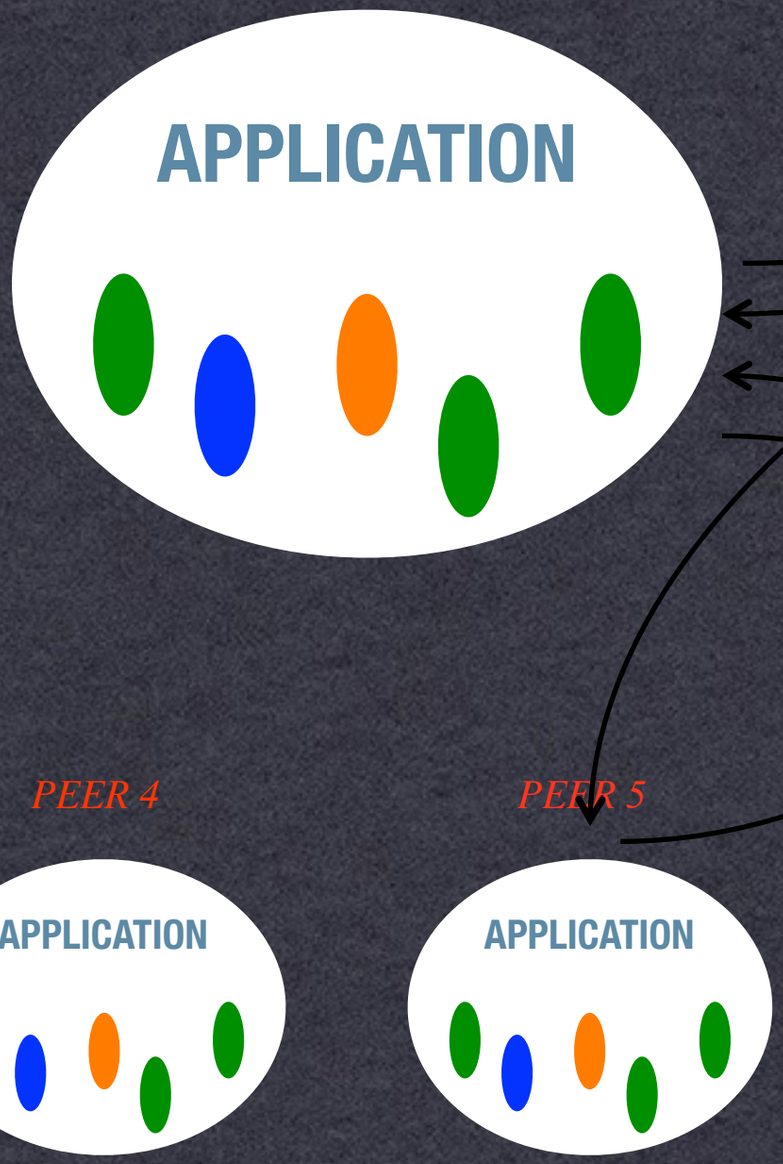
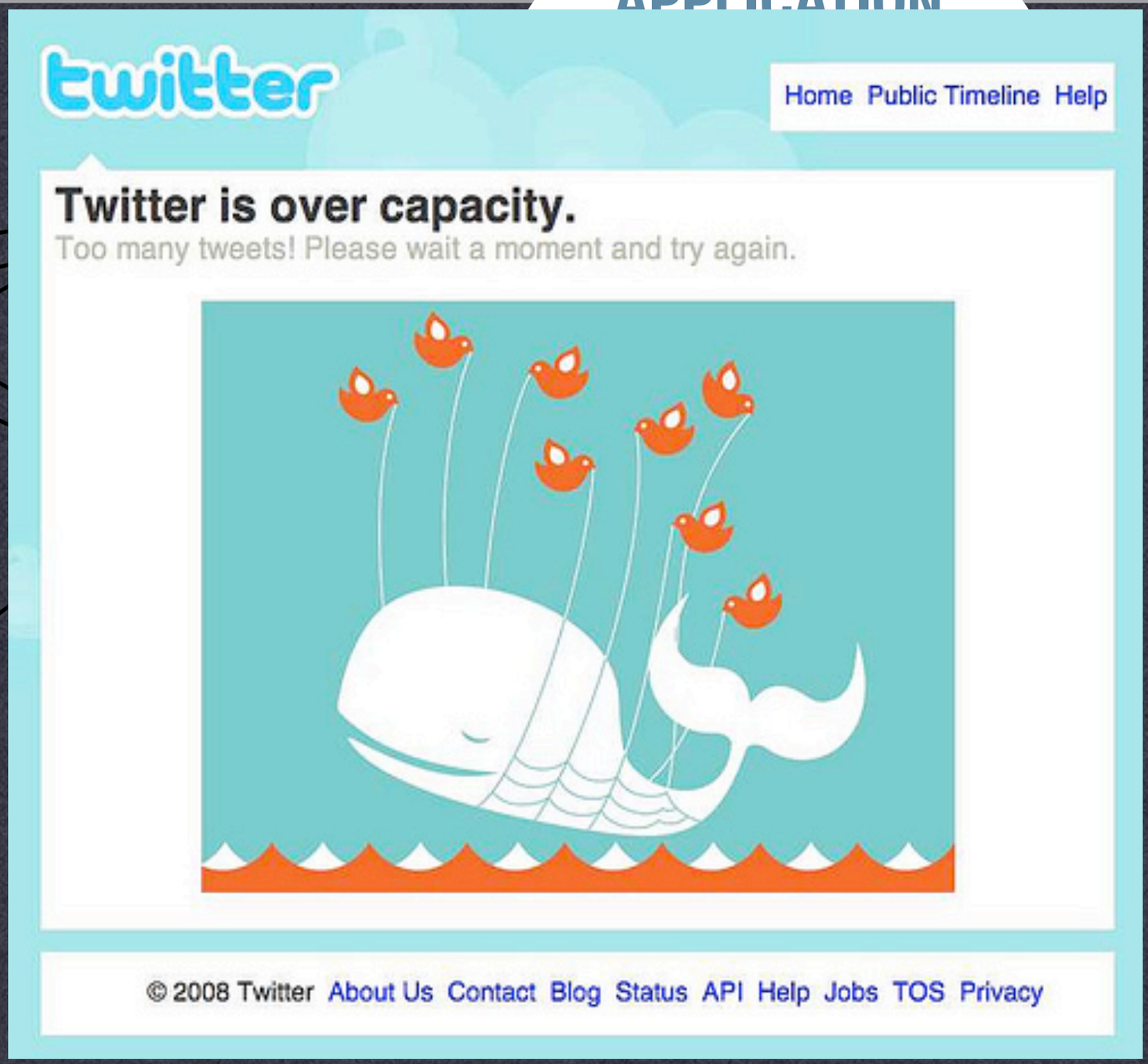




DISTRIBUTED SYSTEMS

INTRODUCTION TO FAULT TOLERANCE

Past lectures …

- Remote operations
  - Latency reduction
    - Pcall, one-way RPC, futures and promises, batched futures, responsibilities, …
- Linguistic heterogeneity
  - CORBA IDL
  - XML

Architectural models

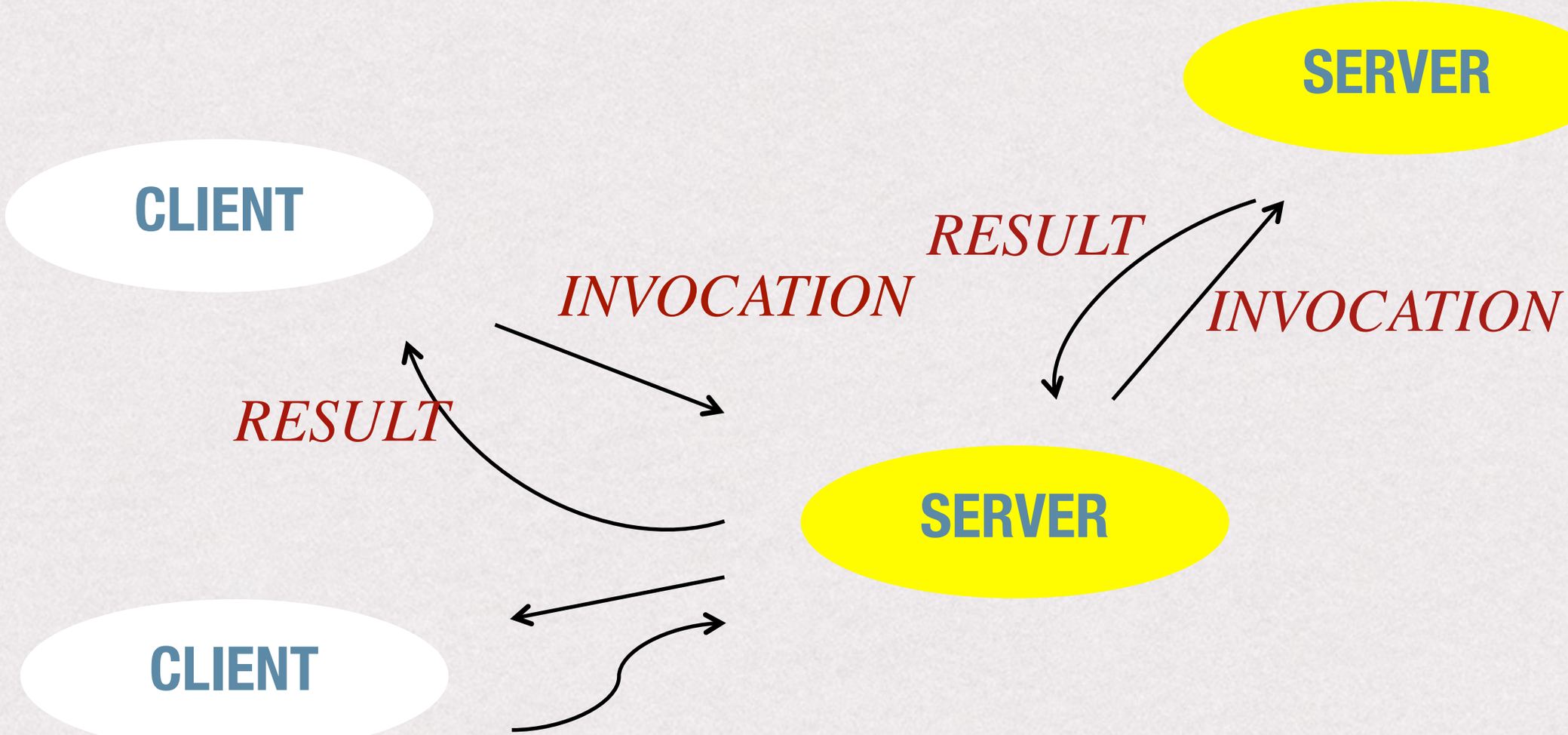
- Distribution of responsibility
  - Client-server
  - Peer-to-peer

SYSTEMS TALK

Client - Server

- Servers manage a set of resources or provide a set of services
- Clients interact with servers to obtain access from resources or result from services
- Servers can be clients; many clients can interact with many servers

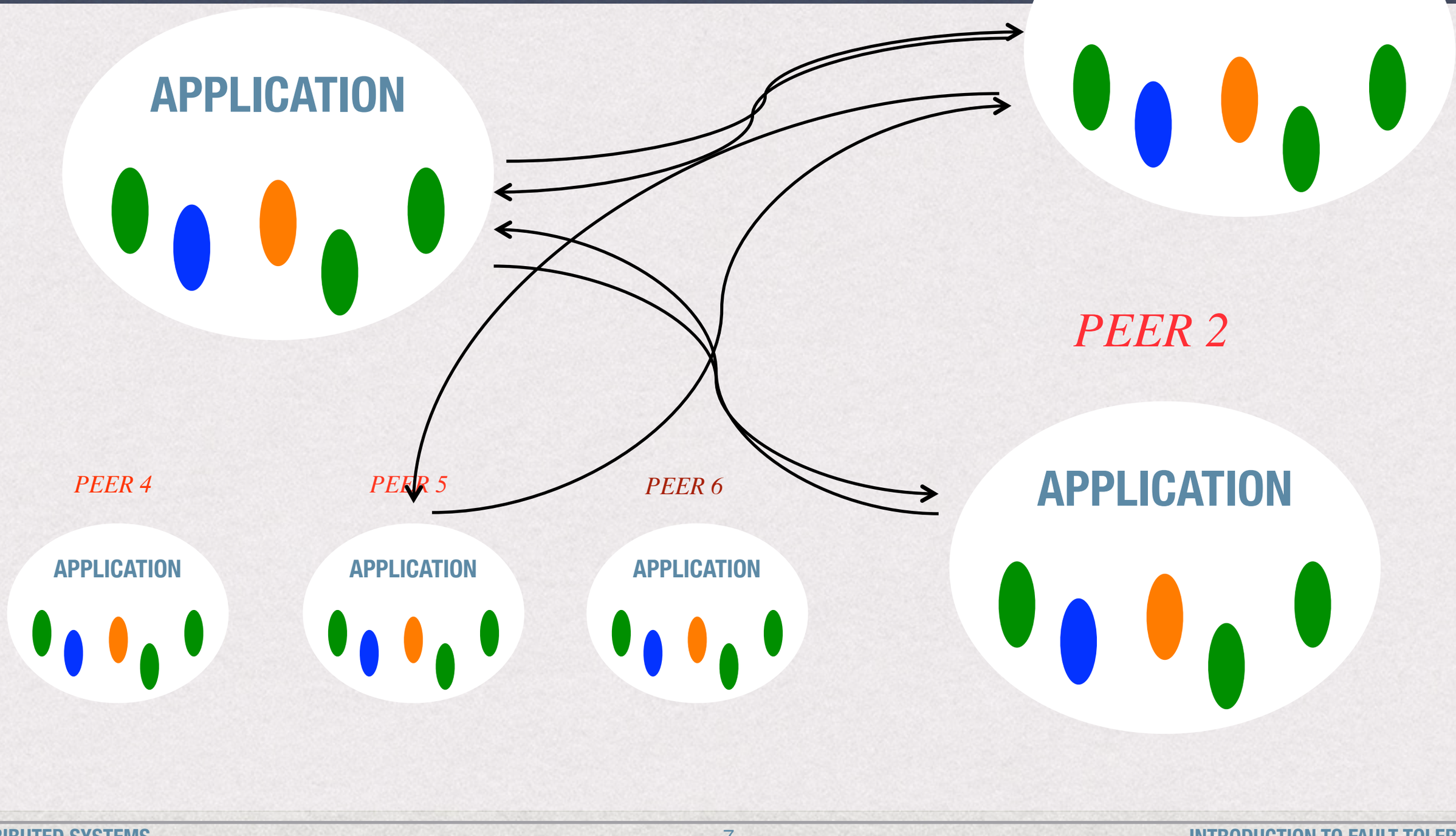
Client - Server



Peer-to-Peer

- All of the processes play similar roles, interacting cooperatively as equal peers
- No distinction between peers
- Scales well as peers are responsible for parts of a service
- Sometimes, provides better service if there is a large number of peers
  - E.g. BT

Peer-to-Peer


PEER 1
APPLICATION
PEER 3
APPLICATION
PEER 2
PEER 4
APPLICATION
PEER 5
APPLICATION
PEER 6
APPLICATION
APPLICATION

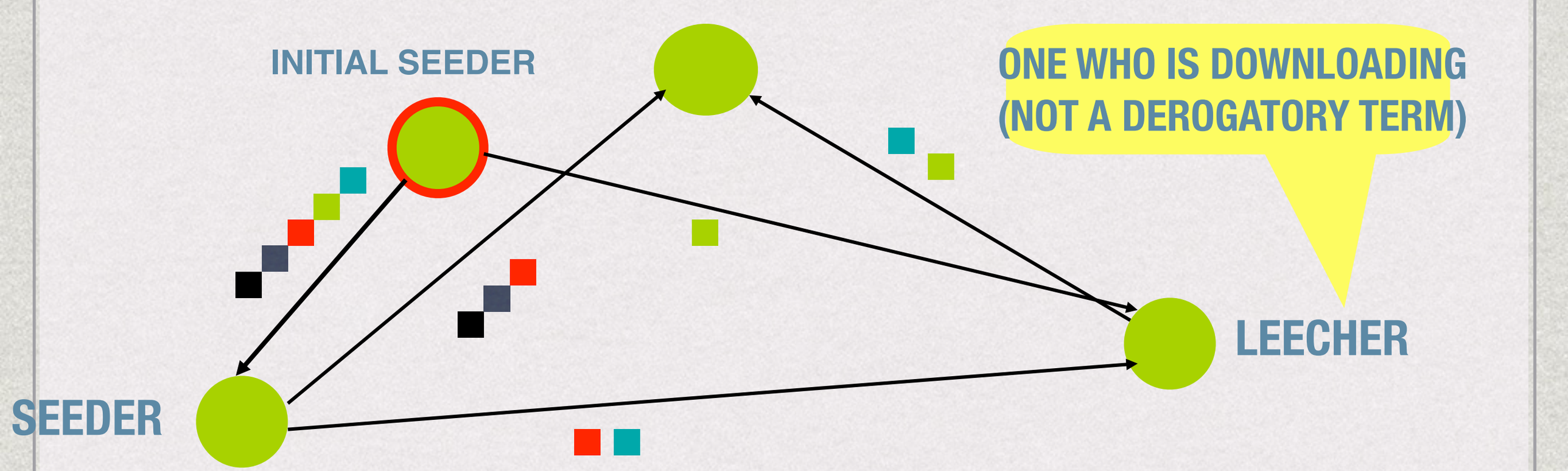
BitTorrent

- Protocol for transmitting large files
- User that wants to distribute a file creates a torrent
- User makes file available to a seed
- File is divided into segments: pieces
- As each peer receives a piece for a file, it becomes a seed for that piece
- Pieces are downloaded non-sequentially

BitTorrent Lingo

SEEDER = A PEER THAT PROVIDES THE COMPLETE FILE.

INITIAL SEEDER = A PEER THAT PROVIDES THE INITIAL COPY.



Simple example

{1,2,3,4,5,6,7,8,9,10}

SEEDER: A

{1,2,3,4}
{1,2,3,4,5}

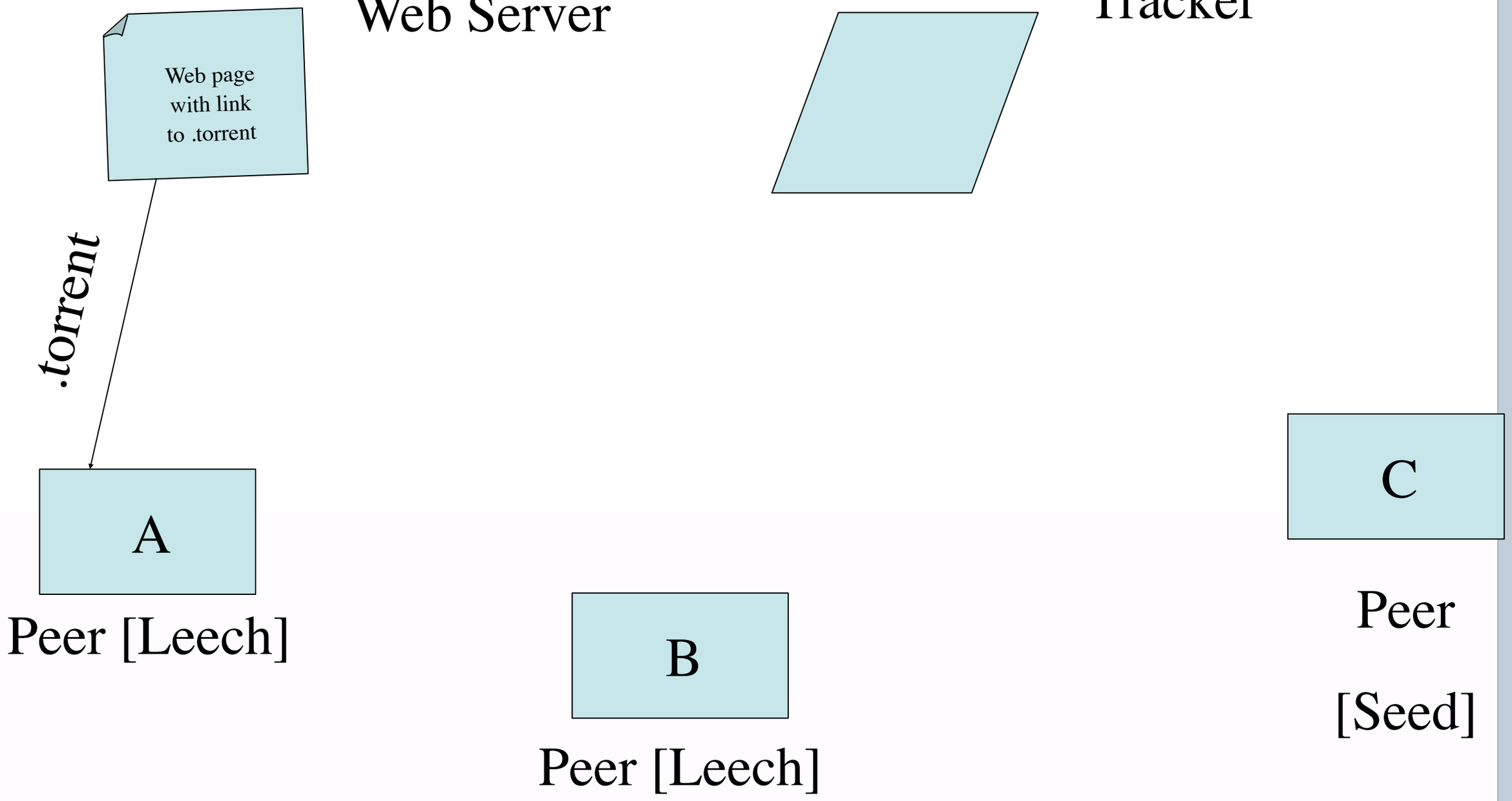
DOWNLOADER B

{}
{1,2,3}

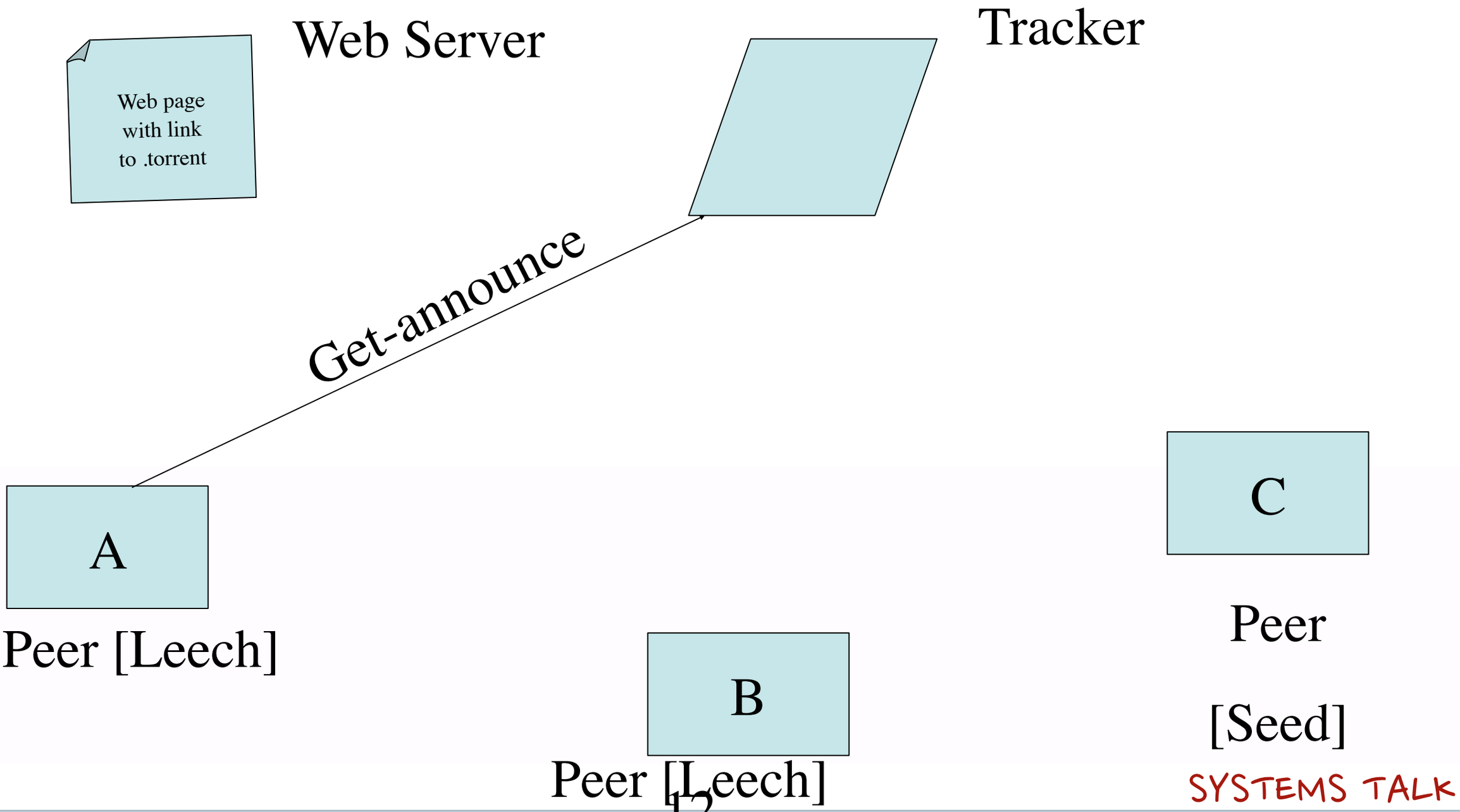
DOWNLOADER C

SYSTEMS TALK

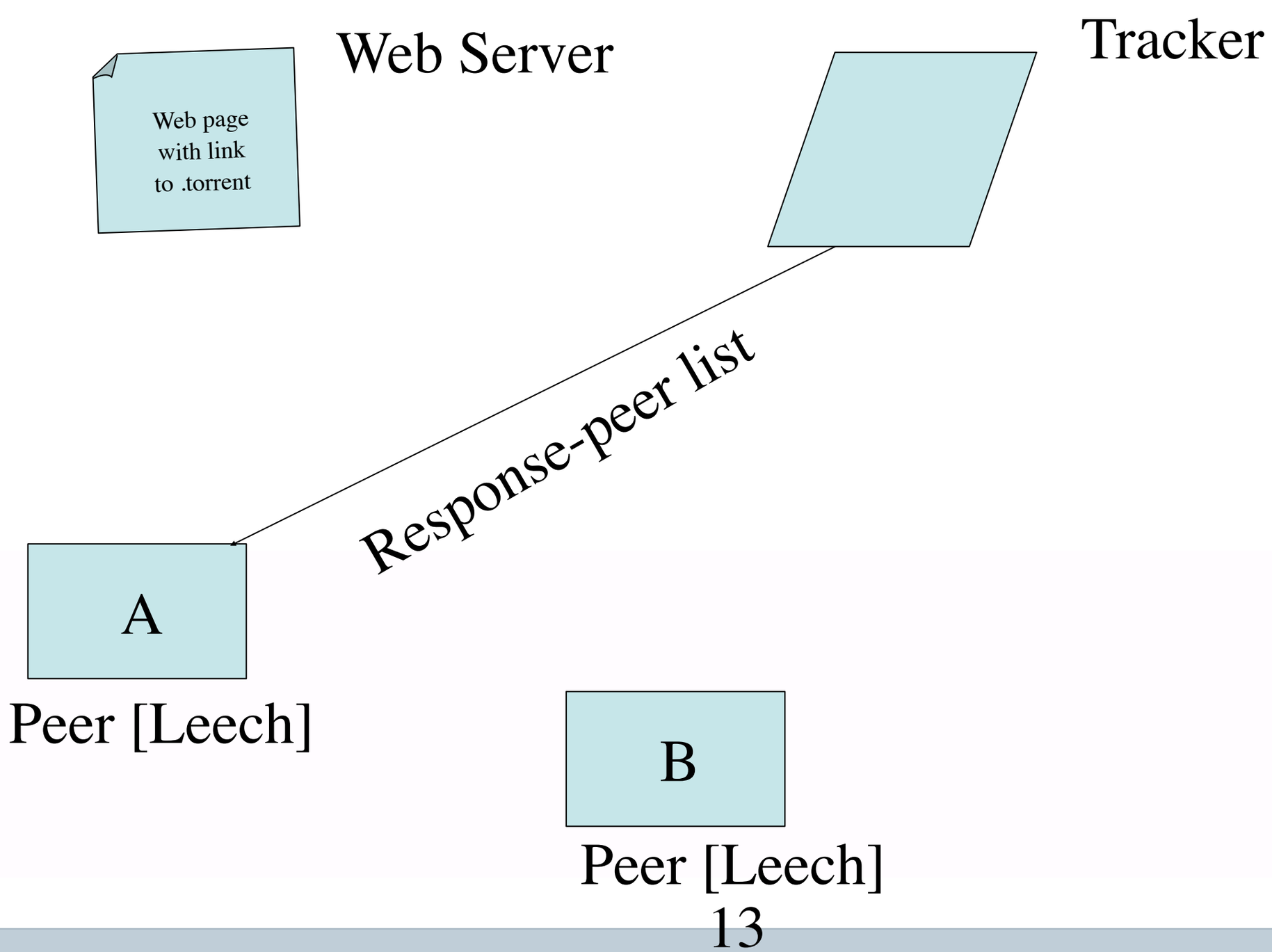
Overview



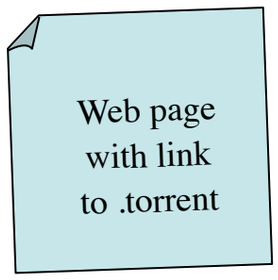
Overview



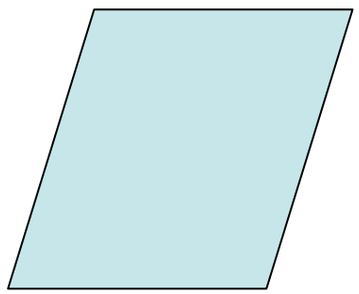
Overview



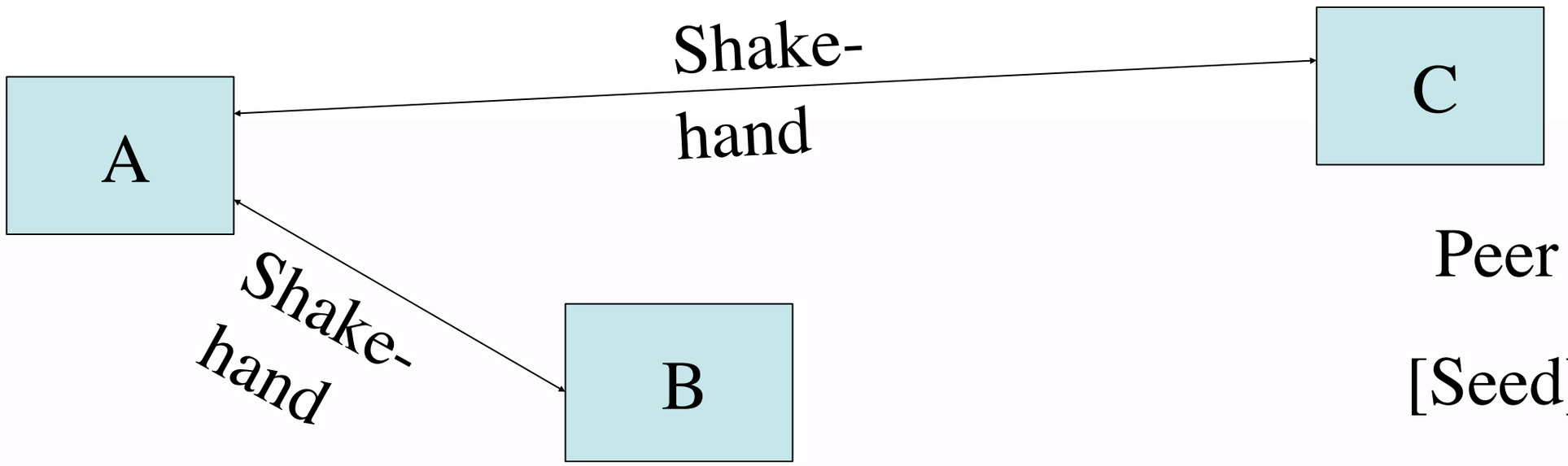
Overview



Web
Server



Tracker

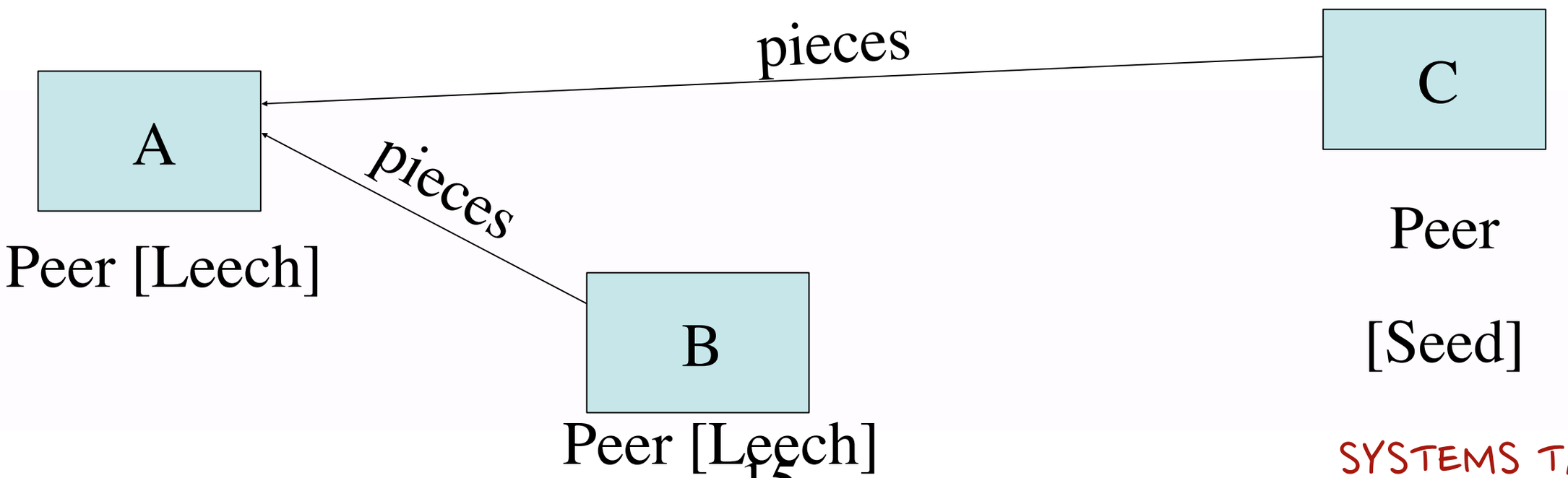


Overview

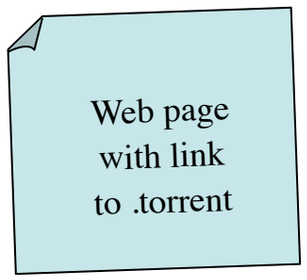
Web page with link to .torrent

Web Server

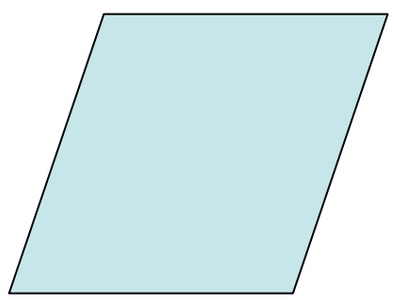
Tracker



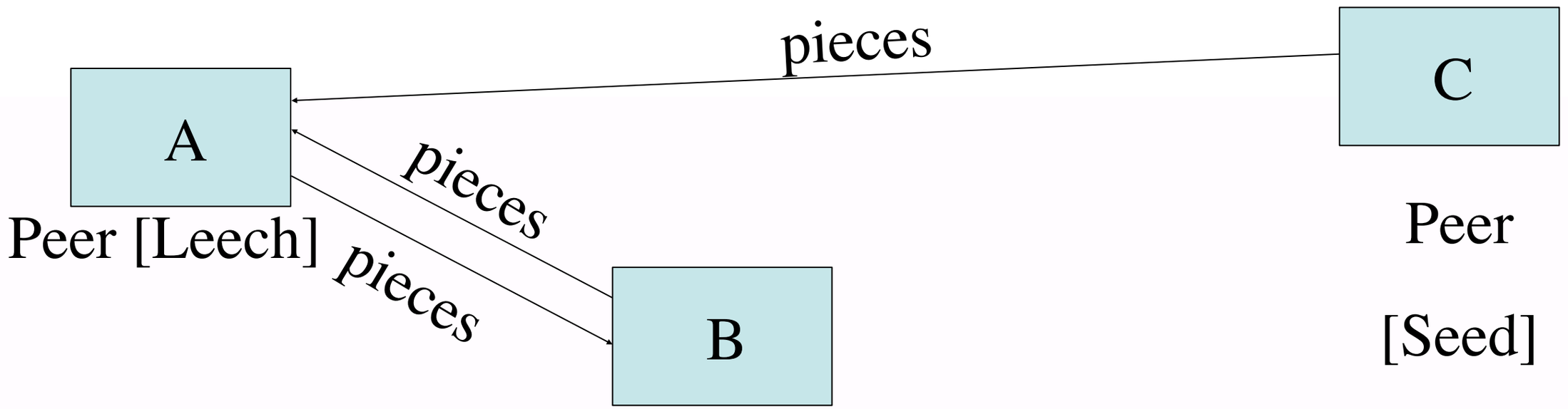
Overview



Web Server

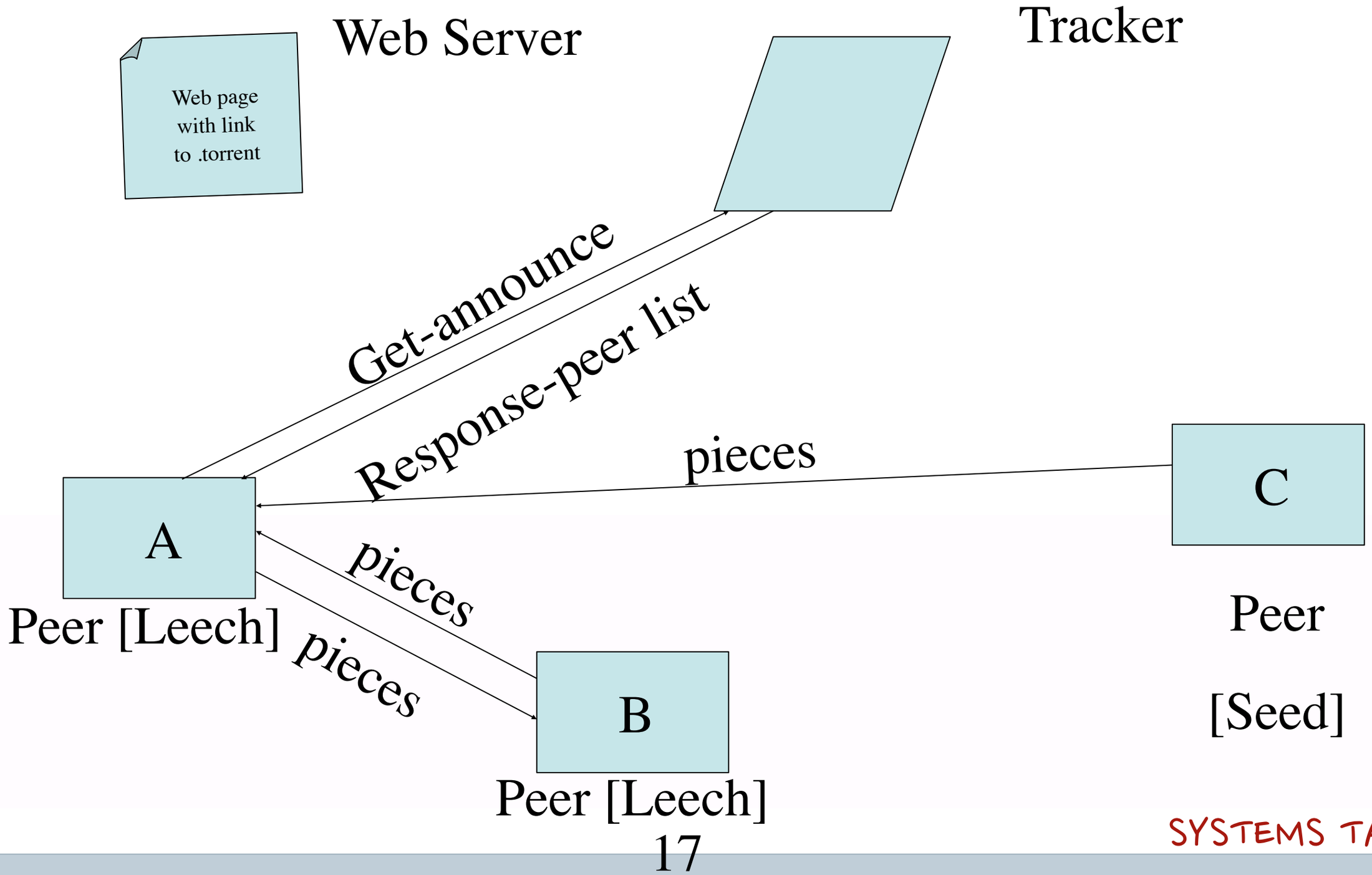


Tracker



SYSTEMS TALK

Overview



Cool stuff

- Distributed trackers
  - Trackerless torrents;
  - DHT implementation
- Decentralized keyword search
- Choking
- Optimistic unchoking
  - Clients try to periodically sending to choked connections

Chocking

- Ensures that nodes cooperate and eliminates the free-rider problem
- Cooperation involves uploaded sub-pieces that you have to your peer
- Choking is a temporary refusal to upload; downloading occurs as normal
- Connection is kept open so that setup costs are not borne again and again
- Based on game-theoretic concepts
  - Tit-for-tat strategy in Repeated Games

Piece Selection

- The order in which pieces are selected by different peers is critical for good performance
- If a bad algorithm is used, we could end up in a situation where every peer has all the pieces that are currently available and none of the missing ones
- If the original seed is taken down, the file cannot be completely downloaded!

Random First Piece

- Initially, a peer has nothing to trade
- Important to get a complete piece ASAP
- Rare pieces are typically available at fewer peers, so downloading a rare piece initially is not a good idea
- Policy: Select a random piece of the file and download it

Rarest Piece First

- Policy: Determine the pieces that are most rare among your peers and download those first
- This ensures that the most common pieces are left till the end to download
- Rarest first also ensures that a large variety of pieces are downloaded from the seed

Chocking Algorithm

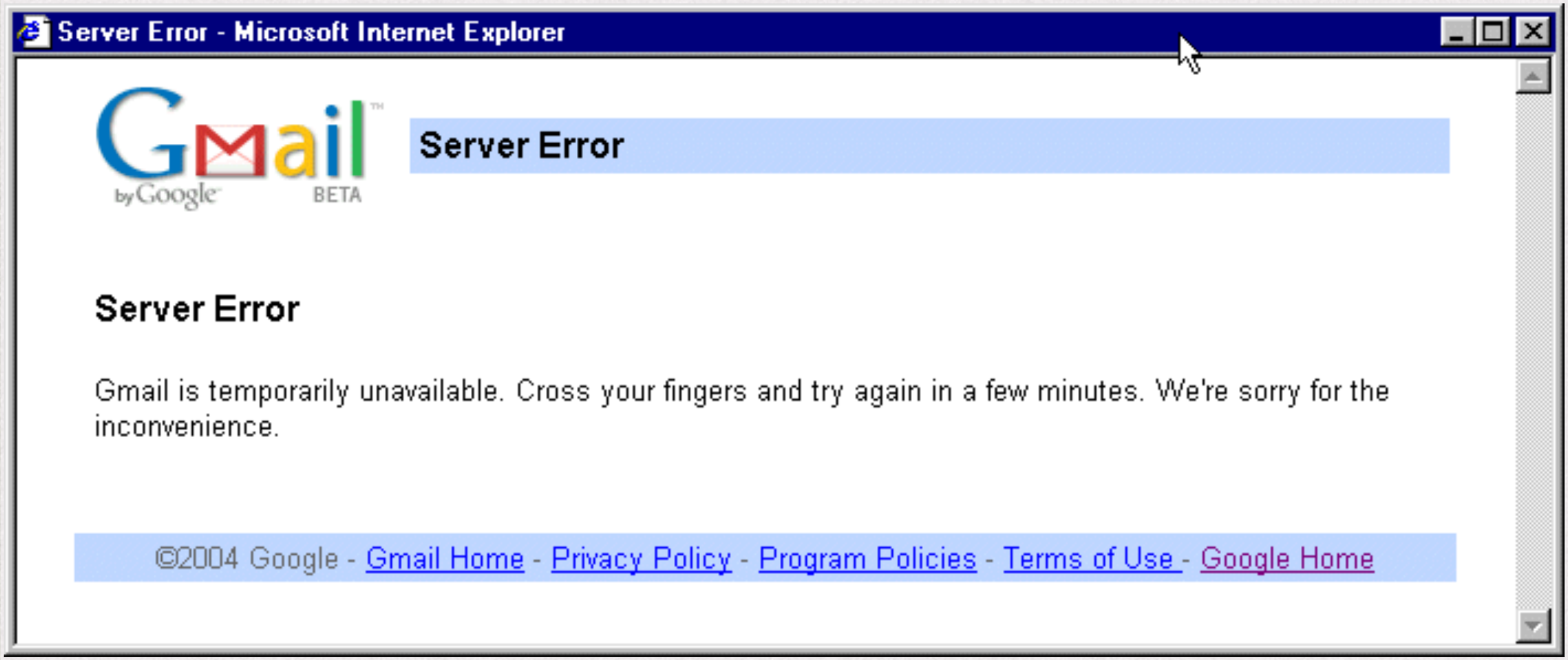
- Goal is to have several bidirectional connections running continuously
- Upload to peers who have uploaded to you recently
- Unutilized connections are uploaded to on a trial basis to see if better transfer rates could be found using them

How BitTorrent works

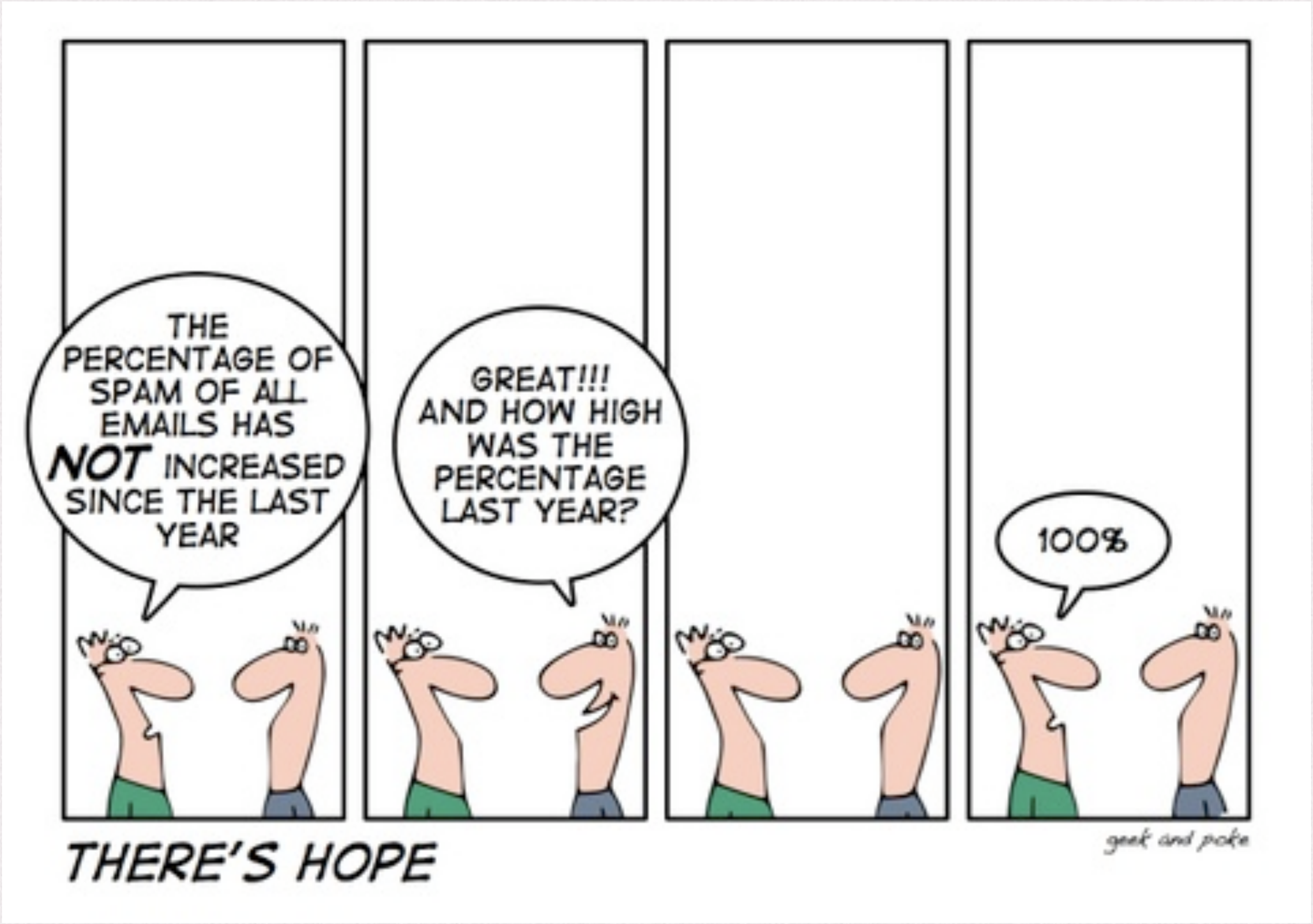
- http://mg8.org/processing/bt.html

This lecture ...

This lecture ...



This lecture ...



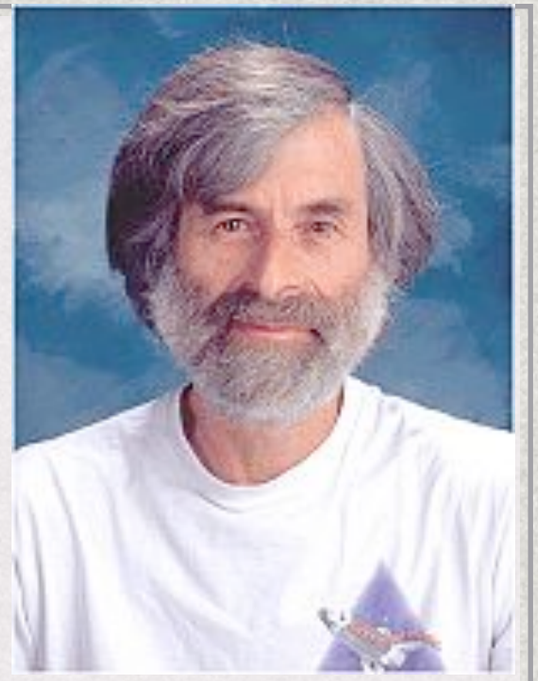
Theme

- A distributed system is, "***One on which I cannot get any work done because some machine I have never heard of has crashed***"

- A well-known quotation from Leslie Lamport

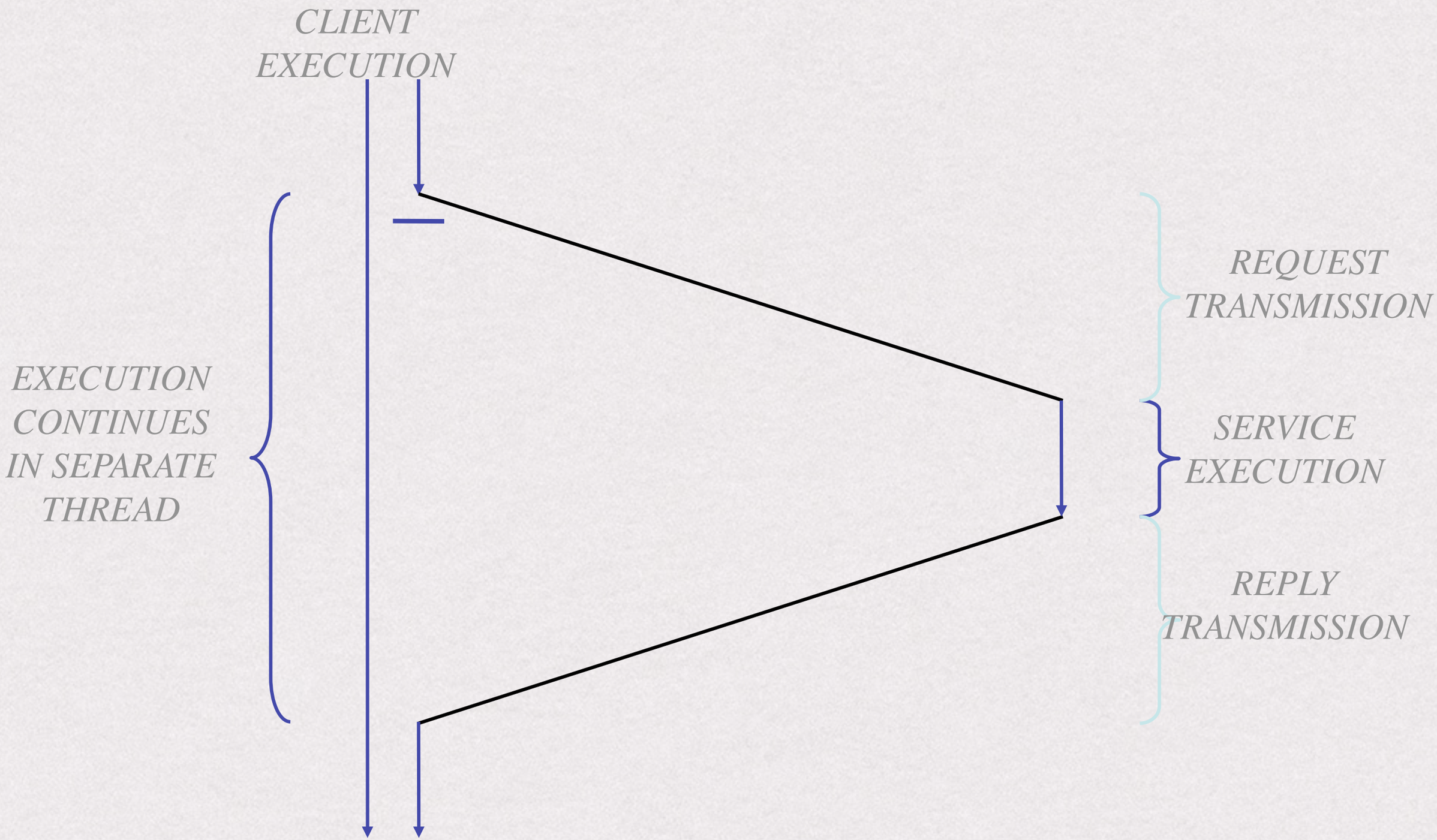
- Concerned with the (increased) propensity of distributed systems to failure, and the complex consequences of those failures.

Leslie Lamport



- Seminal work in distributed systems
  - Time, clocks and the order of events
  - The byzantine general's problem
    - Reaching agreement in the presence of faults
  - Mutual exclusion using the bakery algorithm
  - Snapshot algorithms for determining consistent global states
- The initial developer of LaTeX

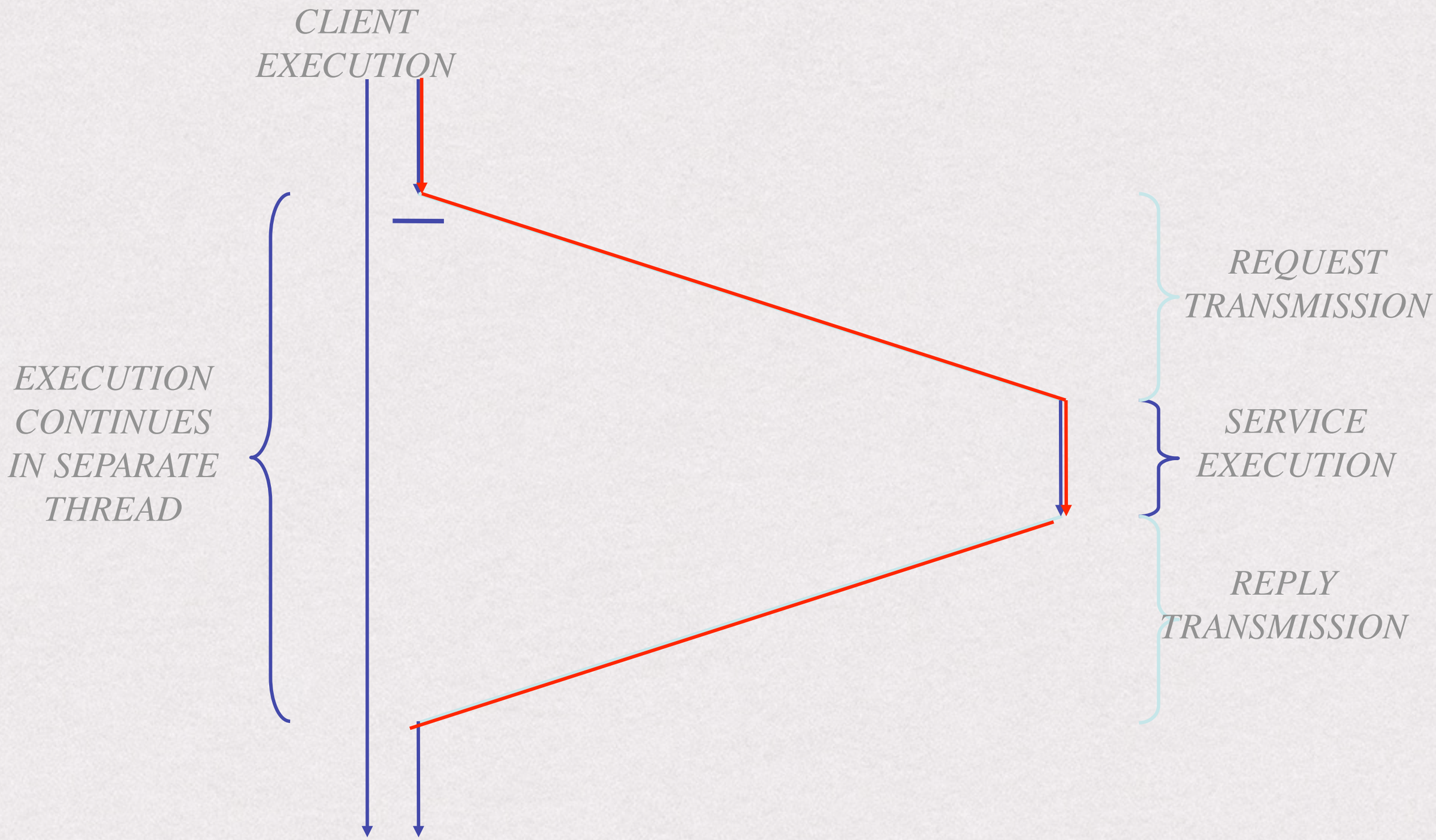
Remote Calls



Questions

- What can fail between the client starting an RPC and the end of that RPC?

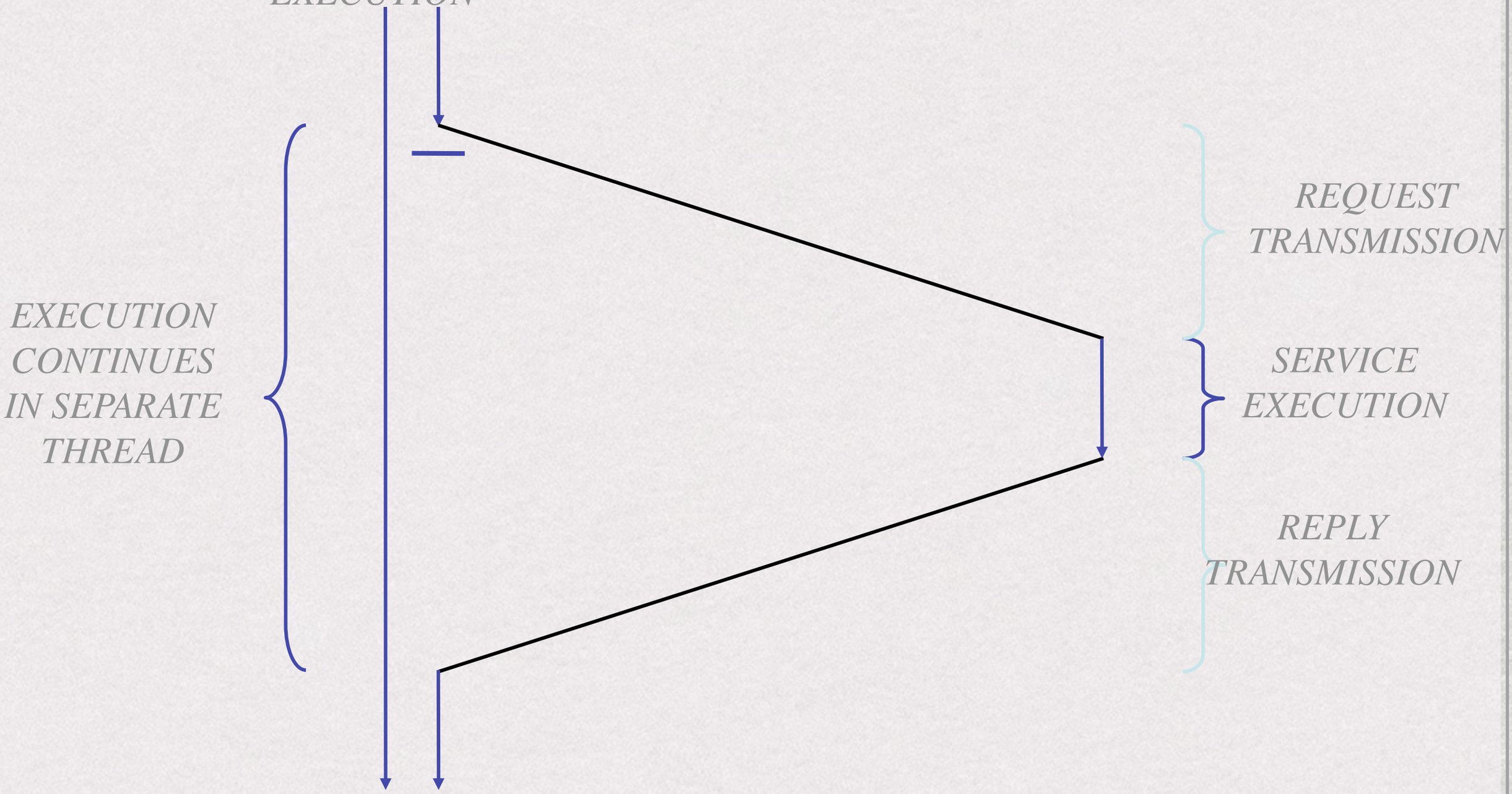
Remote Calls



Questions

- ***What can fail between the client starting an RPC and the end of that RPC?***
- ***What happens to the client when each sort of failure occurs?***

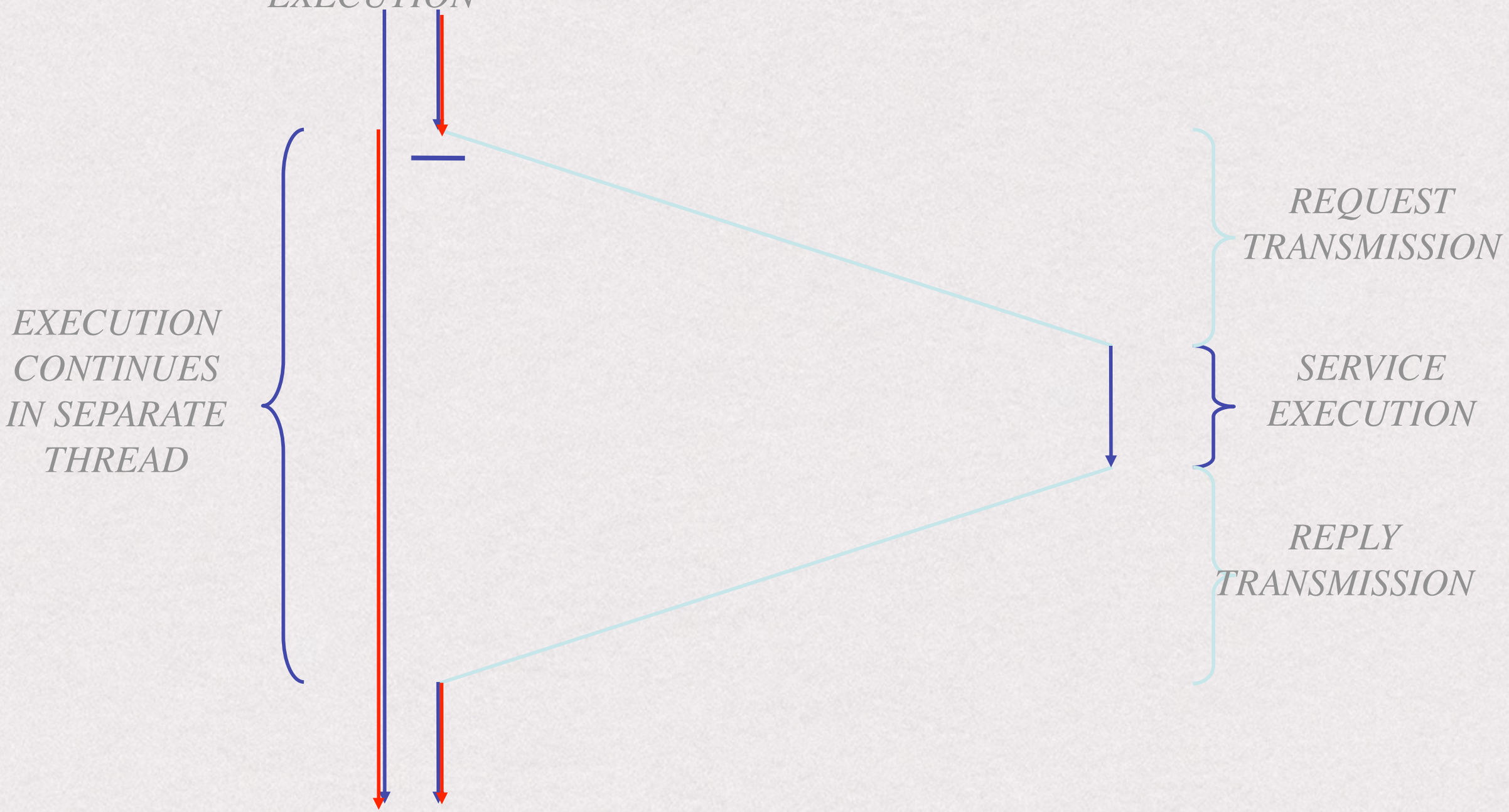
Remote Calls



Questions

- **What can fail between the client starting an RPC and the end of that RPC?**
- **What happens to the client when each sort of failure occurs?**
- **What happens to the server when each sort of failure occurs?**

Remote Calls



Questions

- What can fail between the client starting an RPC and the end of that RPC?
- What happens to the client when each sort of failure occurs?
- What happens to the server when each sort of failure occurs?
- What happens to the distributed system as a whole in the presence of failures?

Questions

- What can fail between the client starting an RPC and the end of that RPC?
- What happens to the client when each sort of failure occurs?
- What happens to the server when each sort of failure occurs?
- What happens to the distributed system as a whole in the presence of failures?
- Or more positively:

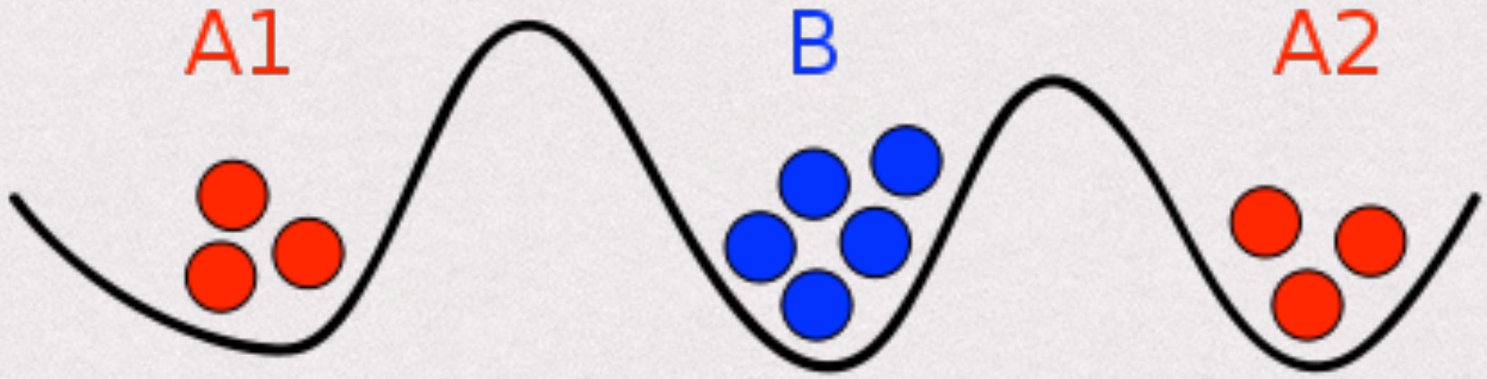
How can we build distributed systems that do useful work in spite of the propensity to failure?

Google failure rates

- In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours;
- 20 racks will fail, each time causing 40 to 80 machines to vanish from the network;
- 5 racks will "go wonky," with half their network packets missing in action; 50% chance that the cluster will overheat

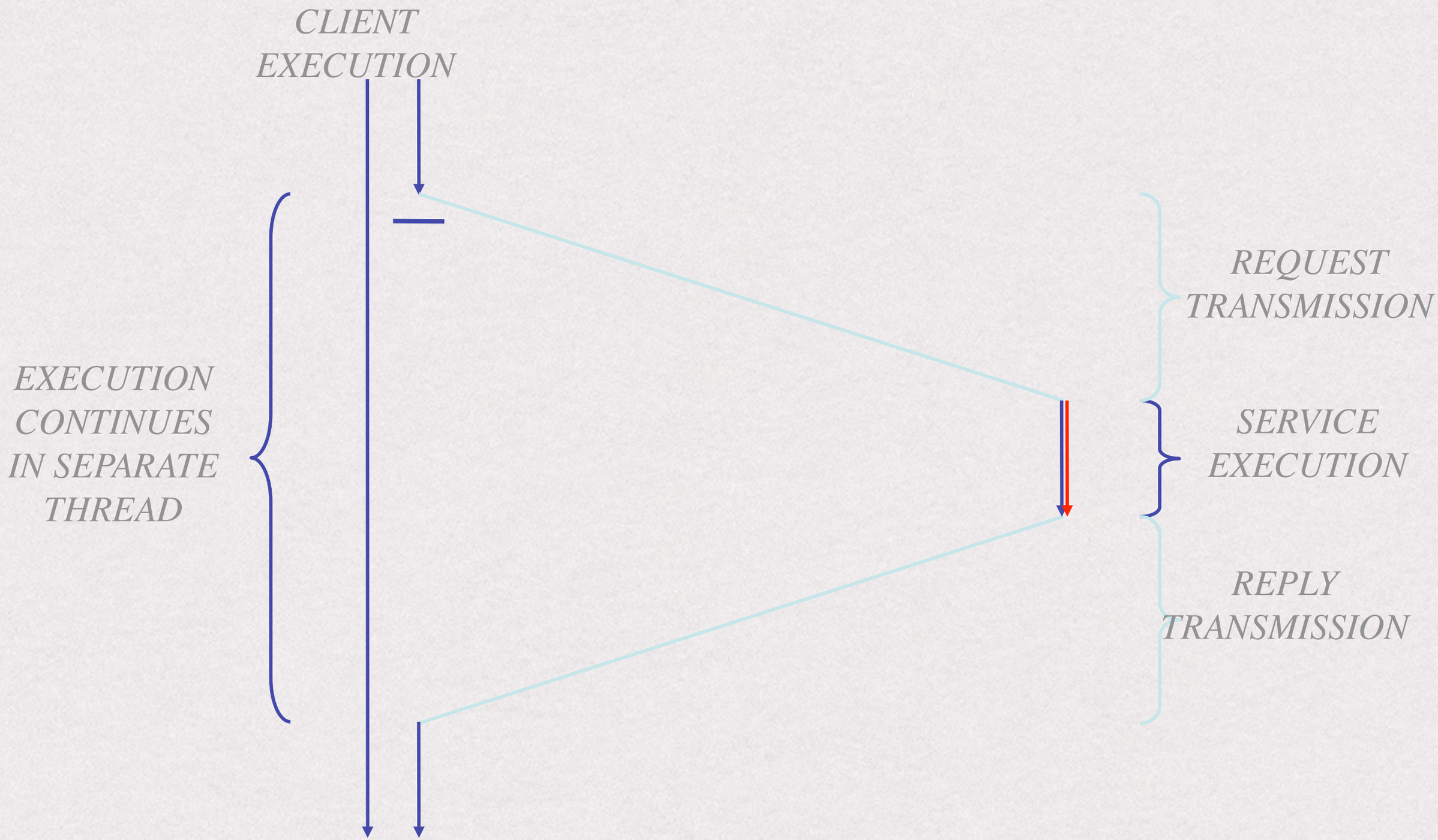
Model of Failure

- The network may lose both request and reply messages.



- Servers may crash, possibly whilst RPCs are in progress, and may then subsequently restart and resume receiving requests
- Clients may crash whilst RPCs are in progress, and may then subsequently restart and resume receiving replies.
- Messages are delivered in the order sent:
  - There are no message re-ordering failures

Remote Calls



Server Failure

- Server crashes and the loss of requests and replies by the network creates a problem for the client.
- The client will use a ***FAILURE DETECTOR*** to discern when such a problem occurs:
  - In principle a failure detector will detect when failures occur.
  - In practice, failure detectors operate by failing to determine that an operation has succeeded within a given time!

Failure Detectors

- A failure detector returns the following:
  - It has not been possible to determine that an operation X has succeeded within time T.
- this is interpreted as:
  - The operation X has failed.
- The operation may have failed, or it may be that the time T is too short:
  - What can one do about this “false failure” result?

Use of Failure Detectors

- Prior to sending a request message, the client sets a timer to expire at some point in the future:
  - If the timer expires before a reply to the request arrives, then the call is ***DECLARED FAILED***.
  - However, this declaration might be wrong – it might just be that the timer duration is too short for processing of the particular request.
- Clearly an infinitely long timer will avoid false failures:
  - This approach is not useful however, since it will not detect true failures either!

THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

Adaptive Timeouts

- Instead, RPC systems use an ***ADAPTIVE TIMEOUT*** approach:
  - If the initial timer expires, the client sets a new timer and sends the server a ***PROBE*** message.
  - The server is configured to acknowledge probes immediately if the request is still being processed.
  - If the client receives the probe acknowledgement before the new timer expires, it knows the server is still working on the request.
  - After receiving a probe acknowledgement, the client sets another timer.
  - Then, at the expiry of this timer, it sets (yet) another timer and sends off another probe.
- This process continues until either the reply is received or one of the probe acknowledgements fails to arrive:
  - For efficiency, the inter probe delay may increase up to a point.

Discriminating between Failures

- The (adaptive) timeout approach does not discriminate between different possible failures:
  - Request (or probe) loss on the way to the server.
  - Server crash.
  - Reply (or probe acknowledgement) loss on the way back to the client.

Discriminating between Failures

- Client's perspective:
  - all failures looks essentially the same – an expected message does not arrive in time.
- It is sometimes possible to find out more detail:
  - Once the client has received at least one probe acknowledgment from the server, it is possible to deduce that the request made it to the server.
  - However, one can't conclude from the absence of a probe acknowledgement that the request did not make it to the server – perhaps the probe or probe acknowledgement was lost.

Re-processing of Requests

- Consider a server operation that adds one to a number:
  - For the RPC system to re-process such a request in the event of failure detection would be incorrect, since its not possible to tell whether the request was processed (to the extent of adding one) previously.
  - Re-processing could lead to the number being increased by two!

```
server.increaseCounter();
```

Re-processing of Requests

- Consider a server operation that adds one to a number:
  - For the RPC system to re-process such a request in the event of failure detection would be incorrect, since its not possible to tell whether the request was processed (to the extent of adding one) previously.
  - Re-processing could lead to the number being increased by two!
- So, re-transmission (and consequent re-processing) of requests runs the risk that a request may be executed more than once:
  - This could be bad, for example a request to deduct some money from a bank account.
  - In these situations, the RPC system should provide AT-MOST-ONCE semantics – a given request must never be processed more than once by the server:
    - The corollary is that sometimes a request will not be executed at all!
- As a result, it is not always permissible for an (at-most-once) RPC system to re-process a request when a failure is detected.

An Optimisation

- Server discriminates lost request failures under certain conditions:
  - Server has an ID that is unique each time it re-starts:
    - For example, the number of times the server has started.
  - The client obtains the server's ID each time it binds, and sends that ID with each request to that server.
  - Each client request is allocated a unique sequence number.
  - On failure detection, instead of a reporting an error, the client resends a copy of the request with its original sequence number.
  - The server maintains (in memory) a list of requests it has processed.
  - When the server receives a request, if it has the current server ID:
    - If in the list of processed requests, send a copy of the reply.
    - If not, process the request.
  - If the request has the wrong server ID, the server has recently crashed, and it can't tell whether it has been processed, so discard it and send back an error message.
- This technique allows the system to overcome some lost request and lost reply failures whilst retaining at-most-once semantics.

An Optimisation

1. CLIENT BINDS TO THE SERVER AND GETS ID

2. WHENEVER THE CLIENT TALKS TO THE SERVER, IT WILL USE SID AND A REQUEST ID

An Optimisation

3. SERVER PROCESSES REQUEST AND SAVES RESULT SOMEWHERE

CLIENT

SERVER

SERVER ID: 1
RID 1: 42

4. IF CLIENT DOES NOT GET A RESPONSE FROM THE SERVER, IT WILL RESEND

CLIENT

SERVER

SERVER ID: 1
RID 1: 42

CLIENT

SID: 1, RID: 1

SERVER

SERVER ID: 1
RID 1: 42

An Optimisation

5. IF THE SERVER ID IS THE SAME, THE SERVER WILL RE-SEND THE REPLY

6. IF SERVER HAS CRASHED IN THE MEANTIME, THE SERVER IDS WILL BE DIFFERENT; ERROR MESSAGE WILL BE SENT

Client Failures

- When a client crashes, it is wasteful to continue processing any requests from that client.
- More importantly, the optimisation described previously relies on **unique** sequence numbers, and a client's sequence numbers are reset when the client restarts:
  - Request messages need to include a unique client ID.
  - This is derived in the same way as the server ID.
  - Taken together, these form a session ID:
    - used instead of the server ID
- Also, the client needs to differentiate a reply to a request it has sent since it booted from a reply to a request that it sent before it crashed:
  - Reply messages also need to have session ID and sequence numbers.
  - Ignore replies from a previous session.

At-least-once Semantics

- There is an alternative RPC semantics:
  - With ***AT-LEAST-ONCE*** semantics, each request is guaranteed to execute to completion.
  - The corollary is that each request may in fact execute (wholly or partially) more than one time.
- Implementation of at-least-once semantics is trivial:
  - Re-send each request until a reply is received!
- So, why is at-least-once semantics not the answer to all problems?
- Operations that are suitable for calling with at-least-once semantics are said to be IDEMPOTENT

Other Semantics

- The at-most-once semantics is more generally useful, but:
  - Even the optimisation to at-most-once doesn't give certainty in all cases – it's still at-most-once, just that the "once" part is more likely.
  - In the event a failure is detected, the operation may have executed partially and then crashed – the RPC system says nothing about this.
- Other possible semantics:
  - ***ZERO-OR-ONCE*** – each operation either executes completely or not at all – failures due to partial executions are prevented.
  - ***RELIABLE NETWORK*** – failures due to loss of request and replies are prevented.
  - ***EXACTLY-ONCE*** – each operation executes to completion.
- How do you think we can go about achieving exactly-once semantics?
  - Hint, think about the optimisation to at-most-once, and what caused it to break down.