( M(i,k) \), read \( M(k,j) \)
  – If both are 1 then write 1 in \( M(i,j) \)
• Each flash \( i,j,k \) in abstract session problem represents a chance for \( p_{i,j,k} \) to read or write a matrix entry.
• With enough interleaving (\( O(\log n) \) sessions), this is guaranteed to compute transitive closure.
Synchronous solution

• Fast algorithm using GlobSynch:
  – Just flash once at every round.
  – $k$ sessions done in time $O(kd)$, assuming GlobSynch takes steps ASAP.
Asynchronous lower bound

- Consider distributed algorithm A that solves the k-session problem.
- Consists of process automata and FIFO send/receive channel automata.

- Assume:
  - $d$ = upper bound on time to deliver any message (don’t count pileups)
  - $l$ = local processing time, $l << d$

- Define time measure $T(A)$:
  - Timed execution $\alpha$: Fair execution with times labeling events, subject to upper bound of $d$ on message delay, $l$ for local processing.
  - $T(\alpha) =$ time of last flash in $\alpha$
  - $T(A) =$ supremum, over all timed executions $\alpha$, of $T(\alpha)$. 

\[ \text{flash}_1 \quad \text{flash}_2 \quad \text{flash}_n \]
Lower bound

• **Theorem 2**: If A solves the k-session problem then $T(A) \geq (k-1) \text{diam } d$.
• Factor of diam worse than the synchronous algorithm.

• **Definition**: Slow timed execution: All message deliveries take exactly the upper bound time $d$.

• **Proof**: By contradiction.
  – Suppose $T(A) < (k-1) \text{diam } d$.
  – Fix $\alpha$, any slow timed execution of A.
  – $\alpha$ contains at least $k$ sessions.
  – $\alpha$ contains no flash event at a time $\geq (k-1) \text{diam } d$.
  – So we can decompose $\alpha = \alpha_1 \alpha_2 \ldots \alpha_{k-1} \alpha''$, where:

    • Time of last event in $\alpha'$ is $< (k-1) \text{diam } d$.
    • No flash events occur in $\alpha''$.
    • Difference between the times of the first and last events in each $\alpha_r$ is $< \text{diam } d$. 

Lower bound, cont’d

• Now reorder events in $\alpha$, while preserving dependencies:
  – Events of same process.
  – Send and corresponding receive.
• Reordered execution will have < $k$ sessions, a contradiction.
• Fix processes, $j_0$ and $j_1$, with $\text{dist}(j_0, j_1) = \text{diam}$ (maximum distance apart).
• Reorder within each $\alpha_r$ separately:
  – For $\alpha_1$: Reorder to $\beta_1 = \gamma_1 \delta_1$, where:
    • $\gamma_1$ contains no event of $j_0$, and
    • $\delta_1$ contains no event of $j_1$.
  – For $\alpha_2$: Reorder to $\beta_2 = \gamma_2 \delta_2$, where:
    • $\gamma_1$ contains no event of $j_1$, and
    • $\delta_1$ contains no event of $j_0$.
  – And alternate thereafter.
Lower bound, cont’d

• If the reordering yields a fair execution of A (can ignore timing), then we get a contradiction, because it contains $\leq k-1$ sessions:
  – No session entirely within $\gamma_1$, (no event of $j_0$).
  – No session entirely within $\delta_1 \gamma_2$ (no event of $j_1$).
  – No session entirely within $\delta_2 \gamma_3$ (no event of $j_0$).
  – ...
  – Thus, every session must span some $\gamma_r - \delta_r$ boundary.
  – But, there are only $k-1$ such boundaries.

• So, it remains only to construct the reordering.
Constructing the reordering

- WLOG, consider $\alpha_r$ for $r$ odd.
- Need $\beta_r = \gamma_r \delta_r$, where $\gamma_r$ contains no event of $j_0$, $\delta_r$ no event of $j_1$.
- If $\alpha_r$ contains no event of $j_0$ then don’t reorder, just define $\gamma_r = \alpha_r$, $\delta_r = \lambda$.
- Similarly if $\alpha_r$ contains no event of $j_1$.
- So assume $\alpha_r$ contains at least one event of each.
- Let $\pi$ be the first event of $j_0$, $\varphi$ the last event of $j_1$ in $\alpha_r$.

Claim: $\varphi$ does not depend on $\pi$.
Why: Insufficient time for messages to travel from $j_0$ to $j_1$:
  - Execution $\alpha$ is slow (message deliveries take time $d$).
  - Time between $\pi$ and $\varphi$ is $< \text{diam } d$.
  - $j_0$ and $j_1$ are diam apart.

Then, we can reorder $\alpha_r$ to $\beta_r$, in which $\pi$ comes after $\varphi$.
Consequently, in $\beta_r$, all events of $j_1$ precede all events of $j_0$.
Define $\gamma_r$ to be the part ending with $\varphi$, $\delta_r$ the rest.
Next time…

• Time, clocks, and the ordering of events in a distributed system.
• State-machine simulation.
• Vector timestamps.
• Reading:
  – Chapter 18
  – [Lamport time, clocks…paper]
  – [Mattern paper]