Engineering Robust Server Software

API/Protocol/Server Design Ideas
Important API/Protocol/Server Design Ideas

- Design for failure
- Design for asynchronous interfaces
  - What does this mean?
- Don't trust anyone or anything
How You Want Things: Synchronous

I need a thing
How You Want Things: Synchronous

I need a thing

Here is the thing
How You Want Things: Synchronous

I need a thing

Here is the thing

I need a different thing
How You Want Things: Synchronous

I need a thing
Here is the thing

I need a different thing
Here is the other thing
Synchronous Processing

• Synchronous processing is straight forward:

```python
connection.send_message(request);
response = connection.read_response();
do_whatever(response);
```

but…
Difficulty With Synchronous Behavior

```java
connection.send_message(request);
response = connection.read_response();

do_whatever(response);
```

Blocked waiting for response all this time
( Thread can't do anything else )

Here is the thing
Also May Not Get Response

I need a thing
Think/Pair/Share

- Send + Receiving:
  - Take a moment to think up approaches for how we can receive data
    - **Constraint**: cannot block this thread waiting for response!
    - **Pros and Cons** of approach?
    - **Bonus**: ties to names/concepts from 550?
Receiving

- (1) Polling:
  - Send just does:
    ```cpp
collection.send_message(request);
collections.push_back(connection);
```
  - Then we periodically try to receive:
    ```cpp
    for (auto &c: connections) {
        if (c.is_response_ready()) {
            response = connection.read_response();
            do_whatever(response);
        }
    }
    ```
Receiving

● (2) Interrupts?
  ● What is user-land equivalent of interrupts?
Receiving

- (2) Interrupts?
  - What is user-land equivalent of interrupts?  **Signals**
  - This is not something you can do easily.
  - TCP supports urgent data (delivers SIGURG)
    - Sender must mark data urgent
    - Not commonly used
      - You could have sender do in your own photo
      - but don't expect to e.g., have web clients mark all data urgent
  - …but similar idea?
Receiving

- (3) Spawn Another Thread To Receive: Pros and Cons?
  - Send just:
    ```java
    connection.send_message(request);
    spawn_thread(receive_data, connection);
    ```
  - Receive is done in receive_data on other thread:
    ```java
    //blocking, but on its own thread
    response = connection.read_response();
    do_whatever(response);
    ```
Receiving

- (3) Dedicated receive threads?  Pros and Cons?
  - Pre-spawn some threads to receive
  - Sender communicates state (what to do) to these threads
Also May Not Get Response

I need a thing

Failure->
- Power failure
- Crash
- Network disconnected
- ...

Andrew Hilton / Duke ECE
Also May Not Get Response

I need a thing

Here is the thing

Network problems

X
Also May Not Get Response

I need a thing

But doesn't TCP guarantee delivery?

Here is the thing

X
Also May Not Get Response

I need a thing

But doesn't TCP guarantee delivery?

Here is the thing

Here is the thing

Here is the thing

I give up
No Way To Tell Where Failure Happened

- We cannot tell the difference between
  - Data not reaching the receiver
  - Data reaching the receiver, but ACK not reaching us
- Is that a big deal?
Data Did Not Reach Receiver

Transfer 3 bit coins from Alice to Bob

Transfer 3 bit coins from Alice to Bob

Transfer 3 bit coins from Alice to Bob

Transfer 3 bit coins from Alice to Bob
ACK Did Not Reach Sender

Transfer 3 bit coins from Alice to Bob

X

X

X

X
Two Generals Problem

• Famous problem: two generals
Two Generals Problem

- I have a valley with the enemy army camped in it
Two Generals Problem

- We have an army camped on each side, each with its own general
Two Generals Problem

- If both generals attack together, they win
- If either attacks alone, they lose
Two Generals Problem

- One wants to send a message to the other to attack

A,
I will attack at dawn if you will
–L
Two Generals Problem

A,
I will attack at dawn if you will
—L

- But that messenger might get captured...
Two Generals Problem

- So now we need an acknowledgement...

A, I will attack at dawn if you will.

L, Yes, I will attack.

—A 

—L
Two Generals Problem

- But the ACK could get lost…

A, I will attack at dawn if you will.

–L

L, Yes, I will attack.

–A
Two Generals Problem

- Now our armies will be defeated…

I never got an ACK. My message was lost. I should **NOT** attack.

I ACKed her message. I **MUST** attack.
Two Generals Problem

- Problem: we can never tell if our ACK got through
  - ACK the ACKs? Need infinite number…

I never got an ACK. My message was lost. I should **NOT** attack

I ACKed her message. I **MUST** attack.
No Way To Tell Where Failure Happened

- We cannot tell the difference between
  - Data not reaching the receiver
  - Data reaching the receiver, but ACK not reaching us
- Why is this such a big deal?
- What does this mean about how you need to design software?
  - You all tell me…
Would Like "Exactly Once,"...but...

- We can **never** ensure "exactly once" semantics
  - Which is what we would really like:
    - Ensure that receiver gets our message exactly once

- So what choices do we have?
At Least Once / At Most Once

- **At least once:**
  - We can know if receiver has gotten message at least once
  - Receive an ACK—got it at least once
  - May send need to send multiple times, may receive multiple times

- **At most once:**
  - Send it once
  - May or may not get it—-at most once semantics.
At Least Once / At Most Once

- At least once:
  - We can know if receiver has gotten message at least once
  - Receive an ACK—got it at least once
  - May send need to send multiple times, may receive multiple times

- At most once:
  - Send it once
  - May or may not get it—-at most once semantics.

"But wait" you say...
At Least vs At Most Once

- TCP may send data multiple times (no ACK -> retransmission)
  - We said multiple sending goes with at least once
- But application receives any piece of data at most once
  - Once, unless connection fails
At Least vs At Most Once

- TCP may send data multiple times (no ACK -> retransmission)
  - We said multiple sending goes with **at least once**
- But **application** receives any piece of data **at most once**
  - Once, unless connection fails
- TCP layer has sequence numbers
  - Can identify duplicates, only passes data to application once
At Least vs At Most Once

- TCP may send data multiple times (no ACK -> retransmission)
  - We said multiple sending goes with at least once
- But application receives any piece of data at most once
  - Once, unless connection fails
- TCP layer has sequence numbers
  - Can identify duplicates, only passes data to application once
- This idea is key:
  - Can receive same data multiple times
  - But only act on it once
FSMS + Idempotent Operations

- Two ideas that work together to handle asynchronous + failures
  - Build protocols/APIs around idempotent operations
  - Build implementations with FSMs
Example: Buy 5 widgets

- Online store, user asks to buy 5 widgets
  - What do we need to do to fulfill this request?
- Pair up and think about this…
Buy 5 widgets

1. We accept the request + give it a unique ID
   E.g., 123456789
Buy 5 widgets

2. Send a request to our inventory management system

"req 87654: Reserve 5 widgets for transaction 123456789"
Buy 5 widgets

3. Receive successful acknowledgement

"ack 87654: 5 widgets reserved for 123456789"
Buy 5 widgets

1. Send request to reserve 5 widgets
2. ACK
3. Charge CC $500

4. Send Credit Card Charge request
   External service: probably has its own unique ID?
5. Receive confirmation of successful card charge
Buy 5 widgets

1. Send request to reserve 5 widgets
2. rcv: ACK
3. Charge CC $500
4. rcv: ACK
5. Email user: order is...

6. Inform user of successful purchase
   E.g., send email?
Buy 5 widgets

1. Send request to reserve 5 widgets
   rcv: ACK

2. Charge CC $500
   rcv: ACK

3. Email user: order is...

4. Send request to pack/ship

5. 6. 7. Send request to warehouse to pack/ship

   req: 8888 Send 5 widgets to 123 Fake St for order 123456789
Buy 5 widgets

1. Send request to reserve 5 widgets
2. rcv: ACK
3. Charge CC $500
4. rcv: ACK
5. Email user: order is...
6. Send request to pack/ship
7. ACK
8. Receive ACK

Now done (other systems may still deal with things)
Buy 5 widgets

1. Send request to reserve 5 widgets
   - rcv: ACK

2. Charge CC $500
   - rcv: ACK

3. Email user: order is...

4. Send request to pack/ship
   - ACK

5. 

But is that all there is to it?

8. Receive ACK

Now done (other systems may still deal with things)
Buy 5 widgets

1. Send request to reserve 5 widgets
2. rcv: ACK
3. Charge CC $500
4. rcv: ACK
5. Email user: order is...
6. Send request to pack/ship
7. ACK
8. Receive ACK

No things could go wrong at pretty much any step!

8. Receive ACK
Now done (other systems may still deal with things)
Buy 5 widgets

1. Send request to reserve 5 widgets
   - rcv: ACK

   - Receiver already has req 87654 -> Ignores message

3. Charge CC $500
   - rcv: ACK

4. Email user: order is...

5. ACK

6. Send request to pack/ship

7. ACK

8. ACK

Timeout
Buy 5 widgets

1. Send request to reserve 5 widgets
   
2. Insufficient widgets in warehouse?
   
   Go to error state (inform user, retry later…)

3. Charge CC $500
   
4. rcv: ACK

5. Email user: order is...

6. Send request to pack/ship
   
7. ACK

8. Timeout
   
   rcv: Failure
Buy 5 widgets

Send request to reserve 5 widgets

rcv: ACK

Charge CC $500

rcv: ACK

Email user: order is...

Send request to pack/ship

ACK

4. Card denied? (stolen, insufficient funds, …)

Need to release reservation
Buy 5 widgets

1. Send request to reserve 5 widgets
2. rcv: ACK
3. Charge CC $500
4. rcv: ACK
5. Email user: order is...
6. Send request to pack/ship
7. Timeout
8. ACK

7. Timeout? Retry

What about other failures here?
Buy 5 widgets

No other failures here:
Confirmed/reserved everything in advance
Buy 5 widgets

States 1, 3, 5, 6, and 4F: send message, go to next state
Buy 5 widgets

States 2, 4, 7, 4R: Wait to receive message (timeout -> retry)
Buy 5 widgets

States 8, 2F, 4D: finished.
Importance of Idempotence

Let us look at just this part and see why idempotence is so useful
Normal Operation

Order Processing Server
- add_request(1234, ...)
- update_state(1234, 2)
- update_state(1234, 3)

Warehouse Server
- reserve_item(req);
- Reserve(5, "widget", 123456789)
- ACK
Normal Operation

Order Processing Server

1. add_request(1234,...)
2. update_state(1234,2)
3. update_state(1234,3)

Warehouse Server

87654: Reserve(5, "widget", 123456789)

4. ACK
5. reserve_item(req);
6. 7
7. 8
8. 9

What happens if server fails at any of these points?
Turned off, crashes, …
Normal Operation

1: request not yet accept (not confirmed with client)
Client needs to re-send request (external API should use idempotency)
Normal Operation

Order Processing Server

1. add_request(1234,...)

2. update_state(1234,2)

3. 87654: Reserve(5, "widget", 123456789)

4. ACK

5. update_state(1234,3)

6. 

Warehouse Server

7. reserve_item(req);

8. 

9. 

2: will just send message when server returns
Normal Operation

Order Processing Server

1. add_request(1234,...)

2. update_state(1234,2)

3. 87654: Reserve(5, "widget", 123456789)

4. ACK

5. update_state(1234,3)

Warehouse Server

6. reserve_item(req);

7. 3: will resend when server returns

8. Good thing warehouse will ignore duplicates!
Normal Operation

1. Order Processing Server
   - add_request(1234, ...)

2. 
   - update_state(1234, 2)

3. 
   - reserve_item(req);

4. 
   - update_state(1234, 3)

5. 
   - reserve_item(req);

6. 
   - update_state(1234, 3)

4: depending on when server returns, might miss ACK.
Missed ACK? Will resend after timeout—idempotency helps here!
Normal Operation

Order Processing Server

1. add_request(1234, ...)

2. reserve_item(req)

3. update_state(1234, 2)

4. ACK

Warehouse Server

5. update_state(1234, 3)

5: will resend after timeout

87654: Reserve(5, "widget", 123456789)
Normal Operation

Order Processing Server

1. add_request(1234,...)
2. update_state(1234,2)
3. update_state(1234,3)

Warehouse Server

4. reserve_item(req);
5. reserve_item(req);
6. will just continue to next step after server returns

87654: Reserve(5, "widget", 123456789)
Normal Operation

Order Processing Server
1. add_request(1234,...)
2. update_state(1234,2)
3. update_state(1234,3)

Warehouse Server
4. reserve_item(req);
5. 87654: Reserve(5, "widget", 123456789)
6. ACK
7. will never send ACK. order processor will retry
Normal Operation

Order Processing Server

1. add_request(1234,...)

2. update_state(1234,2)

3. update_state(1234,3)

Warehouse Server

4. reserve_item(req);

5. reserve_item(req);

8: ACK never sent, order processor will retry, duplicate will be ignored

Note order processor can't distinguish 7 vs 8
Normal Operation

Order Processing Server

1. add_request(1234,...)

2. reserve_item(req);

3. update_state(1234,2)

4. ACK

5. update_state(1234,3)

6. 

Warehouse Server

7. 

8. Reserve(5, "widget", 123456789)

9. 

9: done—nothing special happens
Trust No One

- Another important consideration:
  - Never trust clients
- Server should validate **everything**
  - Client can forge any bit of request
  - Trusting client = huge security hole!
- We will talk more about this when we get to security
  - Especially authentication.