Designing for Failure

Jim Posen, Payments @ Coinbase

Who am I?

- ECE/CS 2014 graduate
- Took compilers with Professor Hilton (you should too!)
- Head of Chronicle development team at Duke
- Currently tech lead on payments team at Coinbase

What is Coinbase?

- Consumer platform for buying, selling, and storing digital currency
- Run a regulated digital currency exchange, GDAX
- We integrate with various global banks and payment processors
- Supports multiple blockchain-based currencies, Bitcoin and Ethereum
- This has *major* security implications
    - Coinbase is a counterparty to these trades
    - We store a lot of money on behalf of our customers
    - Anonymity & irreversibility Bitcoin payments makes security imperative

What does robust mean for Coinbase?

- Different applications have different priorities and requirements
  - Financial websites vs social media sites
- There are often tradeoffs between security/consistency, availability/performance, and execution speed
- For Coinbase, security/consistency is the #1 priority
- We need to maintain security while integrating with banks and blockchains of varying levels of reliability

Overview

How do you write robust code that depends on unreliable APIs and and subsystems?

- Modelling applications as state machines
- Asynchronous communication patterns
- Idempotence

Anatomy of a purchase

- Prerequisites
  - User is signed into site
  - User has already linked a credit card
  - User is authorized to make a purchase
- User makes HTTP request to purchase, providing
  - Amount of Bitcoin
  - Price quote offered by Coinbase
  - Destination Bitcoin address

Processing a purchase

On the backend, we want to:

a. Look up and validate user credit card information
b. Charge the credit card using a third-party processor
c. Cancel the purchase if charge fails
d. Send Bitcoin if charge succeeds
e. Notify the user when the Bitcoin transaction is confirmed by the peer-to-peer network

```
@app.route('/purchases', methods=['POST'])
def buy_bitcoin(params):
  user           = User.find(params['user_id'])
  bitcoin_amount = params['amount']
  usd_amount     = params['total']
  destination    = params['address']
  credit_card    = user.primary_credit_card

  valid = purchase_policy.validate_request(
    bitcoin_amount, usd_amount, credit_card
  )

  if not valid:
    return "Invalid request", 422

  success = credit_card_service.charge(
    credit_card, usd_amount
  )

  if not success:
    return "Credit card charge failed", 422

  transaction_id = bitcoin_service.generate_transaction(
    bitcoin_amount, destination
  )

  while !bitcoin_service.transaction_confirmed(transaction_id):
    time.sleep(1)

  return "Purchase complete!"
```

- validate_request and charge should return more than a boolean
- HTTP responses contain no information
- Never sleep in an HTTP request
- These are the obvious things, what else can go wrong?

Understanding failure modes

Credit card processing

- Request can’t be sent (eg. TLS error)
- Request sent, response not received
- Response received with known error code
- Response received with unknown error code
- Response received with ambiguous error code (eg. HTTP 500)

Bitcoin transaction generation

- Same as credit card processing
- Transaction takes a long time to generate or cannot be generated
- Transaction stays unconfirmed for a long time or is never accepted by the network

```python
@app.route('/purchases', methods=['POST'])
def buy_bitcoin(params):
  user           = User.find(params['user_id'])
  bitcoin_amount = params['amount']
  usd_amount     = params['total']
  destination    = params['address']
  credit_card    = user.primary_credit_card

  valid = purchase_policy.validate_request(
    bitcoin_amount, usd_amount, credit_card
  )

  if not valid:
    return "Invalid request", 422

  success = credit_card_service.charge(
    credit_card, usd_amount
  )

  if not success:
    return "Credit card charge failed", 422

  transaction_id = bitcoin_service.generate_transaction(
    bitcoin_amount, destination
  )

  while !bitcoin_service.transaction_confirmed(transaction_id):
    time.sleep(1)

  return "Purchase complete!"
```

Initial state

Request not valid

Request is valid

Credit card charge failed

Credit card charge succeeded

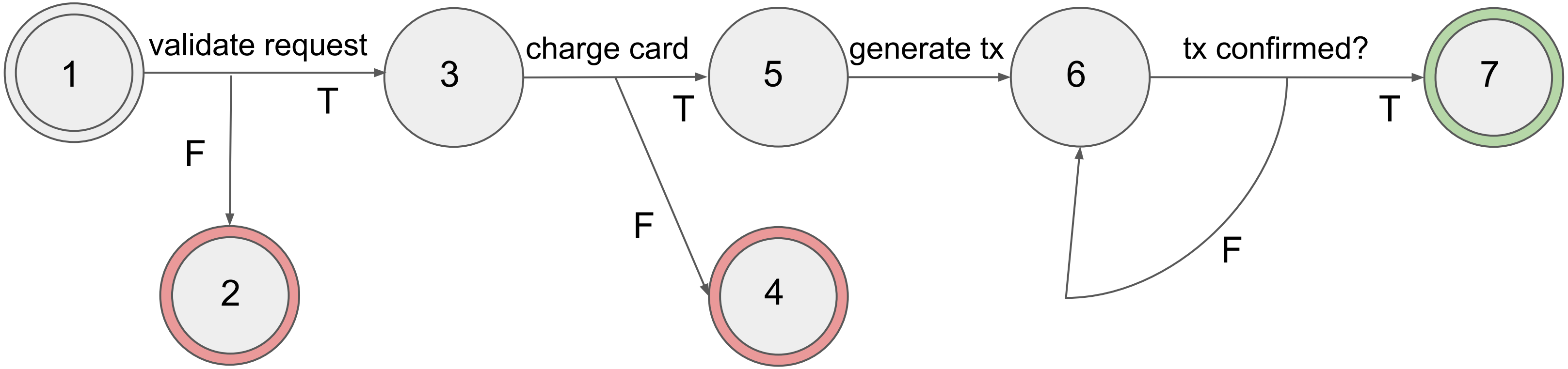
Bitcoin transaction pending

Bitcoin transaction completed

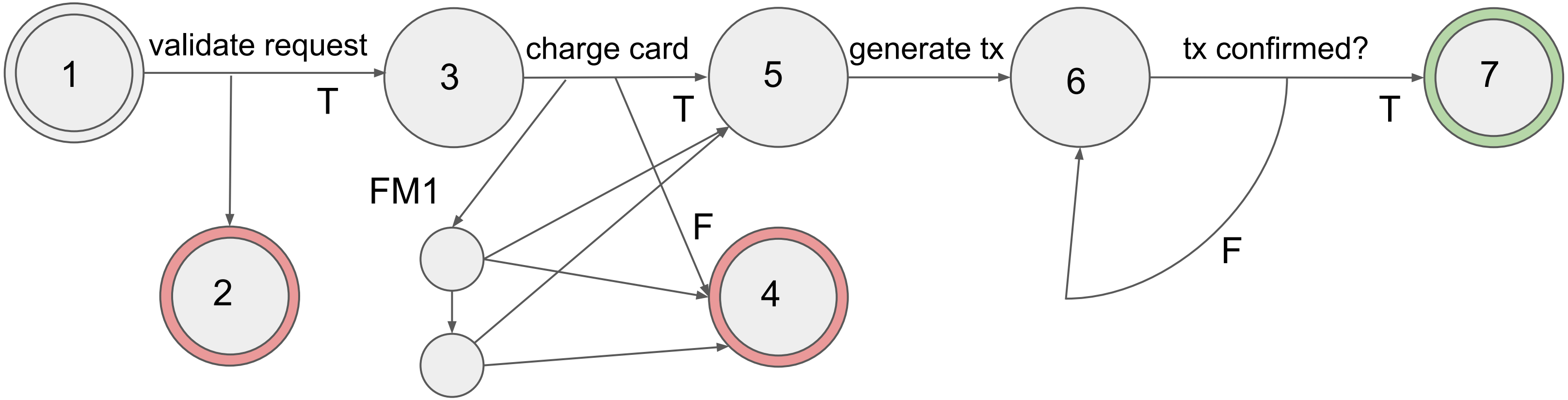
Defining a state machine

- State machines are a good way to model complex logic
- Set of states and conditions required to transition between states
- Each state may have
  - Data that is known by the application
  - An action that may be taken to move the process to completion
  - State transition conditions
- State machine can be represented in database as a status enum and the union of data fields from states

Purchase state machine



Failure mode #1: Request can't be sent



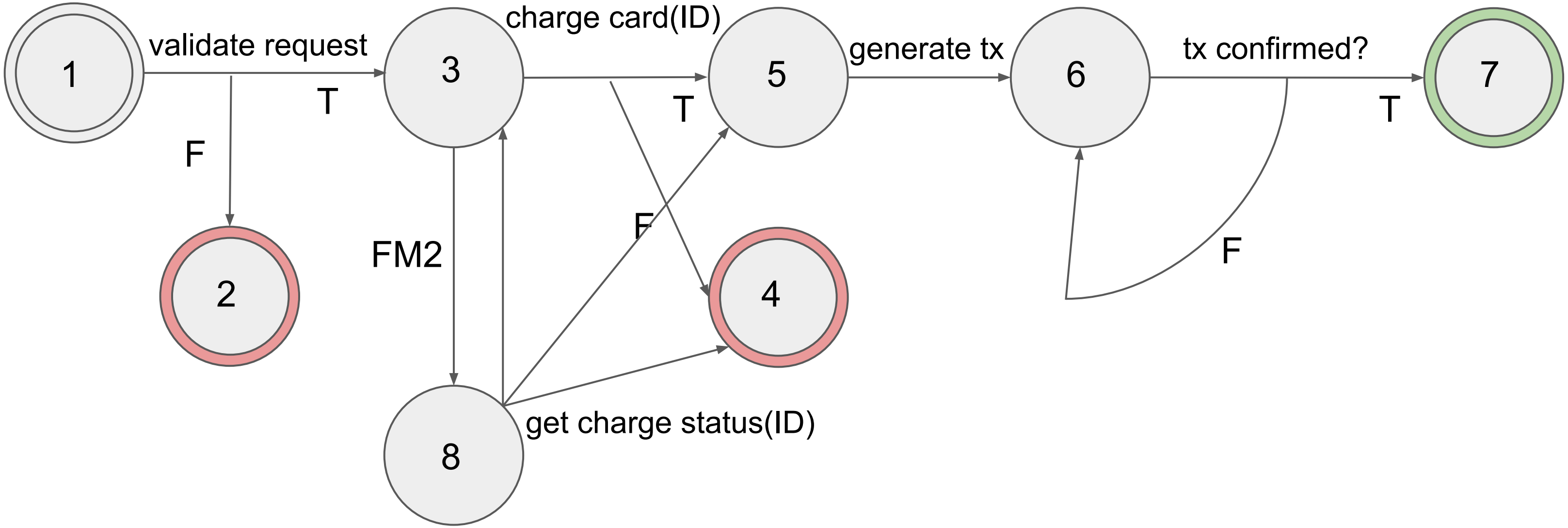
Let's handle this with 2 retries

FM#2: Request sent, response not received

- Say the HTTP connection times out
- Maybe the charge was initiated, maybe not
- What can we do?

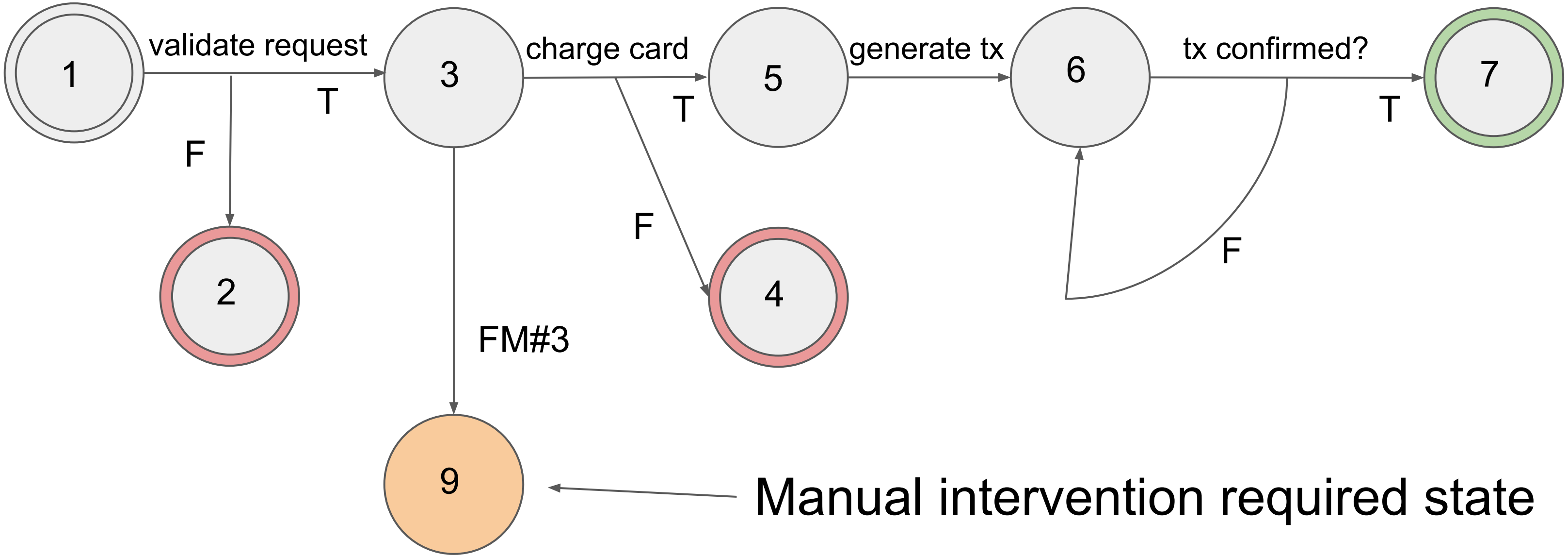
- What if credit card processor API allows you to pass an ID/nonce?
- Generate unique ID before making request as part of state
- Query charge status with ID
- Handle this the same way as ambiguous error codes

FM#2: Request sent, response not received



FM#3: Response with unknown error code

In a system prioritizing consistency over availability, prefer to gracefully stop and alert a developer than do something incorrect



Bitcoin transaction failure modes

- Transaction cannot be generated immediately (not handled yet)
- Transaction is not accepted by the network and must be regenerated
- On average it takes 30 minutes for a Bitcoin transaction to confirm
- We control the interface to the Bitcoin service -- how should we design it?

Asynchronous interfaces

- HTTP request/response model is synchronous
- Client code blocks waiting for response, times out eventually
- Asynchronous operations may take an arbitrary amount of time
- Client code does not block waiting for response
- An asynchronous interface makes sense for Bitcoin processing because we will be able to broadcast and confirm a transaction eventually
  - Asynchronous interface cuts away failure modes
- How to make an asynchronous interface?

Synchronous -> Asynchronous

- Any synchronous call can be made asynchronous by
  - Put message in message queue/buffer with
    - Unique request ID
    - Function name and arguments
    - Calling context
  - Asynchronous processor (separate thread, process, or server) pulls messages from the queue and invokes function
  - Asynchronous processor puts message in return queue with
    - Request ID
    - Function return value

Asynchronous -> Synchronous

- You can always use polling/long polling to wrap a synchronous interface around an asynchronous one
  - The synchronous call may block indefinitely
- Asynchronous processing is more easily parallelizable if messages can be processed out of order
- Use select or language equivalent to poll multiple asynchronous interfaces efficiently

Synchronous vs Asynchronous

Synchronous

- Simpler to implement if processing time is bounded
- Errors can be detected and presented to user immediately
- Can be certain that information is up-to-date

Asynchronous

- Work can be easily parallelized
- Tolerant to unavailability of callee
- Easier to treat a synchronous interface as asynchronous than vice-versa
- May require additional infrastructure, like a message broker

Message queues

- Persistent vs non-persistent queues
  - Redis is commonly used as a persistent queue
  - Unix pipes can be used as non-persistent queues
- Distributed vs non-distributed queues
  - Amazon SQS is a distributed queue
  - Distributed queues delivery messages out of order
- Understanding delivery guarantees
  - At-most-once delivery (fire and forget)
  - At-least-once delivery (retry delivery until acknowledgement)

Idempotence

- An **idempotent operation** is one that when applied multiple times, has the same effect as being applied once
- Idempotent operations can be safely retried
- Implementing idempotent message handlers makes at-least-once delivery workable

Implementing idempotent operations

- Idempotent operation accepts a *unique* identifier or nonce
- Acquire lock on idempotence key
- Read existing status of the operation
- Release lock and exit if operation has already been performed
- Perform operation and write result with idempotence key
- Release lock and exit

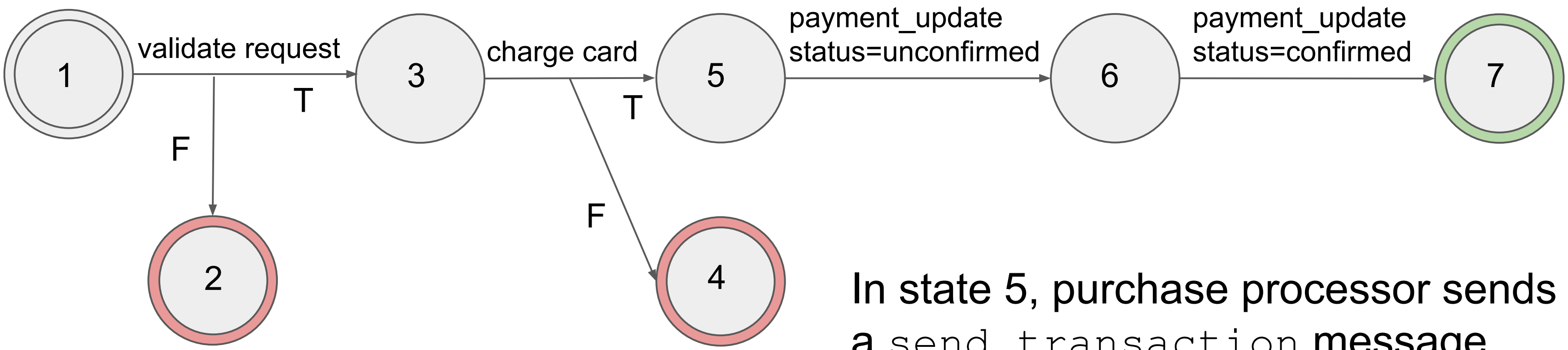
Asynchronous, idempotent Bitcoin processor

- Generate Bitcoin payment ID at the beginning of processing
- Enqueue message `send_transaction(payment_id, amount, address)`
- Bitcoin service/process polls queue and processes *new* payments
- Replies with `payment_update(payment_id, transaction_id, status)`
- If no payment updates (acks) received, you can safely resend messages
- State transition happens on receipt of async message, not as a consequence of caller action

Handling out of order messages

- What if payment_update messages are delivered out of order?
  - We don't want to update a transaction's state with old info
- Sequence numbers to order messages
  - Only process a message if sequence no is higher than last seen
  - TCP uses sequence numbers to order IP datagrams
- Hybrid async/sync processing
  - Async messages notify other process to retrieve transaction status synchronously
  - Availability and latency may be worse, but simpler to implement

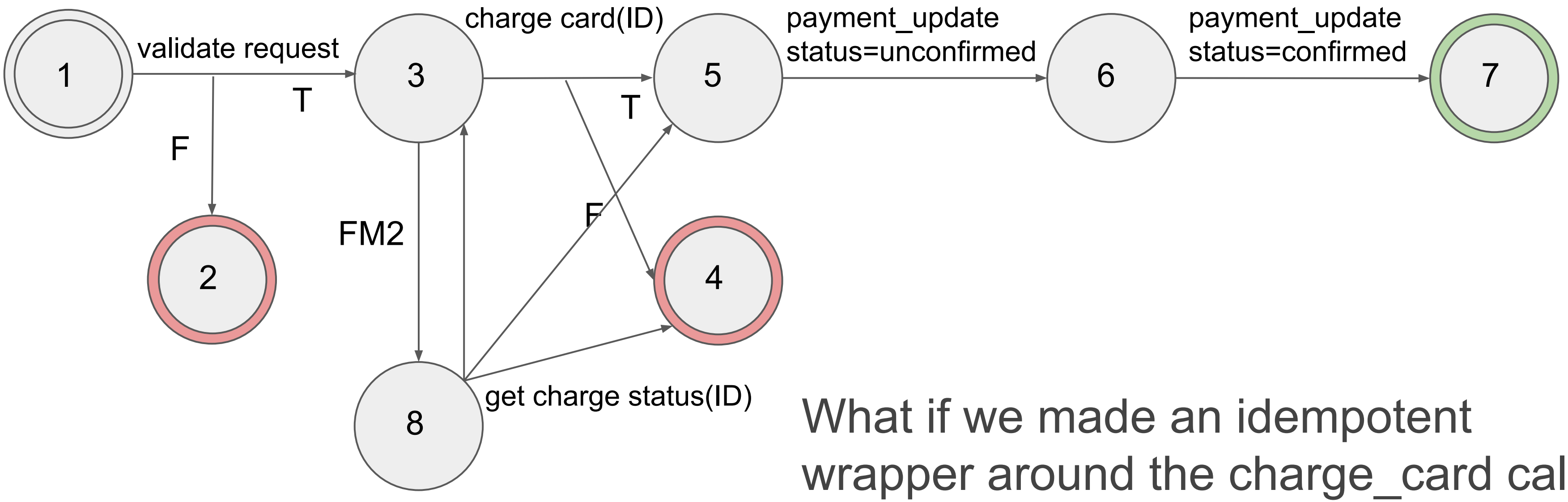
Purchase state machine



In state 5, purchase processor sends a `send_transaction` message, even if already sent before

FM#2: Request sent, response not received

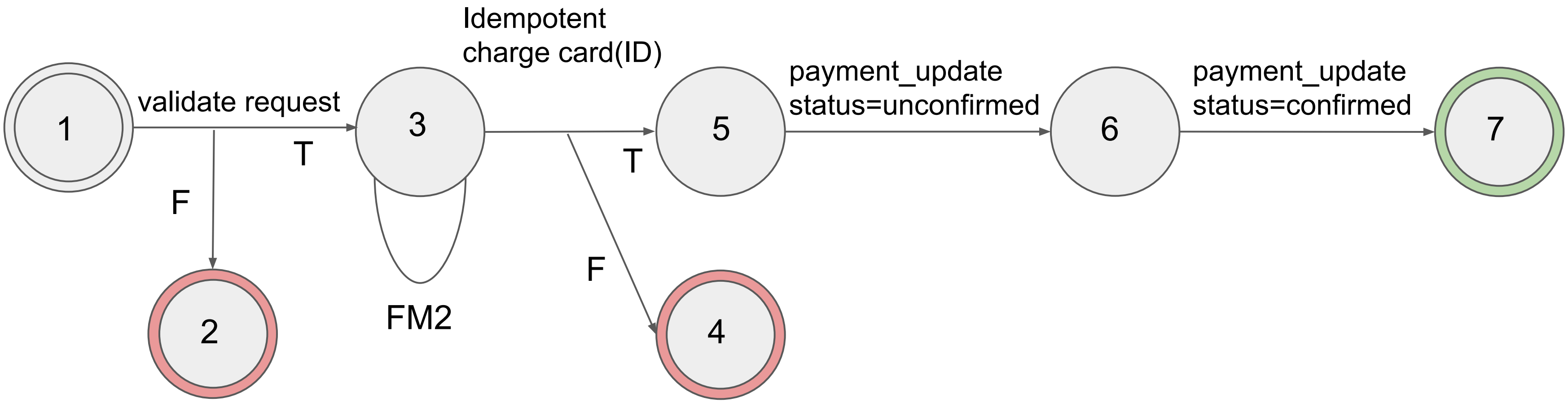
Remember how we handled this? Can we do better?



What if we made an idempotent wrapper around the charge_card call?

FM#2: Request sent, response not received

The logic is the same, but the state machine is simpler



How does the web client handle async?

- HTTP 202 status: The request has been accepted for processing, but the processing has not been completed.
- Client needs to be notified of state transitions
- Websockets can be used to push notifications to the client
- Client polling can be used if websockets are not an option

```
@app.route('/purchases', methods=['POST'])
def buy_bitcoin(params):
  user           = User.find(params['user_id'])
  bitcoin_amount = params['amount']
  usd_amount     = params['total']
  destination    = params['address']
  credit_card    = user.primary_credit_card

  valid = purchase_policy.validate_request(
    bitcoin_amount, usd_amount, credit_card
  )

  if not valid:
    return "Invalid request", 422

  success = credit_card_service.charge(
    credit_card, usd_amount
  )

  if not success:
    return "Credit card charge failed", 422

  transaction_id = bitcoin_service.generate_transaction(
    bitcoin_amount, destination
  )

  while !bitcoin_service.transaction_confirmed(transaction_id):
    time.sleep(1)

  return "Purchase complete!"
```

Even more failures!

What happens if execution spontaneously halts?

Interruptions

- Any part of your code may be interrupted at any time
  - Your process may receive a termination signal
  - Your hard drive may fail
  - Unexpected runtime exception
  - Anything is possible in the cloud
  - Your database could crap out during high load
- You must be able to handle these interruptions
  - State machines make it easy to recover from interruptions
- Can be worthwhile to simulate failures (Neflix’s Chaos Monkey)

Object models

Purchase Processor
- perform_next_action()
- process_message(message)

Purchase Data
- state (Enum)
- card_data
- charge_id
- bitcoin_payment_id
- transaction_id

- persist()

Credit Card Service
- charge_card(ID, card_data)

- private create_charge(ID)
- private get_charge(ID)

Bitcoin Service
- send_transaction
- process_payment_update (message)

Recap: the beauty of state machines

- State machines to orchestrate asynchronous interactions
- Safely retry state transition operations with idempotence
- Loose coupling between state transition operations helps to isolate failures between modules and facilitates testing
- They are easily extensible
  - What if we want to issue a credit card refund if the Bitcoin transaction times out?

Thanks for listening!

If you have any questions you can reach me at jimpo@coinbase.com