CS244b – Distributed Systems

Instructor: David Mazières
CAs: Hiroshi Mendoza and Seo Jin Park
Stanford University

Outline
1. Administrivia
2. Remote procedure call
3. Consensus in asynchronous systems

Administrivia
• Class web page: http://cs244b.scs.stanford.edu/
• All handouts and lecture notes on line
  - Please print them out yourselves
• Each class will involve discussing papers
  - Print, read the papers before class
  - Class (except SCPD) and mailing list participation counts for grade
  - We will post discussion notes afterwards
• Homework questions before most classes (see syllabus)
  - Turn in on paper at start of class (SCPD can submit remotely)
• Staff mailing list: cs244b-staff@scs.stanford.edu
  - Please email all staff rather than individual members

Programming assignments
• Two solo programming assignments in C++11 (or later)
  - Goal: familiarize you with RPC, consensus, consistency
• Final project
  - Perform a small research project in teams of 1–3 students
  - Welcome to use code from first labs
  - Use ideas from papers we’ve discussed in class
  - Turn in short paper, make presentation
• Presentations: Monday, December 11, (12:30–6:30pm??)
  - Present project in mini-conference
  - We will serve food
  - Might need second slot, student PC, or parallel tracks given enrollment

Grading
• Grading based on four factors:
  1. Class participation and homework questions (c, where 0 ≤ c ≤ 1)
  2. Midterm and final quizzes (q)
  3. Lab assignments (l)
  4. Final project paper & presentation (p)
• Combined as follows (subject to adjustment):
  - Compute average: \( a = q/4 + l/4 + p/2 \).
  - Adjusted score is: if \( p > a \) then \( cp + (1 - c)a \) else \( a \).
• Final project is most important component
• With participation, good project overrides bad quiz/lab

Why study distributed systems?
• Most real systems are actually distributed systems
• If you want fault-tolerance or scalability
  - Must replicate across multiple machines
• If you want systems that span administrative realms
  - Web sites, peer-to-peer systems, communication systems

Why study distributed systems?
• Most real systems are actually distributed systems
• If you want fault-tolerance or scalability
  - Must replicate across multiple machines
• If you want systems that span administrative realms
  - Web sites, peer-to-peer systems, communication systems
Class topics

- Distributed programming models
- Dealing with failure, including Byzantine failure
- Scalability
- Techniques: Consensus, Replication, Consistency...
- Case studies: production systems at Google, Amazon, ...
- Byzantine-agreement-based Blockchain mechanisms

Outline

1. Administrivia
2. Remote procedure call
3. Consensus in asynchronous systems

Remote procedure call (RPC)

- Procedure calls are a well-understood mechanism
  - Transfer control and data on single computer
- RPC’s goal is to make distributed programming look like as much as possible like normal programming
  - Code libraries provide APIs to access functionality
  - RPC servers export interfaces accessible through local APIs
  - See [Birrell] for good description of one implementation
- Implement RPC through request-response protocol
  - Procedure call generates network request to server
  - Server return generates response
- Good example of how distributed systems differ...

Procedure vs. RPC

- Consider the following ordinary procedure:
  bool add_user(string user, string password);
- Possible return values: true, false
- Now say you have an RPC version
  - Must somehow set up connections, bind to server, think about authentication, etc., but ignore all that for now
- What are the possible return values of add_user RPC?

  1. true
  2. false
  3. “I don’t know”

RPC Failure

- Normal procedure call has fate sharing
  - Single process: if callee fails, caller fails, too
- RPC introduces more failure modes
  - Machine failures at only one end (caller/callee)
  - Communication failures
- Result: RPCs can return “failure” instead of results
- What are the possible outcomes after failure?
  - Procedure did not execute
  - Procedure executed once
  - Procedure executed multiple times
  - Procedure partially executed
- Many systems aspire to “at most once semantics”
Implementing at most once semantics

- **Danger: Request message lost**
  - Client must retransmit requests when it gets no reply

- **Danger: Reply message may be lost**
  - Client may retransmit previously executed request
  - Okay if operations are idempotent, but many are not (e.g., process order, charge customer, …)
  - Server must keep “replay cache” to reply to already executed requests

- **Danger: Server takes too long to execute procedure**
  - Client will retransmit request already in progress
  - Server must recognize duplicate—can reply “in progress”

Server crashes

- **Danger: Server crashes and reply lost**
  - Can make replay cache persistent—slow
  - Can hope reboot takes long enough for all clients to fail

- **Danger: Server crashes during execution**
  - Can log enough to restart partial execution—slow and hard
  - Can hope reboot takes long enough for all clients to fail

- **Can use “cookies” to inform clients of crashes**
  - Server gives client cookie which is time of boot
  - Client includes cookie with RPC
  - After server crash, server will reject invalid cookie

Parameter passing

- Trivial for normal procedure calls
- RPC must worry about different data representations
  - Big/little endian
  - Size of data types
- RPC has no shared memory
  - No global variables
  - How to pass pointers
  - How to garbage-collect distributed objects
- How to pass unions over RPC?

Interface Definition Languages

- Idea: Specify RPC call and return types in IDL
- Compile interface description with IDL compiler. Output:
  - Native language types (e.g., C/Java/C++ structs/classes)
  - Code to marshal (serialize) native types into byte streams
  - Stub routines on client to forward requests to server
- Stub routines handle communication details
  - Helps maintain RPC transparency, but...
  - Still have to bind client to a particular server
  - Still need to worry about failures

C++ RPC-related systems in use today

- XML or JSON over HTTP – no IDL, hard to parse
- **Cereal** – C++11 structure serializer
- **Google protobufs, Apache Thrift**
  - Compact encoding, defensively coded (protobufs)
  - Good support for incrementally evolving messages
  - Not complete system (protobufs), complex encoding, not C++11
- **Apache Avro** – self-describing messages contain schema
- **Cap’n Proto, Google FlatBuffers**
  - Same representation in memory and on wire, very fast
  - Less mature, non-deterministic wire format, bigger attack surface
- **XDR (+ RPC) – used by Internet standards such as NFS**
  - Simple, good features (unions, fixed- and variable-size arrays, …)
  - Big endian, binary but rounds everything to multiple of 4 bytes

Case study: XDR

```c
enum MyEnum { NO, YES, MAYBE };

struct MyMessage {
  string name<16>; /* up to 16 characters */
  string desc<>; /* up to 2^-32-1 characters */
  opaque cookie<>; /* 8 bytes (fixed) */
  opaque sig<16>; /* 0-16 bytes (variable-length) */
  unsigned int u; /* Unsigned 32-bit integer */
  hyper ii; /* Signed 64-bit integer */
  MyEnum me; /* Another user-defined type */
  int ia[5]; /* Fixed-length array */
  int iv<>; /* Variable length array */
  int ivl<5>; /* Up to 5 ints */
  MyMessage *mep; /* optional MyMessage (or NULL) */
};

typedef MyMessage *OptionalMyStruct;
```
XDR base types

- All numeric values encoded in big-endian order
- int, unsigned [int], all enums: 4 bytes
- bool: equivalent to “enum bool { FALSE, TRUE }”
- hyper, unsigned hyper: 8 bytes
- float, double, quadruple: 4-, 8-, or 16-byte floating point
- opaque bytes[Len] (fixed-size)
  - Encoded as content + 0–3 bytes padding to make size multiple of 4
- string s<MaxLen>, opaque a<MaxLen> (variable-size)
  - 4-byte length + content + (0–3 bytes) padding

XDR containers and structs

- (Fixed) arrays – MyType var[n]
  - Encoded as n copies of MyType
- Vectors – MyType var<> or MyType var<n>
  - Can hold variable number (0–n) MyTypes
  - Encoded as 4-byte length followed by that many
  - Empty maximum length means maximum length $2^2 - 1$ MyTypes
- Optional data – MyType *var
  - Encoded exactly as MyType var<1>
  - Note this means single “present” bit consumes 4 bytes
- struct – each field encoded in turn

XDR union types

union type switch (simple_type which) {
  case value_A:
    type_A varA;
  case value_B:
    type_B varB;
  /* ... */
  default:
    void;
}

- Must be discriminated, unlike C/C++
- simple_type must be [unsigned] int, bool, or enum
- Wire representation:
  - 4-bytes for which + encoding of selected case
  - Special void type encoded as 0 bytes

The identity function

- One of the simplest functions is the identity function
- E.g., in Haskell: id x = x
- In C++:
  ```
  template<typename T> inline T &&
  id(T &&t) {
    return static_cast<T &&>(t);
  }
  ```
- The distributed equivalent turns out to be much harder
  - Problem: agents might not start with the same input
  - So to agree on output, must somehow pick one of the inputs

Outline

1. Administrivia
2. Remote procedure call
3. Consensus in asynchronous systems

Asynchronous systems

- A theoretical model for distributed systems
  - Consists of a set of agents exchanging messages
  - No bound on message delays
  - No bound on the relative execution speed of agents
  - For convenience, model internal events such as timeouts as special messages, so the “network” controls all timing
- Can’t distinguish failed agent from slow network
- Idea of model is to be conservative
  - Want robustness under any possible timing conditions
  - E.g., say backhoe tears fiber, takes a day to repair
  - Could see messages delays a billion times more than usual

1 Unrelated to “asynchronous IO” as used in event-driven systems.
The consensus problem

- Goal: For multiple agents to agree on an output value
  - Each agent starts with an input value
    - Agents’ inputs may differ; any agent’s input is okay to output
  - Agents communicate following some consensus protocol
    - Use protocol to agree on one of the agent’s input values
  - Once decided, agents output the chosen value
    - Output is write-once (an agent cannot change its value)

Bivalent states

- Recall agents chose value 9 in last example
  - But a network outage could look like agent 2 failing
  - If fault-tolerant, Agents 1 & 3 might decide to output 7
  - Once network back, Agent 2 must also output 7

Definition (Bivalent)
An execution of a consensus protocol is in a bivalent state when the network can affect which value agents choose.
### Bivalent states

- Recall agents chose value 9 in last example
- But a network outage could look like agent 2 failing
  → If fault-tolerant, Agents 1 & 3 might decide to output 7
- Once network back, Agent 2 must also output 7

#### Definition (Bivalent)
An execution of a consensus protocol is in a bivalent state when the network can affect which value agents choose.

### Univalent and stuck states

#### Definition (Univalent, Valant)
An execution of a consensus protocol is in a univalent state when only one output value is possible. If that value is i, call the state i-valent.

#### Definition (Stuck)
An execution of a [broken] consensus protocol is in a stuck state when one or more non-faulty nodes can never output a value.

- Recall output is write once and all outputs must agree
  - Hence, no output is possible in bivalent state
- If an execution starts in a bivalent state and terminates, it must at some point reach a univalent state

### FLP intuition

- Consider a terminating execution of a bivalent system
- Let m be last message received in a bivalent state
  - Call m the execution’s deciding message
  - Any terminating execution requires a deciding message
- Suppose the network had delayed m
  - Other messages could cause transitions to other bivalent states
  - Then, receiving m might no longer lead to a univalent state
  - In this case, we say m has been neutralized

#### Overview of FLP proof.
1. There are bivalent starting configurations
2. The network can neutralize any deciding message
3. Hence, the system can remain bivalent in perpetuity

### There exists a bivalent state

#### Scenario A
- Assume you could have liveness with an agent failure
  → If all inputs 0, correct agents must eventually output 0
  - Similarly, if all inputs 1, correct agents must eventually output 1
- Now say we start flipping one input bit at a time
- Find 0- and 1-valent states differing at only one input
  - Suppose node with this differing input fails
  - By assumption, the system nonetheless reaches consensus
  - Hence output depends on network; at least one state was bivalent
There exists a bivalent state

### Scenario A

- Assume you could have liveness with an agent failure
- If all inputs 0, correct agents must eventually output 0
  - Similarly, if all inputs 1, correct agents must eventually output 1
- Now say we start flipping one input bit at a time
  - Find 0- and 1-valent states differing at only one input
    - Suppose node with this differing input fails
    - By assumption, the system nonetheless reaches consensus
    - Hence output depends on network; at least one state was bivalent

### Scenario B

- Assume you could have liveness with an agent failure
- If all inputs 0, correct agents must eventually output 0
  - Similarly, if all inputs 1, correct agents must eventually output 1
- Now say we start flipping one input bit at a time
  - Find 0- and 1-valent states differing at only one input
    - Suppose node with this differing input fails
    - By assumption, the system nonetheless reaches consensus
    - Hence output depends on network; at least one state was bivalent

---

**Any message can be neutralized**

- Let $m$ be a deciding message for value 0 from state $b$
  - Consider a message schedule from $b$ to a 1-valent state
  - If $m$ is on the path, it leads to a bi-valent state
  - If $m$ is not on the path, append it to the (1-valent) path
- Apply $m$ to each node on the path
  - Either $m$ will lead to a bi-valent state, or it will produce differing univalent states on adjacent nodes $c_0$ and $c_1$

---

- Let $m$ be a deciding message for value 0 from state $b$
  - Consider a message schedule from $b$ to a 1-valent state
  - If $m$ is on the path, it leads to a bi-valent state
  - If $m$ is not on the path, append it to the (1-valent) path
- Apply $m$ to each node on the path
  - Either $m$ will lead to a bi-valent state, or it will produce differing univalent states on adjacent nodes $c_0$ and $c_1$

---

- Let $m$ be a deciding message for value 0 from state $b$
  - Consider a message schedule from $b$ to a 1-valent state
  - If $m$ is on the path, it leads to a bi-valent state
  - If $m$ is not on the path, append it to the (1-valent) path
- Apply $m$ to each node on the path
  - Either $m$ will lead to a bi-valent state, or it will produce differing univalent states on adjacent nodes $c_0$ and $c_1
Any message can be neutralized

- Let \( m \) be a deciding message for value 0 from state \( b \)
- Consider a message schedule from \( b \) to a 1-valent state
  - If \( m \) is on the path, it leads to a bi-valent state or to a 1-valent one
  - If \( m \) is not on the path, append it to the (1-valent) path

→ Apply \( m \) to each node on the path
  - Either \( m \) will lead to a bi-valent state, or it will produce differing univalent states on adjacent nodes \( c_0 \) and \( c_1 \)

Any message can be neutralized

- Let \( m' \) be the message that transitions between \( c_0 \) and \( c_1 \)
  - If \( m, m' \) received by different agents, order won't matter
    - But if delivering both messages yields a 1-valent state, delivering just \( m \) can’t yield a 0-valent state
  - Hence, either \( m \) is neutralized at \( c_1 \), or same agent \( A \) received \( m \) and \( m' \), making order significant
  - Yet if \( A \) slow after \( c_0 \), system must terminate without it

Consider a run that terminates without \( A \)
  - Let \( x_1, \ldots, x_n \) be the messages received (by nodes other than \( A \))
  - Let \( e \) be a univalent state reached during the run

→ Deliver \( x_1, \ldots, x_n \) to terminating states after \( m \)
  - Since \( m \) and \( x \) received by different nodes, can re-order
  - Means \( e \) not univalent (leads to both 0- and 1-valent states)!

Contradiction means \( m \) must be neutralized somewhere
Coping with FLP

- This class will cover
  - Many systems that require consensus
  - Many techniques for consensus

- Safety is generally pretty important

- But can reasonably weaken liveness requirement
  - Termination not guaranteed doesn’t mean it won’t happen
  - If your algorithm prevents completely stuck states
    …can often make it terminate “in practice”

- Can weaken asynchronous system assumption

- Can make agents non-deterministic
  - Have all nodes flip a coin to pick value—might all pick same value
  - Make it intractable for network to “guess” pathological delivery
    100% accurately in perpetuity