LECTURE 6: MESSAGE-ORIENTED COMMUNICATION II: MESSAGING IN DISTRIBUTED SYSTEMS
Lecture Contents

• What do we mean by Distributed Systems?
• Middleware in Distributed Systems
• Types of Distributed Communications
  – Remote Procedure Call (RPC):
    • Parameter passing, Example: DCE
    • Registration & Discovery in DCE
  – Message Queuing Systems:
    • Basic Architecture, Role of Message Brokers
      – Example: IBM Websphere
    • Advanced Message Queuing Protocol (AMQP)
      – Example: Rabbit MQ
  – Multicast Communications:
    • Application Layer Messaging
      • Epidemic Protocols
SECTION 6.0: WHAT DO WE MEAN BY DISTRIBUTED SYSTEMS?
Distributed Systems

• What?
  – Multiple networked cooperating computers
  – Eg: Internet E-Mail, Google MapReduce, Dropbox, etc.

• Why?
  – to connect physically separate entities
  – to achieve security via physical isolation
  – to tolerate faults via replication at separate sites
  – to increase performance via parallel CPUs/mem/disk/net

• Drawbacks
  – Complex, hard to debug
  – New classes of problems, e.g. partial failure (did he accept my e-mail?)
  – Lamport: A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

• Advice: don't distribute if a single system will work

• Question: What is the difference between a distributed system and a decentralized one?
Fallacies of Distributed Computing*

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

• **Question:** What is the underlying theme?

*https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing*
System Design Choices

E.g. building a shared filesystem like Dropbox

- Architecture
- Implementation
- Performance
- Fault tolerance
- Consistency
System Design Choices - Architecture

- **Choice of interfaces**
  - Typed (rigid) vs Generic

- **Single machine room or unified wide area system?**

- **Client/server or peer-to-peer?**

- **Interacts w/ performance, security, fault behavior.**
System Design Choices - Implementation

- How do clients/server communicate?
- Can you hide network stuff from application logic?
  - Challenges: dealing with failures, performance, security, management, ...
- Do you use a structuring framework/middleware like RPC, RMI, MPI, MapReduce, etc
System Design Choices - Performance

- Distribution can hurt: network b/w and latency bottlenecks
  - CPU vs I/O speed
- Distribution can help: parallelism, pick near server
- Ideal: scalable design, rarely perfect -> only scales so far
  - Load balancing
System Design Choices – Fault Tolerance

- 1,000s servers => some fail
- Can I use my files if there's a failure?
  - Cannot distinguish network failure from server failure
- Replicate the data on multiple servers?*
- Opportunity: operate from two "replicas" independently if partitioned?
- Opportunity: can 2 servers yield 2x availability AND 2x performance?
- Replicate the requests?

System Design Choices – Consistency

• Specify a contract with apps or users about the meaning of operations e.g. "read yields most recently written value"

• Hard due to partial failure, replication/caching, concurrency

• Problem: keep replicas identical

• Problem: clients may see updates in different orders
SECTION 6.1:
MIDDLEWARE/FRAMEWORKS FOR DISTRIBUTED SYSTEMS
Role of Middleware

• **Observation**
  – Role to provide common services/protocols in Distributed Systems
  – Can be used by many different distributed applications

• **Middleware Functionality**
  – (Un)marshalling of data: necessary for integrated systems
  – Naming protocols: to allow easy sharing, discovery of resources
  – Security protocols: for secure communication
  – Scaling mechanisms, such as for replication & caching (e.g. decisions on where to cache etc.)
  – A rich set of comms protocols: to allow applications to transparently interact with other processes regardless of location.
Classification of Middleware

Classify middleware technologies into the following groups:

1. **Bog-standard Sockets**
   - The basis of all other middleware technologies.

2. **RPC – Remote Procedure Call (more later)**
   - RPCs provide a simple way to distribute application logic on separate hosts.
3. **TPM - Transaction Processing Monitors:**
   - TPMs are a special form of MW targeted at distributed transactions.

4. **DAM - Database Access Middleware:**
   - DBs can be used to share & communicate data between distributed applications.
Classification of Middleware (/3)

5. Distributed Tuple:
   - Distributed tuple spaces implement a distributed shared memory space.

6. DOT (Dist Object Technology) / OOM (Object-Oriented M/w):
   - DOT extends the object-oriented paradigm to distributed applications.
7. **MOM (Message Oriented Middleware):**

- In MOM, messages are exchanged asynchronously between distributed applications (senders and receivers).

8. **Web services:**

- Web services expose services (functionality) on a defined interface, typically accessible through the web protocol HTTP.
9. **Peer-to-peer middleware:**
   - Have seen above how MW often follows particular *architectural style*.
   - In P2P, each peer has equal role in comms pattern (eg routing, node mgmt).
   - More on this later...

10. **Grid middleware:**
    - Provides computation power services (registration, allocation, de-allocation) to consumers.
Summary of Communications Middleware

- Essentially a range of types of communications middleware
- All can be used to implement others, all are suited to different cases
  - All carry some payload from one side to another <with details>
  - Some of these payloads are ‘active’ and some are ‘passive’
  - Also differ in granularities and whether synchronous or not.
SECTION 6.2: COMMUNICATION IN DISTRIBUTED SYSTEMS
Terminology for Distributed Communications

- **Persistent Communications:**
  - Once sent, the “sender” stops executing.
  - “Receiver” need not be in operation – communications system buffers message as required until delivery can occur.

- **Transient Communications:**
  - Message only stored as long as “sender” & “receiver” are executing.
  - If problems occur either deal with them (sender is waiting) or message is simply discarded ...
Persistence & Synchronicity in Communications

a) Persistent asynchronous communication
b) Persistent synchronous communication

Diagram:
- **(a)**: Persistent asynchronous communication
  - A sends message and continues
  - B is not running
  - Buffering
  - A stopped running
  - B starts and receives message

- **(b)**: Persistent synchronous communication
  - A sends message and waits until accepted
  - Message is stored at B's location for later delivery
  - Accepted
  - A stopped running
  - B is not running
  - B starts and receives message
  - Time
c) **Transient asynchronous communication**

d) **Receipt-based transient synchronous communication**
Persistence & Synchronicity in Communications (/3)

e) Delivery-based transient synchronous communication at message delivery

f) Response-based transient synchronous communication
SECTION 6.3: REMOTE PROCEDURE CALL (RPC)
Remote Procedure Call (RPC)

- **Rationale:** Why RPC?
- **Distribution Transparency:**
  - Send/Receive don’t conceal comms at all – need to achieve *access* transparency.
  - Want to simplify programming model by hiding most details
- **Answer:** Make a client function call on a server just like a local function call:
  - RPC allows programs to communicate by calling procedures on other machines.
  - Ideally client and server programmers don’t need to worry about network
    - This is a fantasy
- **Mechanism**
  - When a process on machine A calls a procedure on machine B, calling process on A is suspended,
  - Execution of the called procedure takes place on B.
  - Info ‘sent’ from caller to callee in parameters & comes back in result.
  - No message passing at all is visible to the programmer.
  - Application developers familiar with simple procedure model.
RPC Concept

- RPC aims for this level of transparency

  Client:
  \[ z = f(x, y) \]

  Server:
  \[
  f(x, y) \{
  \text{compute}
  \text{return } z
  \}
  \]

- RPC Sequence Diagram:

- Software structure
**Basic RPC Operation**

1. Client procedure calls client stub
2. Stub builds message, calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to dispatcher.
5. Dispatcher unpacks parameters, looks up handler from a table (fn id, server). Sends to server handler.
6. Server works, returns result to stub.
7. Stub builds message, calls local OS.
8. OS sends message to client’s OS.
9. Client OS gives message to client stub.
10. Stub unpacks result, returns to client.
**RPC: Parameter Passing**

- **Parameter marshalling:** format data into packets

  More than just wrapping parameters into a message:
  - Client/server machines may have different data representations (e.g. byte ordering)
  - Wrapping parameter means converting value into byte sequence
    - How are basic data values represented (integers, floats, characters)?
    - How are complex data values represented (arrays, unions)?
    - How can you represent dynamic memory structures like Linked List?
    - How about a reference?

  => RPC must support transforming data into machine-dependent representations.
**RPC: Parameter Passing (/2)**

- **Assumptions Regarding RPC Parameter Passing:**
  - Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
  - All data to be operated on is passed by parameters. Excludes passing references to (global) data.

- **Conclusion**
  - Full access transparency cannot be realized

- **Observation:**
  - A remote reference mechanism enhances access transparency: Remote reference offers unified access to remote data
  - Remote references can be passed as parameter in RPCs
RPC Limitations

• Name/address binding: how does the client know who to talk to?
  – What if there are multiple servers?

• Thread management/dealing with asynch nature of network calls
  – If >1 RPC call outstanding, need to match responses to requests

• What to do about failures?
  – What if the network/server is just slow instead of down?
  – Failure for RPC = you never see a response
    • NB Maybe failure happened *after* the server responded
RPC Failure Modes

• What could go wrong:
  – Request from client -> server lost
  – Reply from server -> client lost
  – Server crashes after receiving request
    • Before processing the request
    • After processing the request
• Client crashes after sending request
Partial failures – A New Kind of Failure for Distributed Systems

• In local computing:
  – if machine fails, application fails
  – (fate sharing!)

• In distributed computing:
  – if a machine fails, part of application fails
  – cannot tell the difference between a machine failure and network failure

• How to make partial failures transparent to client?
Strawman solution

• Make remote behavior identical to local behavior:
  – Every partial failure results in complete failure
    • You abort and reboot the whole system
  – You wait patiently until system is repaired

• Problems with this solution:
  – Many catastrophic failures
  – Clients block for long periods
  – System might not be able to recover
  – (Also, why do distribution in the first case if during failures the distributed system acts like a non-distributed system)
Real solution: break transparency

- Possible semantics for RPC:
  - Exactly-once (what local procedure calls provide)
    - Impossible in practice (side effects)
  - At least once:
    - Ideal for idempotent operations
  - At most once
    - Zero, don’t know, or once
  - Zero or once
    - Transactional semantics (databases!)
    - Requires solving distributed atomic commitment (hard!)
Exactly-Once?

- Sorry - no can do *in general*
  - Side effects
- Imagine that message triggers an external physical thing (say, a drone delivers a burrito to an office)
- The drone could crash immediately before or after delivery and lose its state. Don’t know whether the burrito was delivered or not.
Real solution: break transparency

• 2 main options: At-least-once, At-most-once
• **At-least-once**: Just keep retrying on client side until you get a response.
  – Client must set a timer for request
    • If timer expires => re-send the request
    • After N re-tries => abandon
  – Server just processes requests as normal, doesn’t remember anything. Simple!
• Simple problem (non-replicated key/value server)
  – Client sends Put(a). Server gets request, but network drops reply. Client sends Put(a) again
  – Should server respond "yes"? or "no"?
  – What if it isn't a put but "deduct $10 from bank account"
Real solution: break transparency – Better solution

• **At-most-once**: Server might get same request twice...
  – Server must be able to identify requests
    • If it has seen it before, do not execute handler, just re-send the response
  – Must re-send *previous* reply and not process request (implies: keep cache of handled requests/responses)
  – Strawman: remember *all* RPC IDs handled. -> Ugh! Requires infinite memory.
  – Real: Keep sliding window of valid RPC IDs, have client number them sequentially.

• Question: how do you allocate unique IDs?
What if an At most Once Server Crashes?

• If at-most-once duplicate info is in memory, server will forget
  – =&gt; will accept duplicate requests!
• Maybe it should write the duplicate info to disk?
  – Performance?
• Maybe a replica server should also replicate duplicate info?
RPC Example: Distributed Computing Environment (DCE)

- **Writing A Client and Server in DCE:**

  ![Diagram of RPC Example in DCE]

  - Uuidgen
  - Interface definition file
  - IDL compiler
  - Client code
  - Client stub
  - Header
  - Server stub
  - Server code
  - C compiler
  - Client object file
  - Client stub object file
  - Server stub object file
  - Server object file
  - Linker
  - Runtime library
  - Client binary
  - Server binary
DCE Client to Server Binding

- **Registration & Discovery:**
  - Server registration enables client to locate server and bind to it.
  - Server location is done in two steps:
    1. Locate the server’s machine.
    2. Locate the server on that machine.

![Diagram showing client-server interaction](image-url)
Summary:
Expose remoteness to client (even though you don’t want to)

• Expose RPC properties to client, since you cannot hide them

• Application writers have to decide how to deal with partial failures
  – Consider: E-commerce application vs. game
  – Some frameworks/middleware are application-specific
References

• Martin Kleppmann, Designing Data-Intensive Applications, O’Reilly, 2017