Distributed Systems
Andrew Tanenbaum + Maarten Steen

Building Microservices Ch4

Communication styles, pros/cons

① synchronous blocking ② asynchronous nonblocking
③ request-response collab ④ event-driven collab

In-process, calls w/in a single process
inter-process, calls b/w different processes across a network

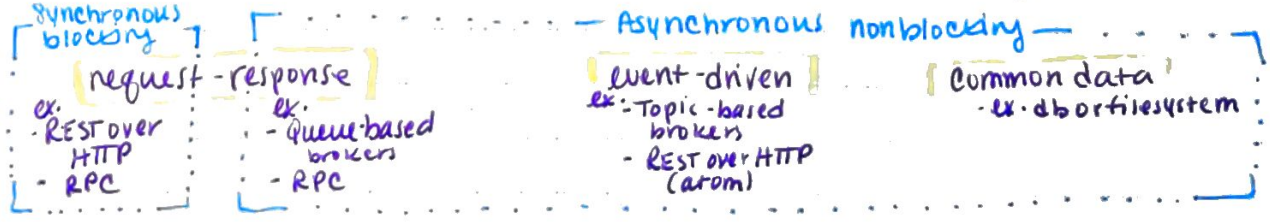
a) Performance
- in-process > inter-process, the compiler can make optimizations may cause you to rethink your API if you're comparing in vs. inter-
- need to be aware of the size of your params b/c they will be passed over a network b/w microservices + serialized/deserialized
- need to be aware of when a call is happening over a network

b) Changing Interfaces
- the service exposing the interface + the service using the interface are deployed separately so breaking rollouts to the interface need to either.

c) Error Handling
- in-process the errors are likely deterministic + relatively straightforward
- inter-process you can have a host of downstream or network issues
  * crash failure (server crash)
  * omission failure (sent something, no response or events stop coming)
  * timing failure (too late or too early)
  * response failure (response received, but incomplete or wrong info)
  * arbitrary failure (aka Byzantine failure, something has gone wrong but no one can agree on if/why the failure occurred

  → may be transient errors
  → need to accurately describe errors, whether with HTTP codes or otherwise so that clients can respond correctly

inter-service comms styles



inter-μservice comms cont'd.

- Synchronous blocking, make call + wait
- Async & nonblocking, send call + don't wait
  - Request-response, make call, wait, expect response
  - Event-driven, send event + don't care who listens
  - Common data, share a data source

understand needs around

- reliable comms - acceptable latency - vol. of comms. - security needs

@First choose b/w request-response + event driven + you can mix + match communication styles

Sync blocking + easily understood
- temporal coupling b/w instances of μservices
- potential cascading issues b/c of outages
- resource contention in chained blocking calls

async nonblocking

↳ request-response: make request then any instance can receive response
↳ Common data: upstream μservice changes common data to be used by ↓stream μservice
↳ event-driven: upstream broadcasts event, downstream can listen

+ temporal decoupling
+ handy if triggered work is slow
- ↑ complexity
- ↑ choices

*async/await functions use a sync blocking call

impl · common data

data ~~dlkr~~ lake, dump info in
data warehouse, μservice would need to know structure of warehouse
↳ for both data is assumed to run in one direction upstream to downstream
files, w/ polling to check for new files
db

+ simple
+ interoperability (old & new systems, etc)
+ good if you're sending a lot of data at once
- ↑ latency
- potential coupling
- potially more points of failure (ex. the filesystem itself)

Pattern: request/response

can be blocking or nonblocking (ex. w/ notification responses)

implementation: sync vs. async

sync., open network connection & hold until response is received

async: send request to queue, work is done by consumer who then publishes to another queue that the work is complete. The services both need to know where to read & send responses to. The queue can hold many requests until the consumer is ready for them. Can't rely on talking w/ the same service instance that created the request so relevant state info. would need to be housed elsewhere

* you will need to handle timeouts for sync or async
* can run calls in parallel vs. in sequence to ↓ latency, async/await can help
  → good for when a response is needed before more work can be done
  → " " " you need to know if a call didn't work, can retry

pattern: Event-driven comms

- service emits event about something that happened. Others may or may not listen.
- event: statement about something that has occurred, typically w/in the emitting service
- pushes the responsibility of knowing what to do in the face of an event to the listening services
- event is a thing that happened, a message is a thing we send over an async comms mechanism

"The message is the medium, the event is the payload"

implementation

- Producers use an API to publish an event to the broker (ex. RabbitMQ)
  Broker handles subscriptions & possibly tracing what msgs the Consumer has seen
- Keep your middleware dumb & keep the smarts in the endpoints
- Can also use an HTTP Feed, but no good for ↓ latency (ex. ~~Atom~~ ← Atom
- * be wary of the sunk cost fallacy, try Atom if you need it but switch to a message broker if you need to
- What should be in the event? just an ID, which may result in a barrage of requests for more information
  → or Fully Detailed events, ~~not~~ everything you'd be happy to share via an API
  Can ↓ coupling + act as a historical record. Just watch how large the msg is

Enterprise Integration Patterns by Gregor Hohpe + Bobby Woolf

What should be in the event cont'd

- You could also send two events, one w/ sensitive info + one w/o (ex. PII)
  - ↳ need to manage visibility + fault tolerance (dying before sending 2nd msg)
- once you put data in an event it's a part of your contract. Removing it could break others

When to use it?

↓ coupling ↑ broadcasting ↑ downstream services work out what to do

But ↑ complexity, you can use a mix of styles though.

- a message hospital (or dead letter queue) + max retries may help avoid catastrophic failovers where your workers die one by one trying to handle a bad message that crashes them
- ensure ↑ monitoring + using correlation IDs