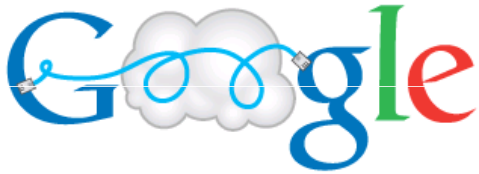
Hooking Stuff Together – Programming the Cloud

Gregor Hohpe

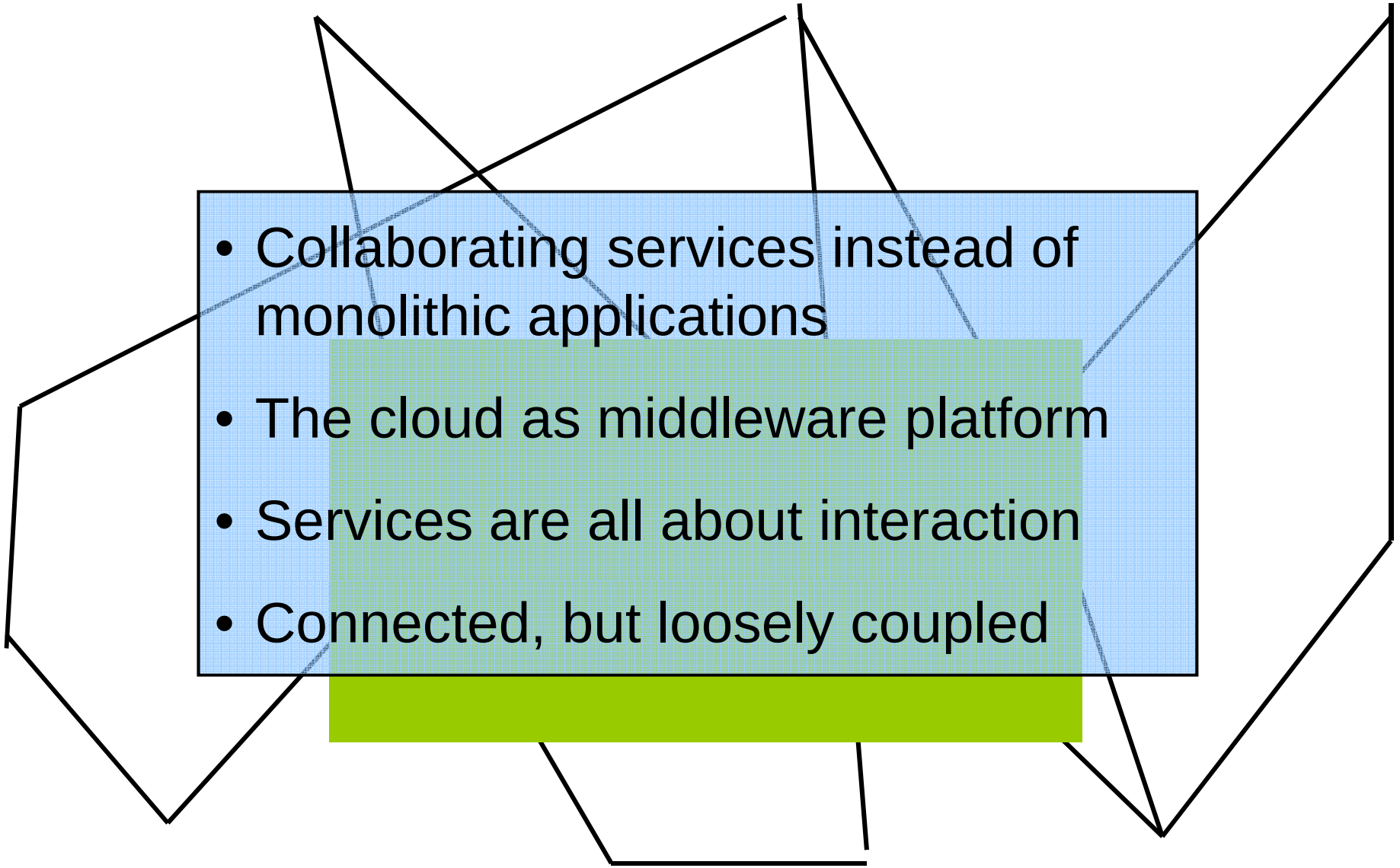


www.eaipatterns.com

www.conversationpatterns.com

~~Yesterday's~~ Software Environment

Today's



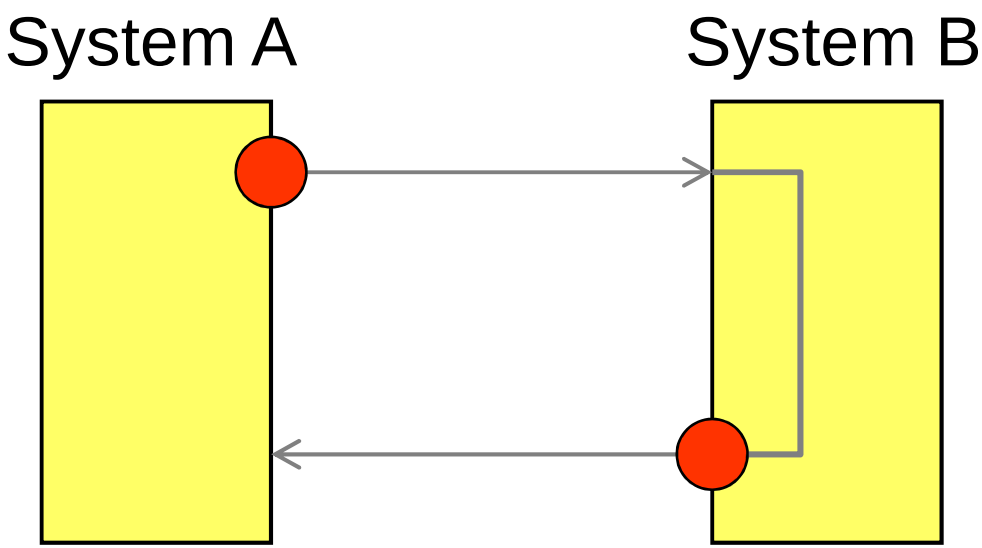
- Collaborating services instead of monolithic applications
- The cloud as middleware platform
- Services are all about interaction
- Connected, but loosely coupled

Less is More?

- NO Call Stack
- NO Transactions
- NO Promises
- NO Certainty
- NO Ordering Constraints
- NO Assumptions

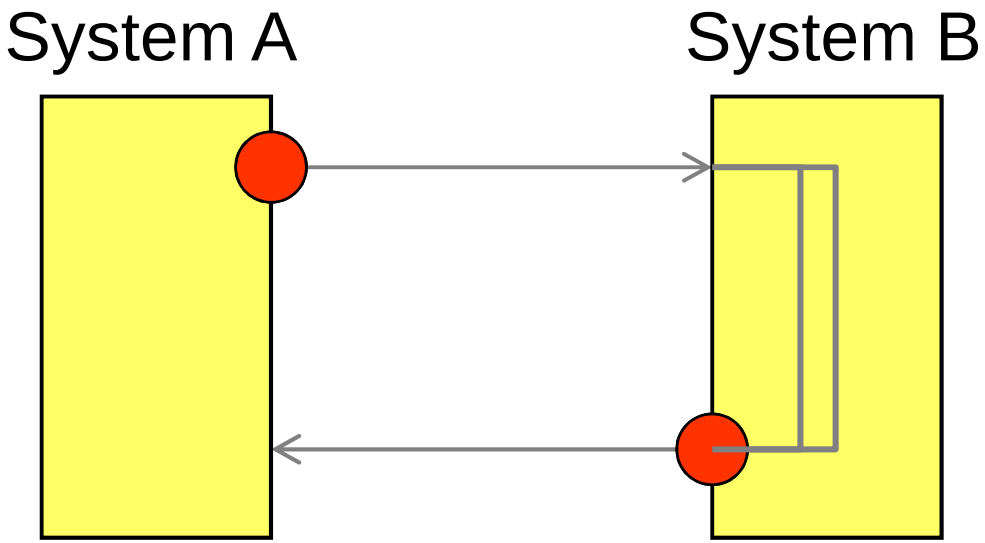
Scary?	Yes!
Cool?	Yes!
Way to go?	Yes!

A Simple Interaction



What if the response does not come?

Communication Problems



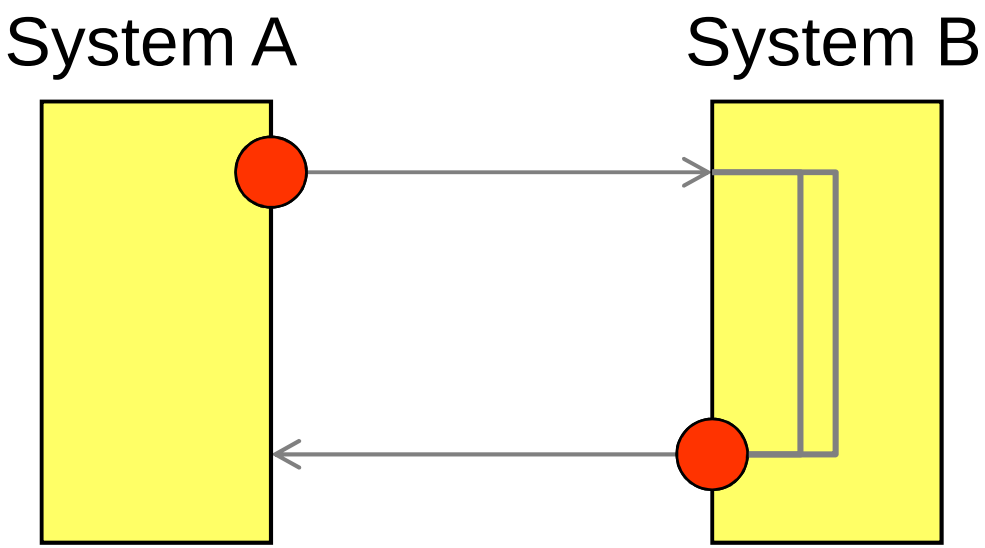
Lost Request?

Lost Response?

System B Crashed?

Retry?

Delayed Response

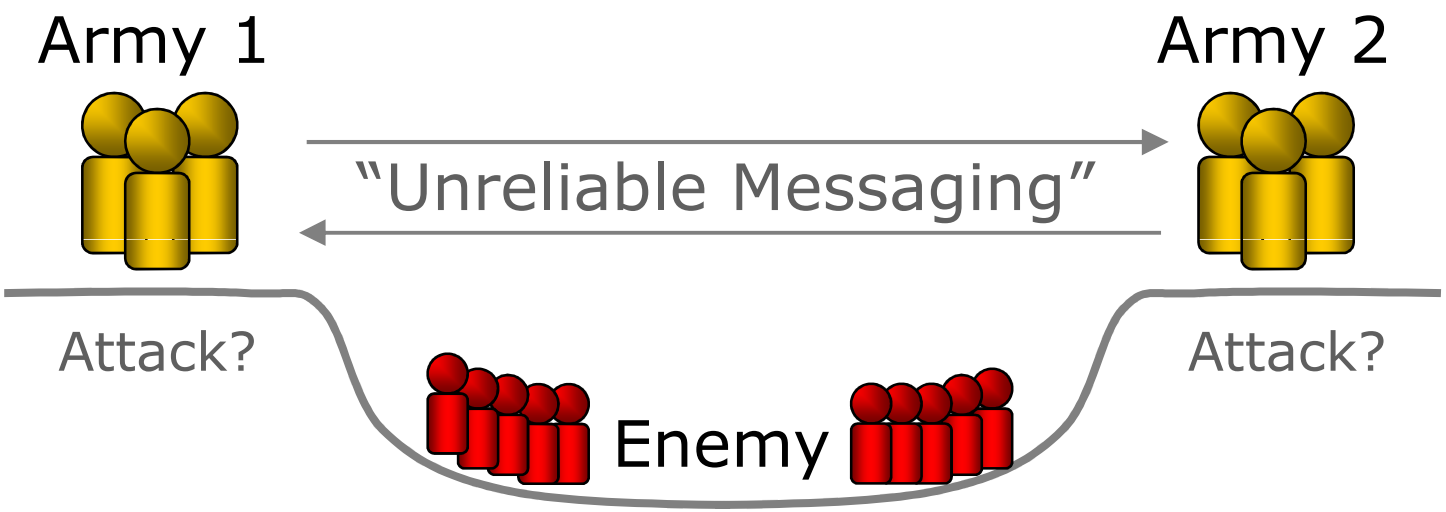


Executed Once?

Executed Twice?

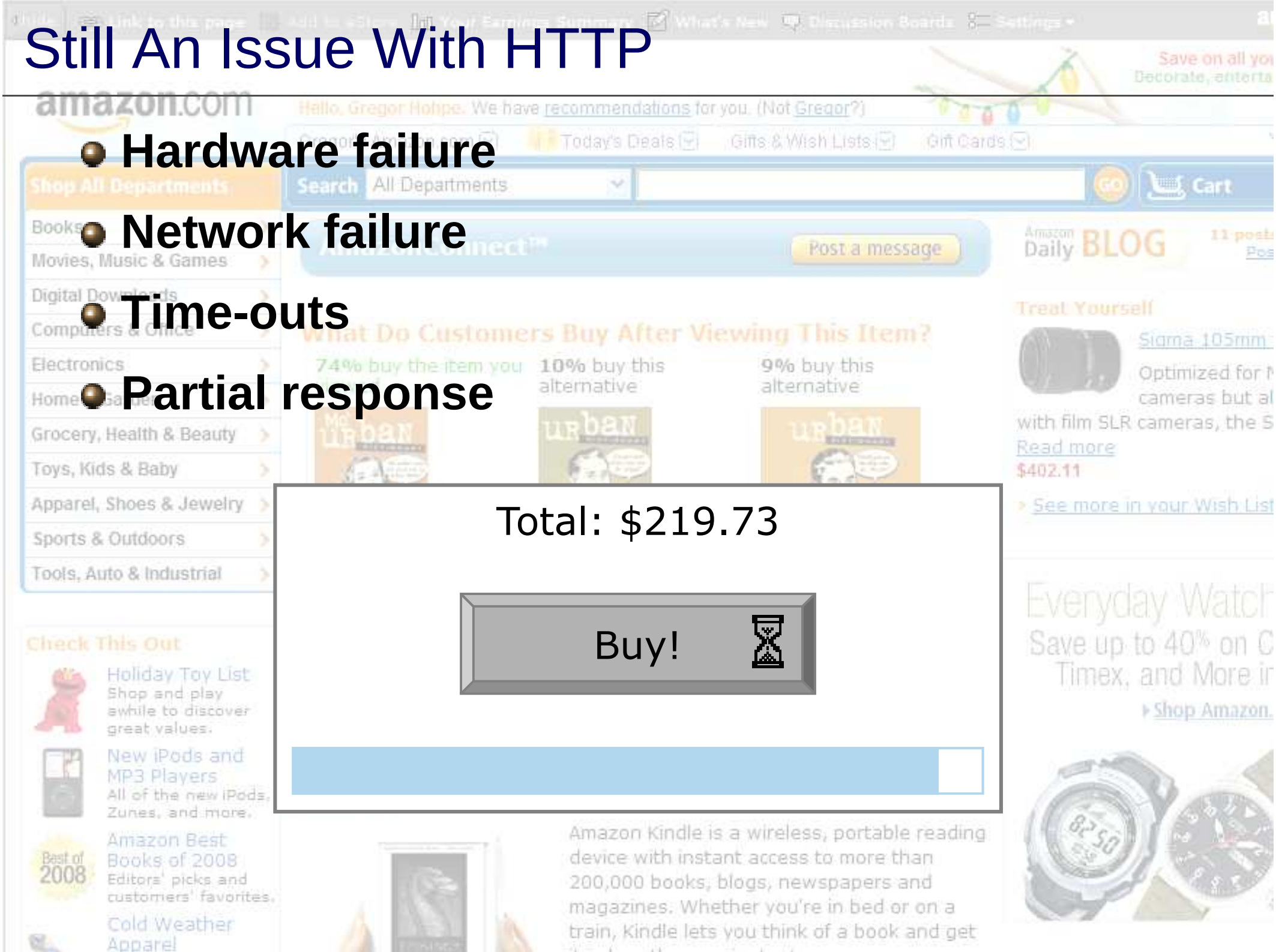
Inherent State Uncertainty

- System A is never 100% sure what state System B is in
- This problem does not occur in a monolithic system
- Compare Byzantine General's Problem



Still An Issue With HTTP

- **Hardware failure**
- **Network failure**
- **Time-outs**
- **Partial response**



What About Distributed Transactions?

- **Require coordinator**
- **Even 2 Phase Commit has windows of uncertainty**
- **Not practical for long running interactions**
  - **Locks not practical / economical**
  - **Isolation not possible / practical**
- **Usually not supported**
- **Don’t scale**

“Life Beyond Distributed Translations – an Apostate’s Opinion”
--Pat Helland

Now What?

- Live with uncertainty
- Simplicity is King
- Interaction
- Asynchrony
- New programming models
- Behold the Run-time
- Patterns Renaissance

Living With Uncertainty

ACID (before)

- **Atomic**
- **Consistent**
- **Isolated**
- **Durable**

Predictive

Accurate

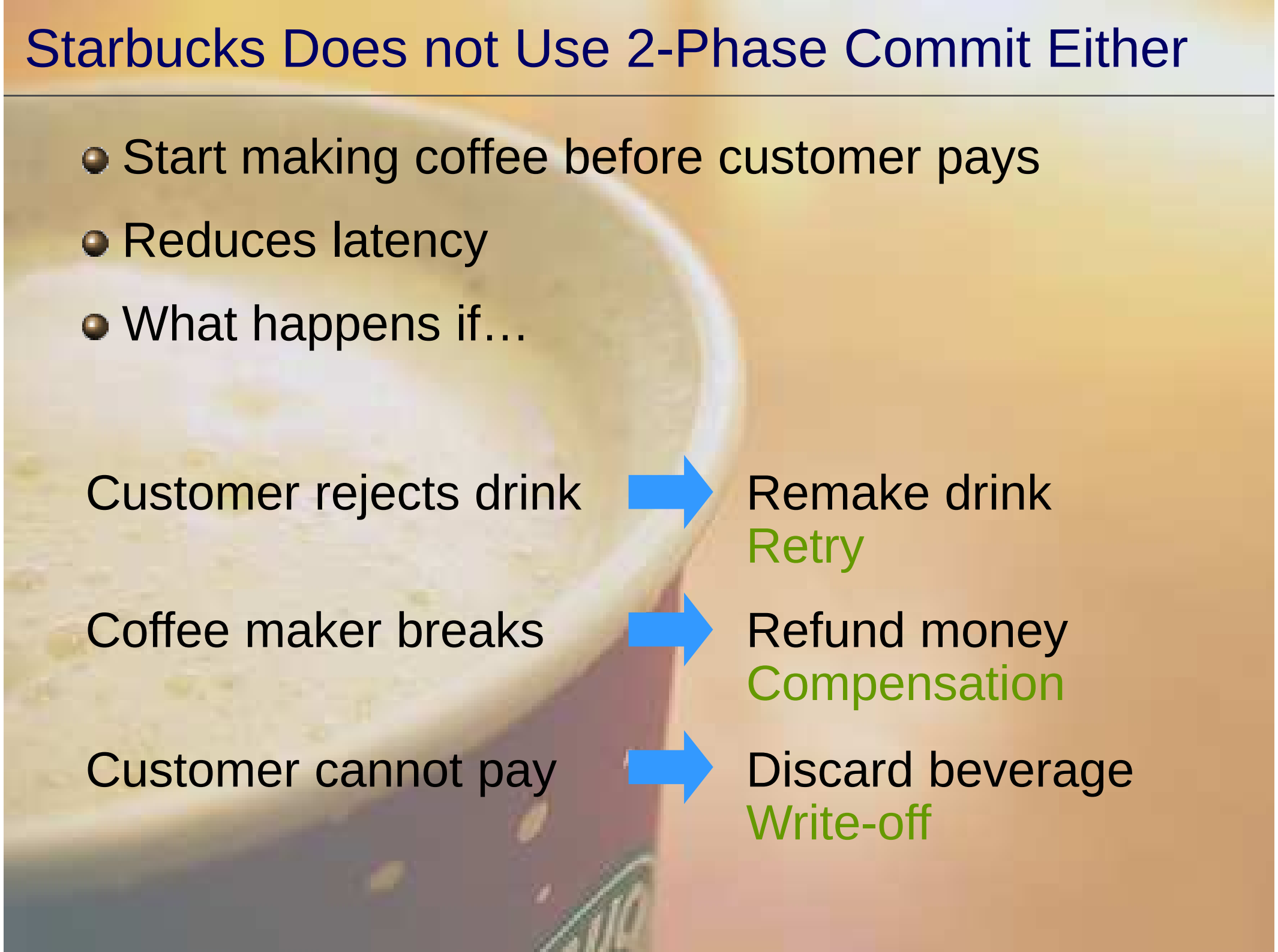
ACID (today)

- **Associative**
- **Commutative**
- **Idempotent**
- **Distributed**

Flexible

Redundant

Starbucks Does not Use 2-Phase Commit Either



- Start making coffee before customer pays
- Reduces latency
- What happens if…

Customer rejects drink → Remake drink
Retry

Coffee maker breaks → Refund money
Compensation

Customer cannot pay → Discard beverage
Write-off

Simplicity is King

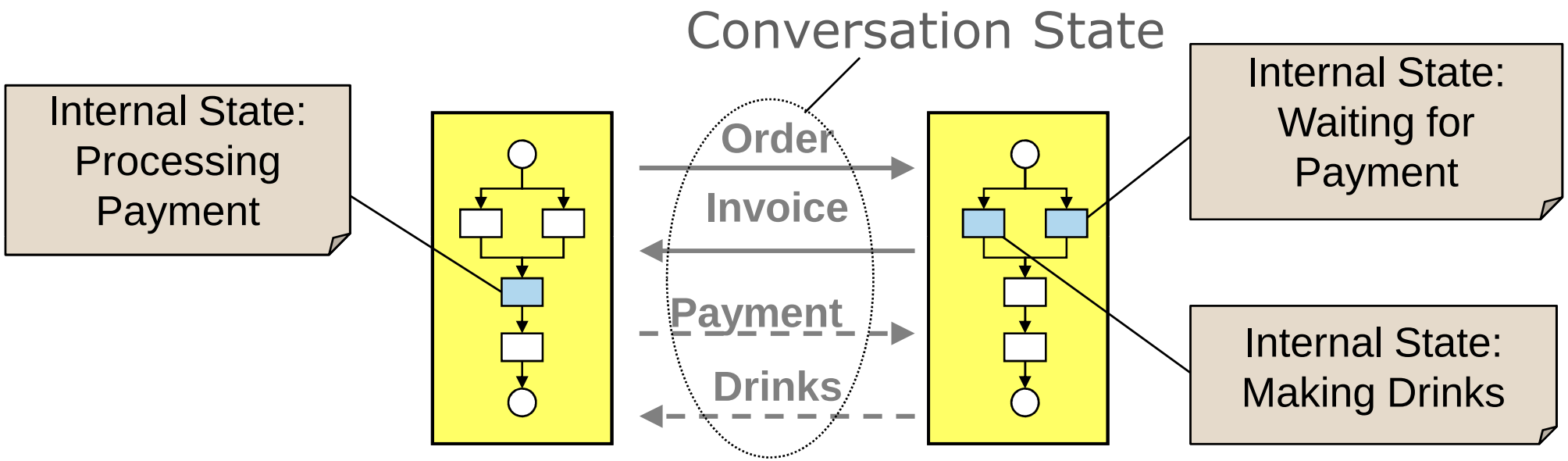
- Even simple things become complicated in a distributed environment
- If it looks complicated on paper it’s likely to be impossible in practice
- If you can’t understand it, other developers likely won’t either
- A well understood failure scenario can be better than an incomprehensible and unproven “failsafe” system

Focus on Interaction

- In the OO world interaction is essentially free
- Powerful structural mechanisms: inheritance, composition, aggregation
- In the cloud, more focus shifts to interaction. Structural composition mechanisms are limited.

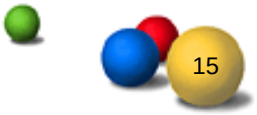
Conversations

- Series of related messages between parties
- Not handled at lower layer
- Endpoints keep some conversation state
- Protocol design



Asynchrony

- Exchange through messages, not RPC
- Waiting for the results of an HTTP request is not a smart use of a 3 GHz processor
- Request and response message typically handled by different parts of your program, even if the same TCP connection
- Reduced assumptions about timing and state



Programming Abstraction: MapReduce

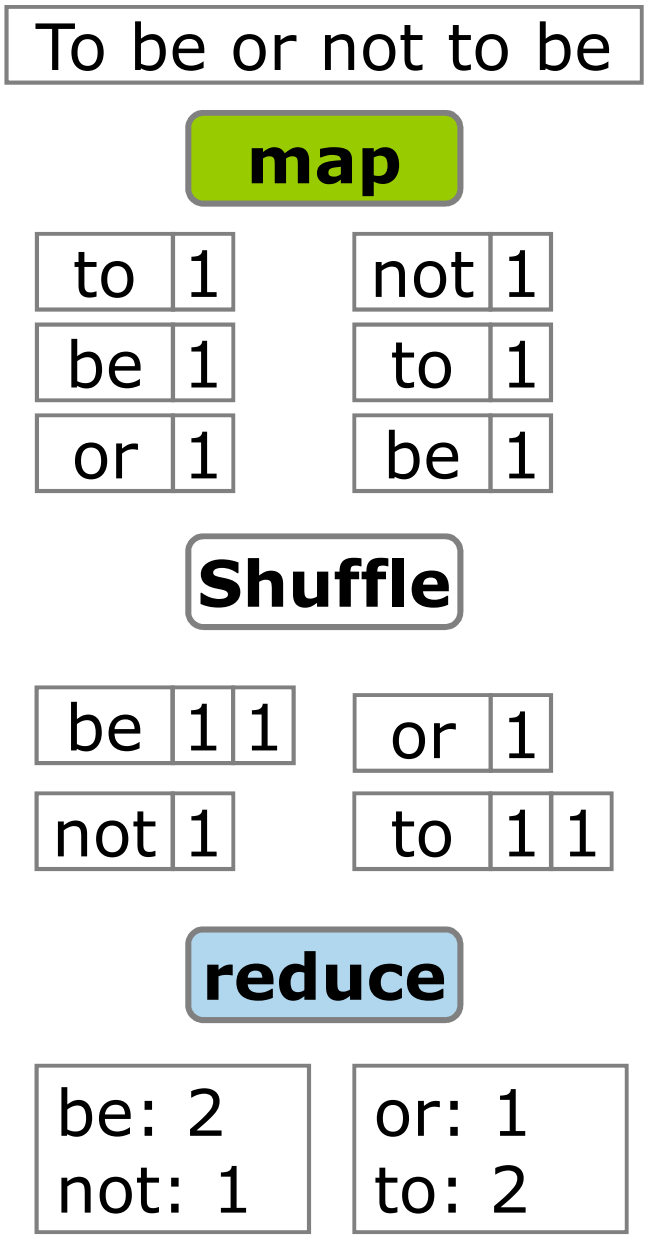
- Represent computing problems as Map and Reduce step
- Inspired by functional programming
- “Embarrassingly parallel problems”
- True framework: don’t call us, we’ll call you

```
map(in_key, data)
 → list(key, value)


reduce(key, list(values))
→ list(out_data)
```

http://research.google.com/archive/mapreduce.html

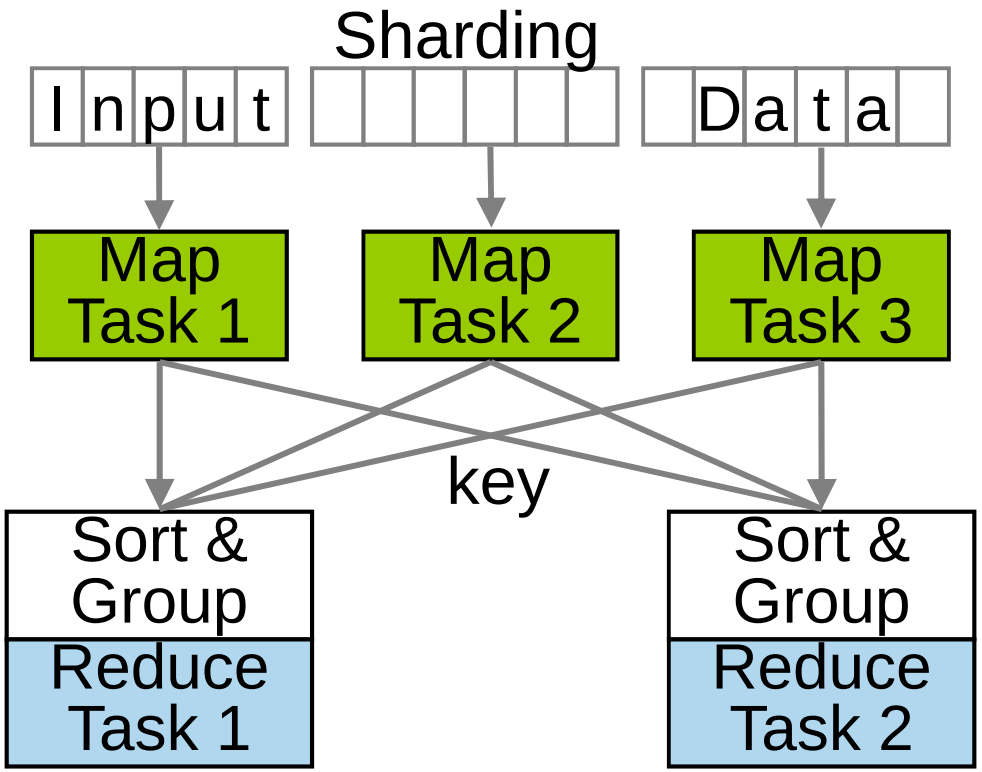
MapReduce: Word Frequency

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(key + ": " + result);
```



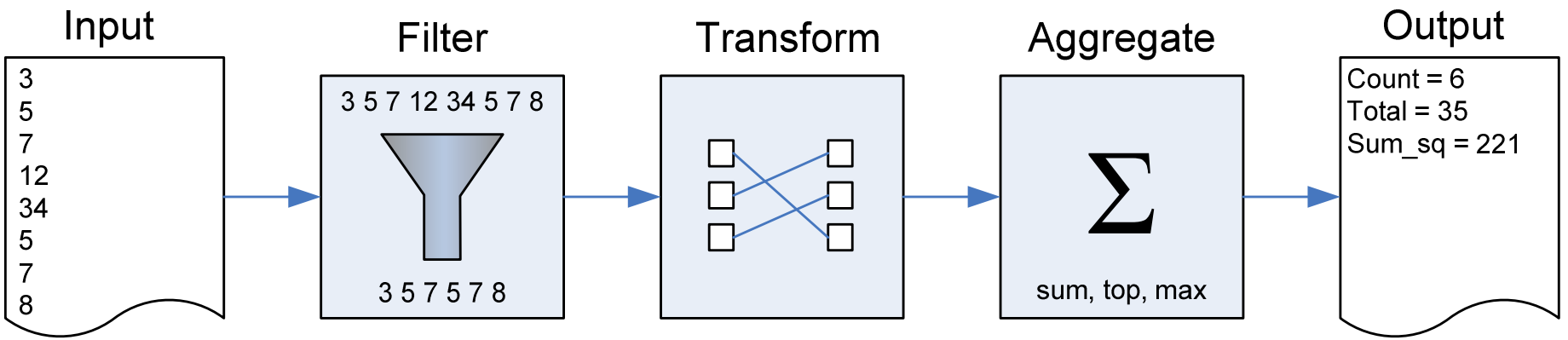
MapReduce Run-time

- Distribute data among many machines, execute same computation at each machine on its dataset
- Open source implementation: Hadoop



Domain Specific Language: Sawzall

- Commutative and associative operations allow parallel execution and aggregation




```
count: table sum of int;
total: table sum of float;

x: float = input;

emit count <- 1;
emit total <- x;
```

http://labs.google.com/papers/sawzall.html

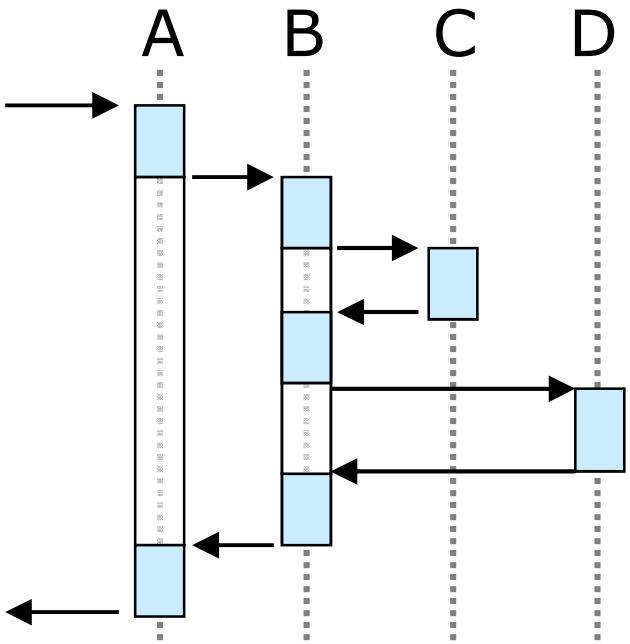
Behold the Run-time

- Some programming abstractions are great, e.g. MapReduce
- In a single-threaded call-stack machine, programming model and execution model match fairly closely
- In a highly distributed dynamic system, they are very different!
- Monitoring, run-time analysis, and visualization critically important

Behold the Run-time

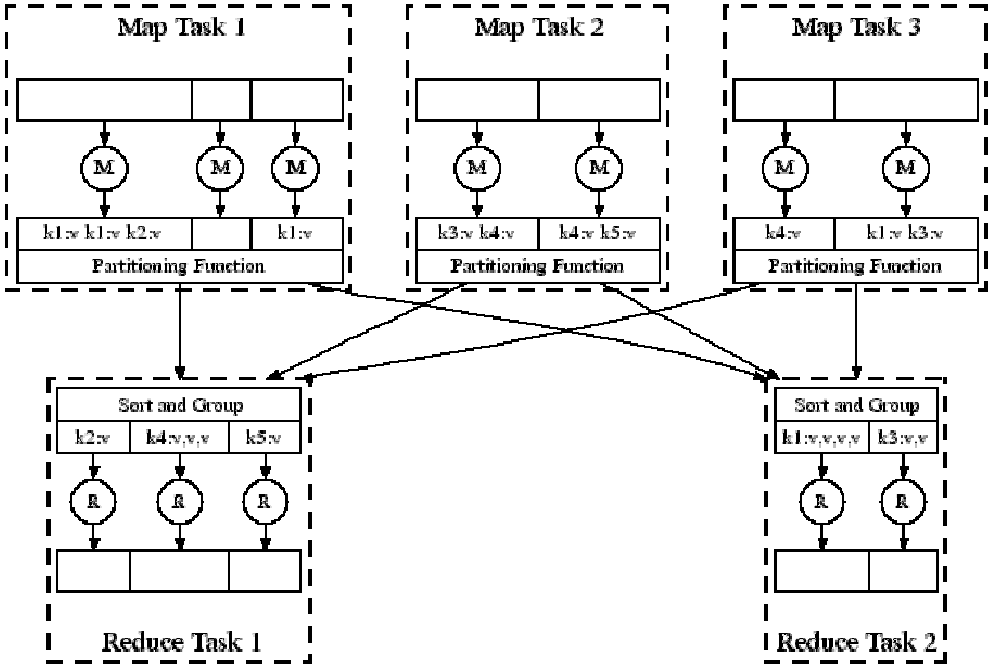
- Call Stack

```
void a() {
  b();
}

Void b() {
  c();
  d();
}
```



- MapReduce

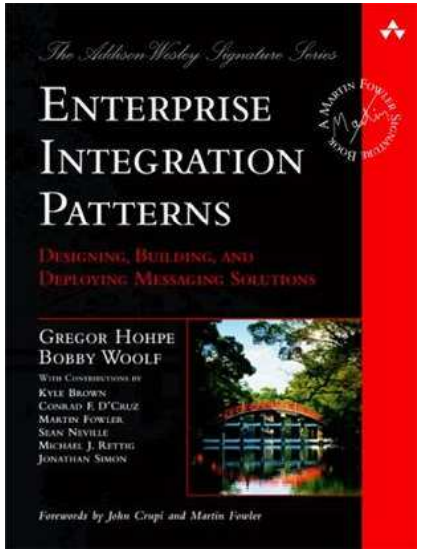
```
map(in_key, data)
 → list(key, value)


reduce(key, list(values))
→ list(out_data)
```



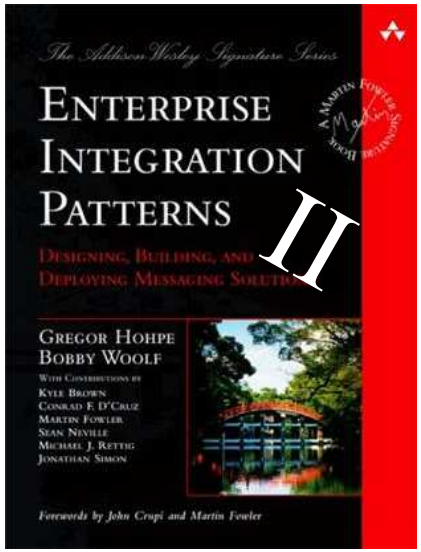
My Work

- Messaging Patterns (65)
  - Messaging Systems
  - Messaging Channels
  - Message Construction
  - Message Routing
  - Message Transformation
  - Messaging Endpoints
  - System Management
- Conversation Patterns
  - Discovery
  - Establishing a Conversation
  - Multi-party Conversations
  - Reaching agreement
  - Resource Management
  - Error Handling

www.eaipatterns.com



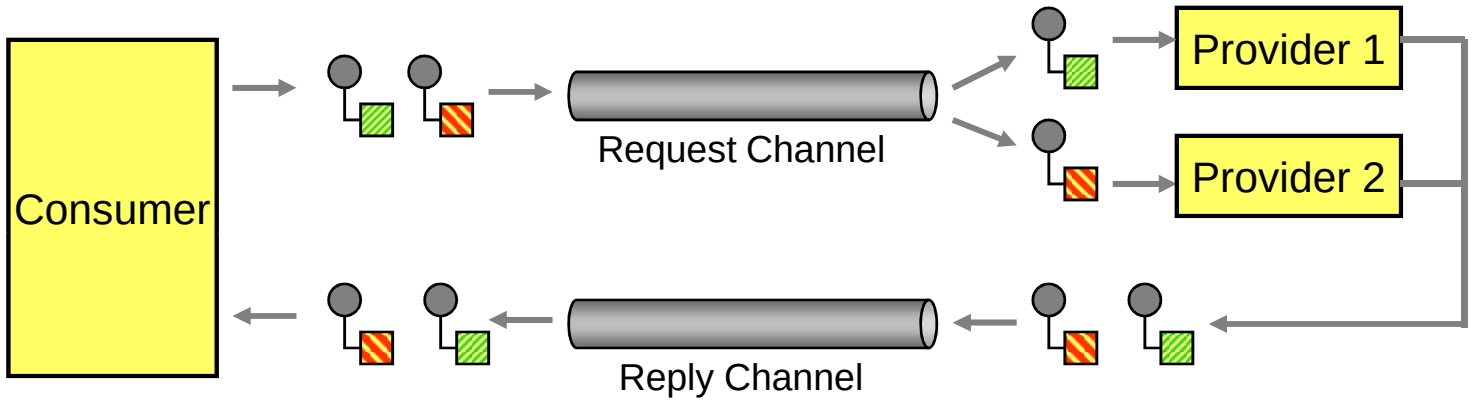
www.conversationpatterns.com



Patterns – 10 Years After GoF

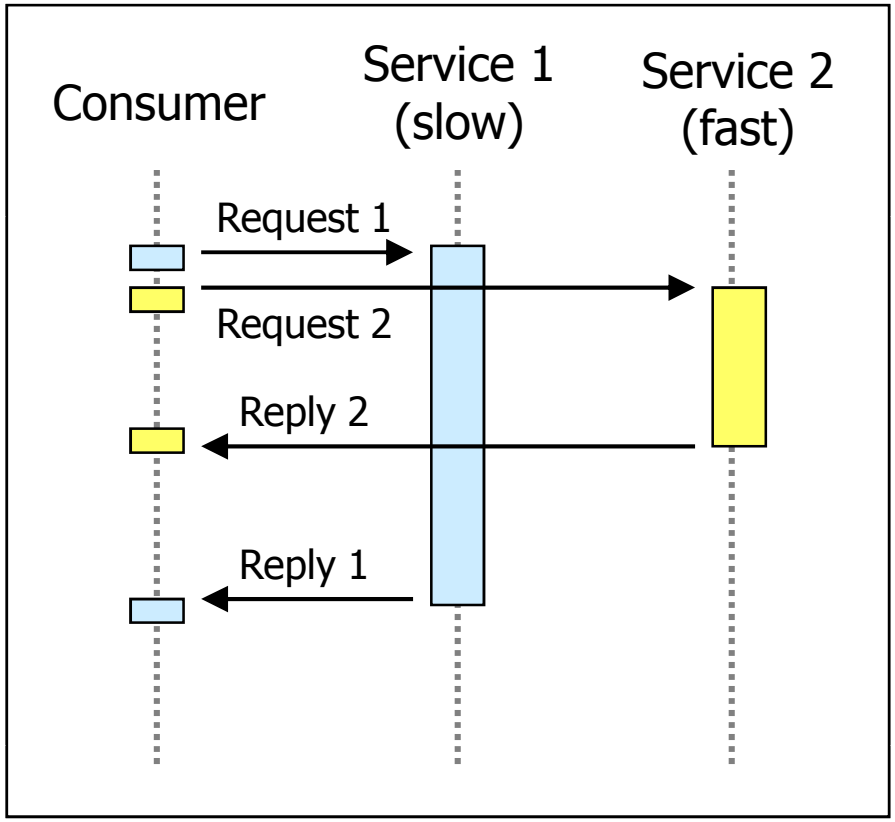
- New programming models bring new patterns.
- “Mind sized” chunks of information
  (Ward Cunningham)
- Human-to-human communication
- Expresses intent (the “why” vs. the “how”)
- Makes assumptions explicit
- Observed from actual experience

Multiple Service Providers



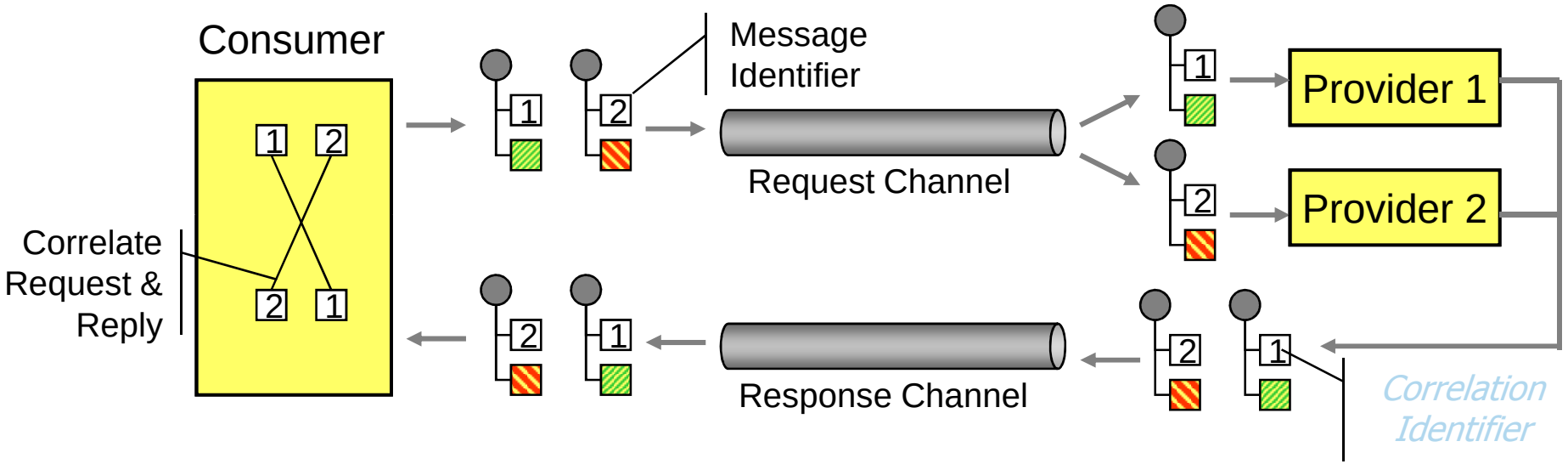
- Request message can be consumed by more than one service provider
- *Point-to-Point Channel* supports *Competing Consumers,* only one service receives each request message
- Channel queues up pending requests

Multiple Service Providers



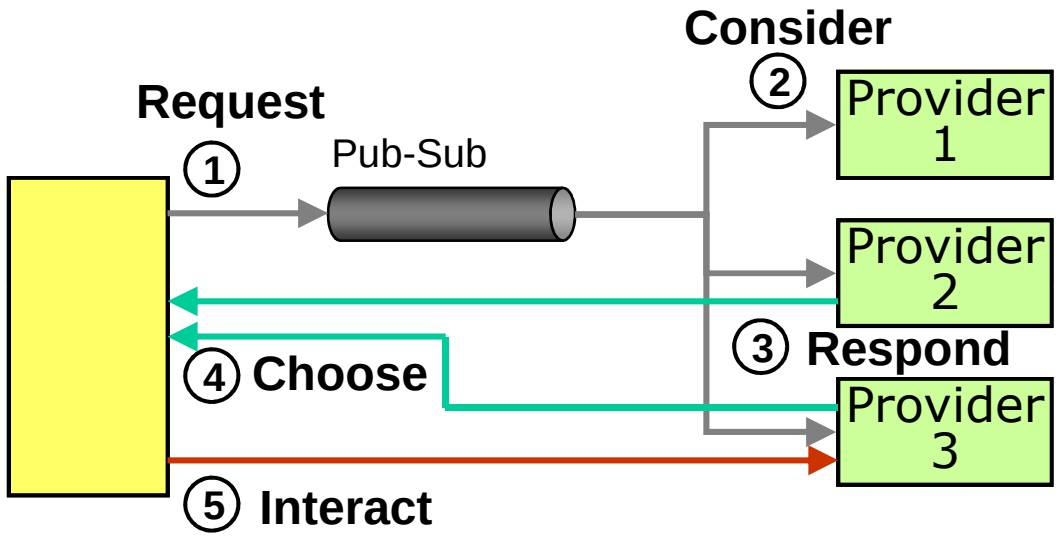
- Reply messages get out of sequence
- How to match request and reply messages?
  - Only send one request at a time
    → very inefficient
  - Rely on natural order
    → bad assumption

Pattern: *Correlation Identifier*



- Equip each message with a unique identifier
  - Message ID (simple, but has limitations)
  - GUID (Globally Unique ID)
  - Business key (e.g. Order ID)
- Provider copies the ID to the reply message
- Consumer can match request and response

Conversation Pattern: *Dynamic Discovery*

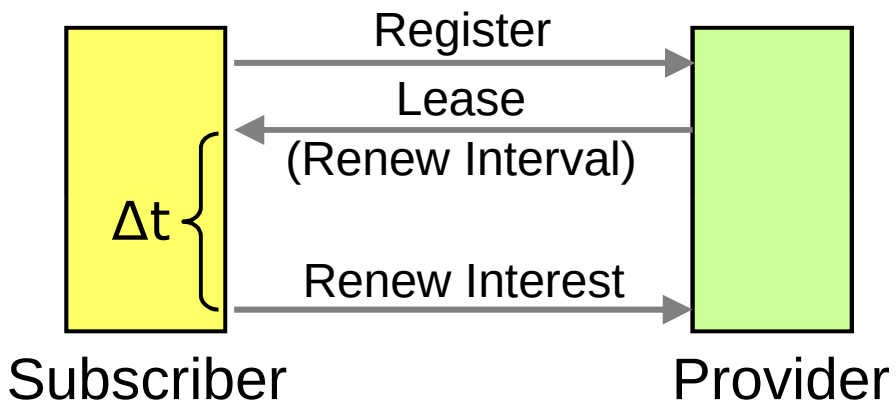


1. Broadcast request
2. Provider(s) consider whether to respond (load, suitability)
3. Interested providers send responses
4. Requestor chooses “best” provider from responses
5. Requestor initiates interaction with chosen provider

- Examples: DHCP, TIBCO Repository discovery

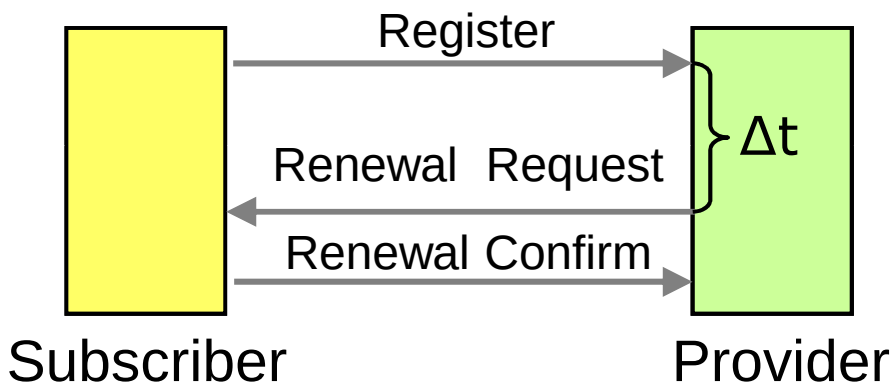
Conversation Pattern: *Renewing Interest*

Automatic Expiration



- “Lease” model
- Heartbeat / keep-alive
- Subscriber has to renew actively
- Example: Jini

Renewal Request



- “Magazine Model”
- Subscriber can be simple
- Provider has to manage state for each subscriber

Keep These in Mind

- Live with uncertainty
- Simplicity is King
- Interaction
- Asynchrony
- New programming models
- Behold the Run-time
- Patterns Renaissance

Fin