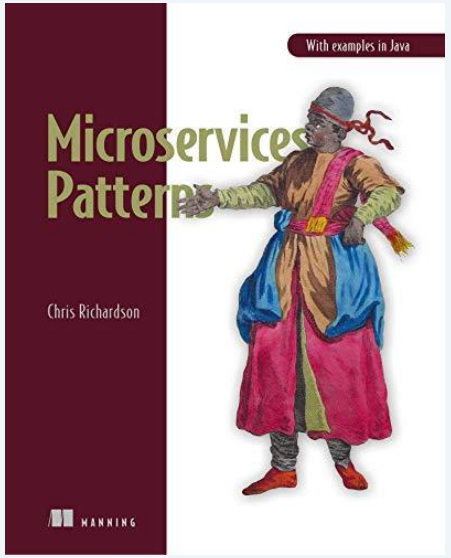
Interprocess Communication Patterns I

Synchronous Communication

Objectives

- Communication between services
- Remote Procedure Invocation Pattern:
  - Context & Problem
  - Forces
  - Solutions
  - Resulting Context
  - Issues
- The Circuit Breaker Pattern
- Netflix Fault Tolerance



Communication between services

- Before we introduced microservices communication was easy!
  - All modules in one codebase
  - All modules accessing one database
  - Method/function calls, global variables, event frameworks
- Now we have a problem that didn't exist before
  - How can service A and service B talk to each other?

What comms are needed?

	One –to-one	One-to-many
Synchronous	Request/Response • Response in timely fashion (might event block) • Tight coupling One-way notifications	-
Asynchronous	Async Request/Response • Client doesn't block • Loose coupling One-way notifications	Publish/subscribe • Client publishes a notification message, one or more servers consume Publish/async response • Client publishes request, waits a certain time for responses

Choosing the right pattern

- This will depend on your particular application needs.
- Synchronous communication is **much** simpler to implement
- Asynchronous communication is more likely to handle large scale/high traffic applications
  - Also introduces lots of other issues.
- The pattern language helps us to decide what is right

Pattern: Remote Procedure Invocation (RPI)

Synchronous interprocess communication

Context

- The Microservices Architecture Pattern has been applied
- The services must handle requests from external clients and services
  - This requires service collaboration which mean inter-process communication

Forces

- Services often need to collaborate
- Synchronous communication means tight runtime coupling
  - Both client and server must be available for the duration of the request

API-first Design

- Microservices are all about communicating through APIs
- It is very important to define a services APIs precisely
  - Perhaps using protobufs, SwaggerHub or an IDL
- No matter how good your microservices are, without properly defined APIs they won't work!
- Ensure you version your services correctly
  - MAJOR.MINOR.PATCH
  - MAJOR - Incompatible changes to the API
  - MINOR- Backward-compatible enhancements to the API
  - PATCH - Backward-compatible bug fix

Solution

- Use Remote Process Invocation for inter-service communication
- The client uses a request/reply based protocol to make requests
- Many examples
  - REST
  - gRPC
  - Apache Thrift

We will look at these as examples

Resulting Context / Consequences

Benefits

- Simple and familiar
- Request/reply is easy
- System simplicity – no intermediate broker

Drawbacks

- Usually only supports request/reply
- Can lead to tight coupling

Issues

- Reduced availability since client and service must be available for the duration
- Client needs to discover service instances

Related Patterns

- Messaging (asynchronous communication) is an alternative
- The reduced availability issue can be addressed by the circuit breaker pattern
- The service discovery issue must be addressed by either
  - Client side service discovery, or
  - Server-side discovery

The Circuit Breaker Pattern

- This pattern addresses the reliability /availability issue resulting from using the RPI pattern.

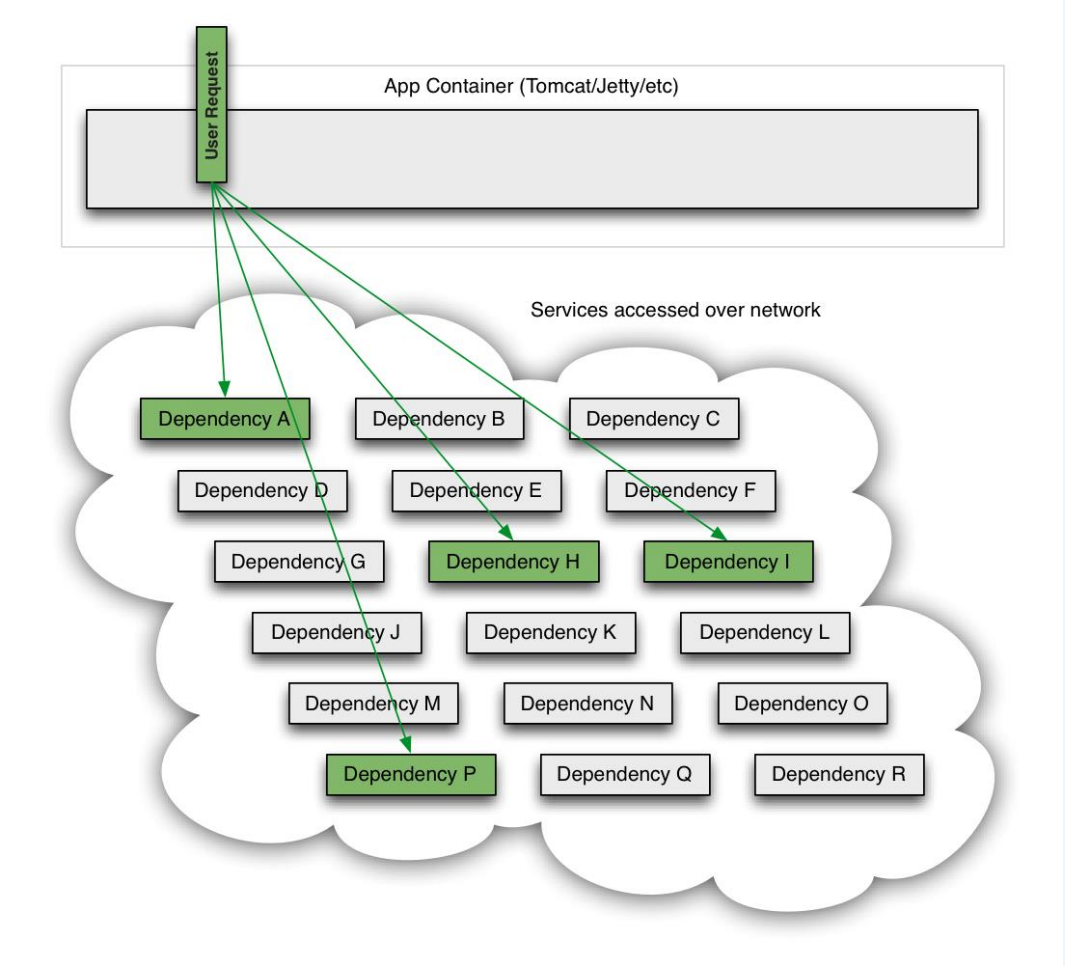
- The **context** here
  - Synchronous invocation of a service can fail
    - Because the service is unavailable
    - Because the service is exhibiting unacceptable latency

- The failure of one service can lead to cascading failure of others

- The **forces** are simple
  - Cascading failures are unacceptable

Netflix Fault Tolerance Approach

- Without taking steps to ensure fault tolerance 30 dependencies with 99.99% uptime each would result in 2+ hours downtime/month
  - $99.99^{30} = 99.7$
  - 24*30 * 0.03% = 2.16 hours
- When a single API dependency fails at 50+ requests/sec, all request threads can block in seconds.



https://netflixtechblog.com/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a

Netflix Fault Tolerant Approach

- Network Timeouts and Retries
  - Never block indefinitely – using timeouts ensures resources are never tied up indefinitely.
- Limiting the number of outstanding requests from a client to a server
  - Impose an upper bound on the outstanding requests a client can make
  - If it's over this, fail automatically – probably pointless to make additional requests
- Use the Circuit Breaker Pattern

Aside: Netflix Chaos Simians (there's loads more)

- Chaos Monkey

We created Chaos Monkey to randomly choose servers in our production environment and turn them off during business hours.

- Chaos Kong

Building on the success of Chaos Monkey, we looked at an extreme case of infrastructure failure. We built Chaos Kong, which doesn't just kill a server. It kills an entire AWS Region.

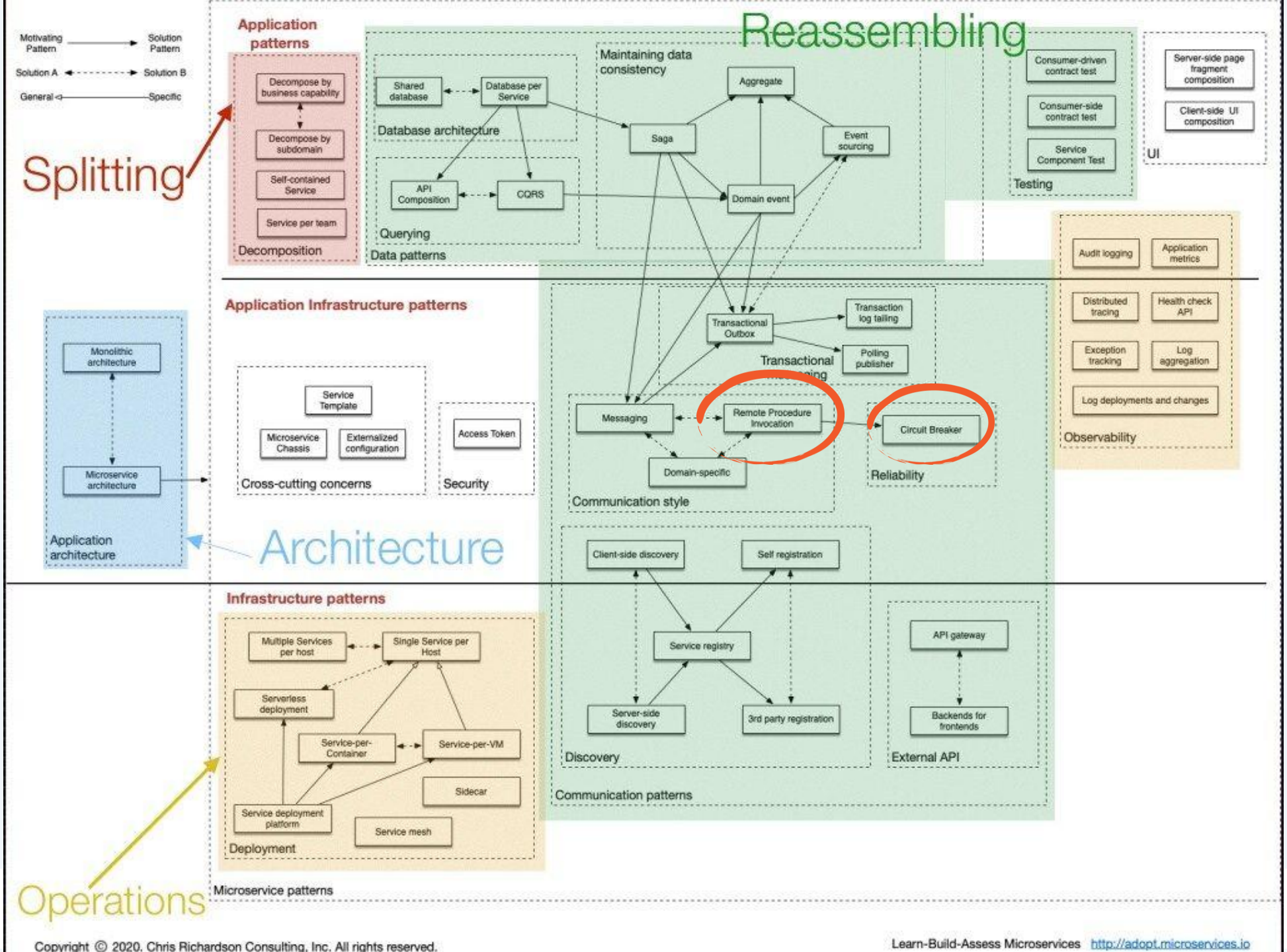
https://netflixtechblog.com/tagged/chaos-engineering
https://www.gremlin.com/chaos-monkey/the-simian-army/

Circuit Breaker Pattern: Solution

- When the number of consecutive failures to a service crosses a threshold, the circuit breaker trips.
- For the duration of a timeout period all attempts to invoke the service fail immediately
- After the timeout expires a limited number of test requests are passed
  - If they succeed, normal operation resumes
  - If there is a failure, a new timeout period begins

Resulting Context

- Benefits
  - Services handle the failure of the services they invoke
- Issues
  - Choosing timeout values is a challenge – false positives or excessive latency.



Learn-Build-Assess Microservices http://adopt.microservices.io

Summary

- Communication between services
- Remote Procedure Invocation Pattern:
  - Context & Problem
  - Forces
  - Solutions
  - Resulting Context
  - Issues
- The Circuit Breaker Pattern
- Netflix Fault Tolerance

Questions or Comments?