Building Software Systems

Lecture 1.4

Architectural Tactics

SAURABH SRIVASTAVA

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT (ISM) DHANBAD

What you (should) know till now?

Solution Architecture involves bifurcating the IT solution from Process elements

The IT Solution may have different types of requirements

We looked into a bit more details as far as the Quality Requirements are concerned

The IT Solution needs to be architected; in particular, it needs a Software Architecture

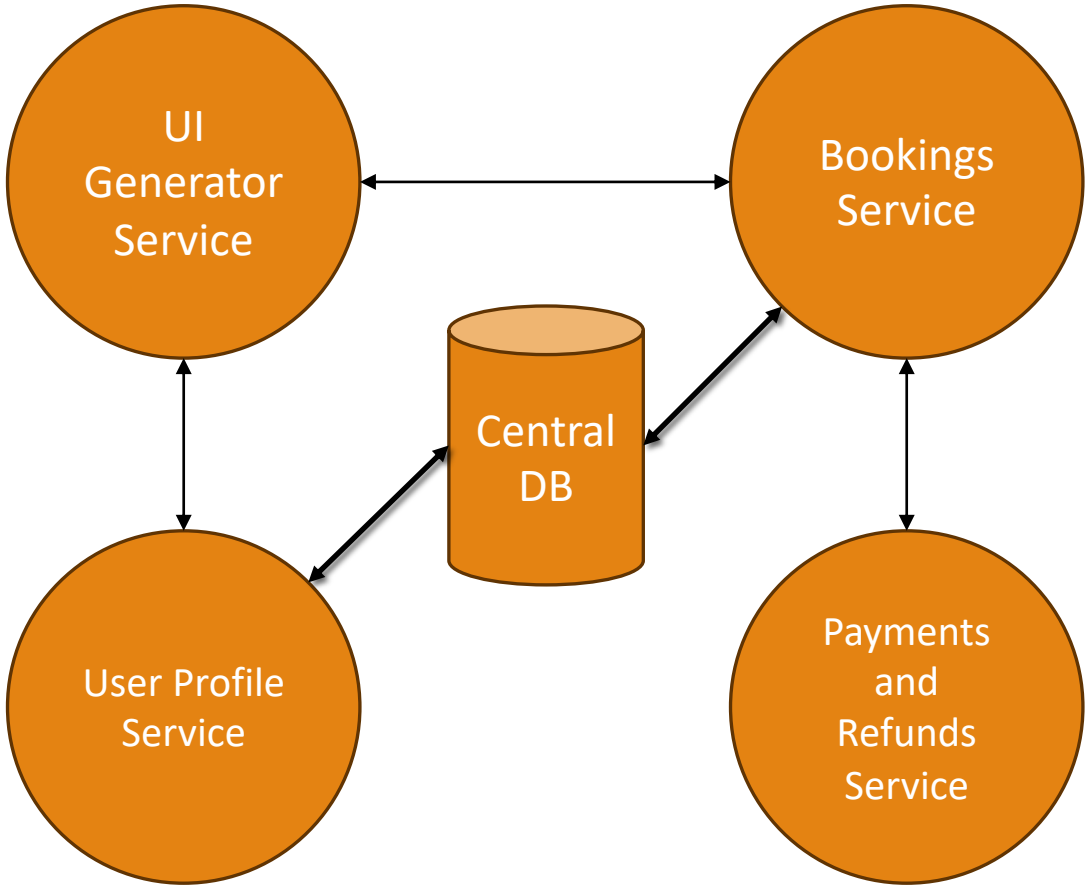
- We also looked at some ways to document it as well

Today we will look at another crucial aspect of designing systems …

- … especially, large-scale systems !!

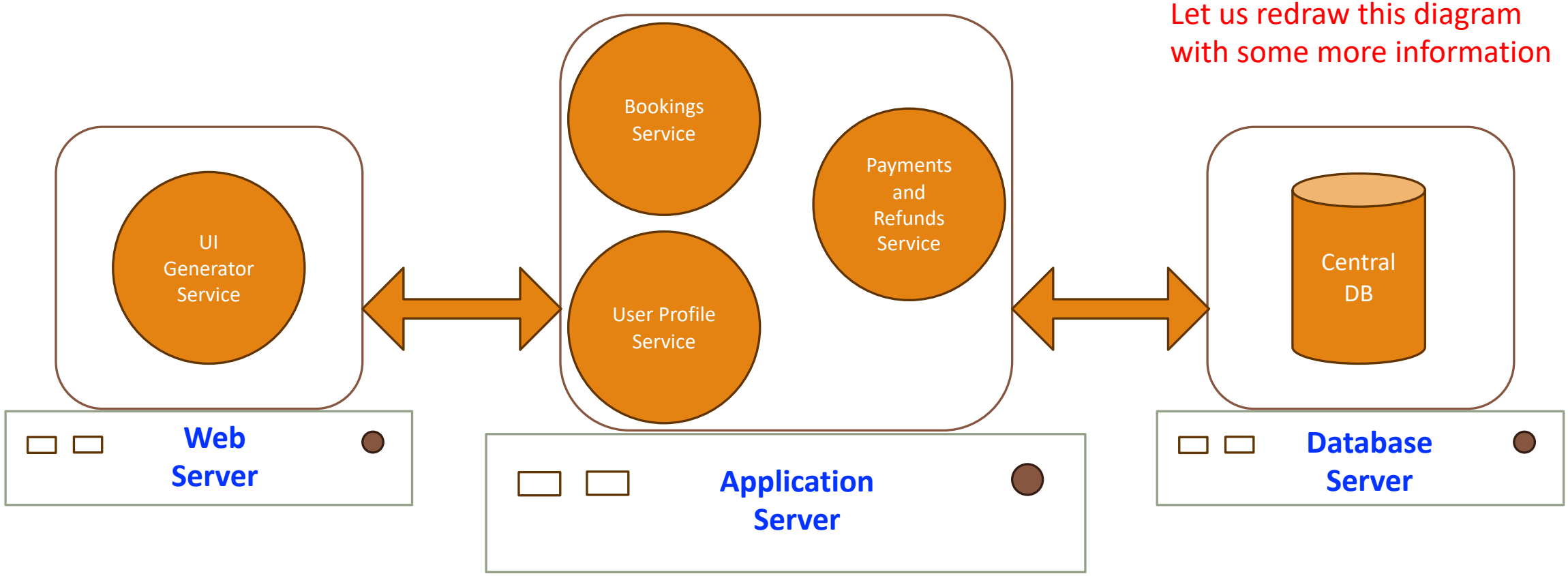
Revisit – Online Ticket Booking System

Process View



Revisit – Online Ticket Booking System

Let us redraw this diagram with some more information



Modifying the Ticket Booking System

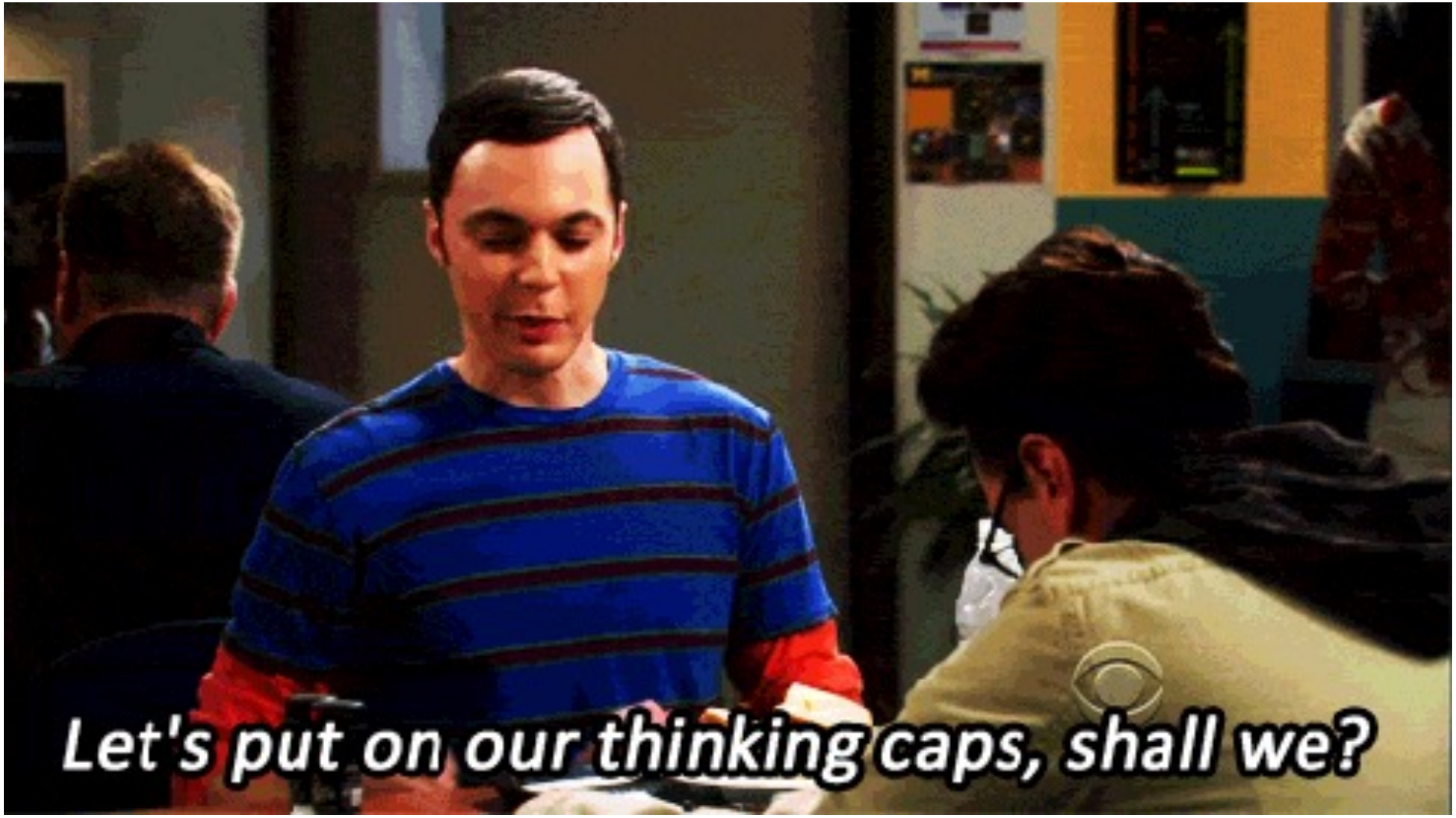
Assume that you booked a ticket with the Airline

- Everything went well, you paid the money, and they assured you a seat on the flight

But just then, catastrophe struck

- The server that was hosting the Database, faced a Hard drive failure !!
- Now what? The Airline now has no records that you booked a ticket with them

Clearly, that is not acceptable; so, what should we do?


Let's put on our thinking caps, shall we?

Modifying the Ticket Booking System

Assume that you booked a ticket with the Airline

- Everything went well, you paid the money, and they assured you a seat on the flight

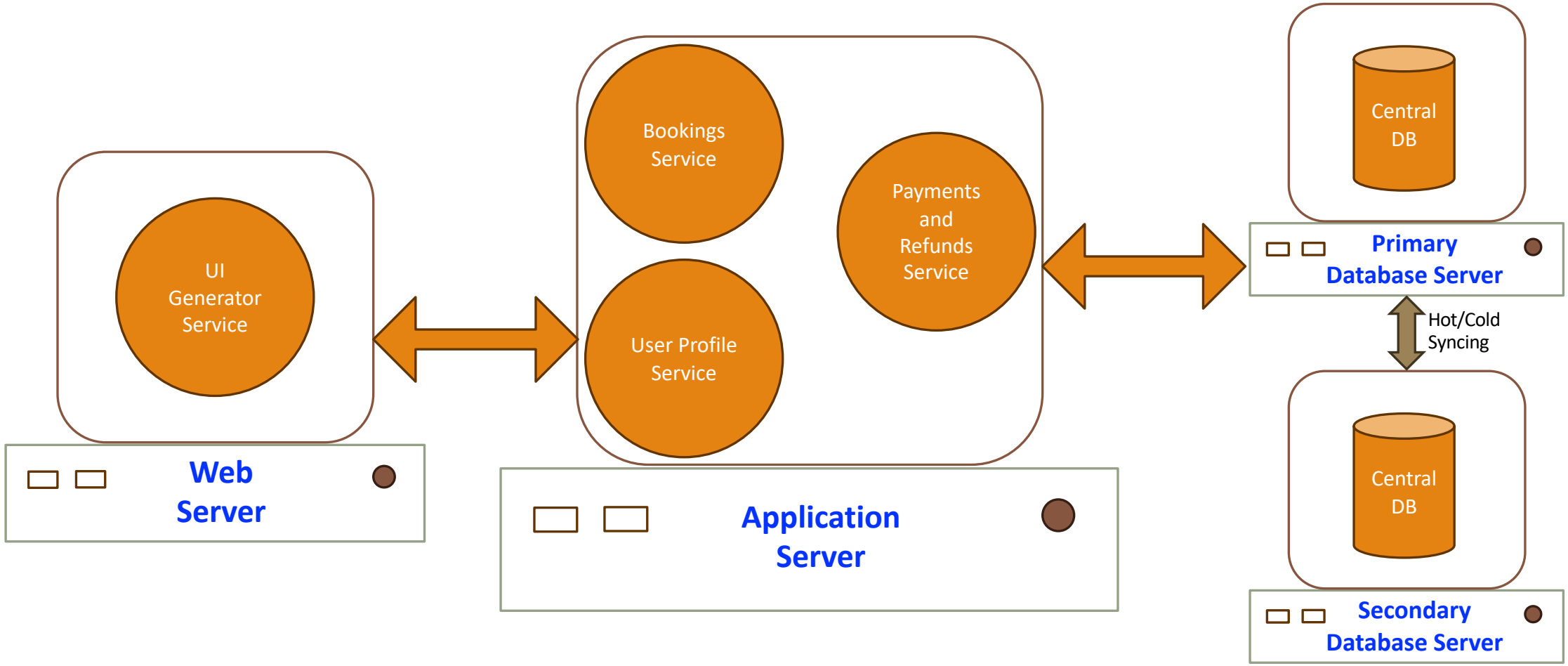
But just then, catastrophe struck

- The server that was hosting the Database, faced a Hard drive failure !!
- Now what? The Airline now has no records that you booked a ticket with them

Clearly, that is not acceptable; so, what should we do?

- There is a natural response – “we need to have a backup of the database”

Revisit – Online Ticket Booking System



Modifying the Ticket Booking System

Assume that you booked a ticket with the Airline

- Everything went well, you paid the money, and they assured you a seat on the flight

But just then, catastrophe struck

- The server that was hosting the Database, faced a Hard drive failure !!
- Now what? The Airline now has no records that you booked a ticket with them

Clearly, that is not acceptable; so, what should we do?

- There is a natural response – "we need to have a backup of the database"
- The backing up can be of two types – *Hot* or *Cold*

Hot Syncing means that the state of the two databases shall always (well, almost always) be the same

- For instance, a write operation must be executed by both instances, before we declare it a success

Cold Syncing means that the state of the two databases shall be synced "periodically"

- The period could be a few minutes, a few hours, or even, a few days (depending on criticality of the data)

Architectural Tactics

What did we just do?

- Provided that a failure in the Database server would have brought down the whole web application, …
- … we have essentially "increased" the *Availability* attribute of the system
- In fact, considering that we have provided protection the Airline's reputation as well, …
- … actually, we have also improved the system's *Reliability* attribute

But think about this – what happens to the *Performance* attribute in the case of "hot syncing"?

- Take it as an axiom – managing states of databases in a distributed setting is always going to be a challenge
- Although it can be minimised to a good extent, there may still be a "syncing" overhead on Performance
- What you just did is called the application of the *Redundancy* Architectural Tactic

A definition for an Architectural tactic is as shown below:

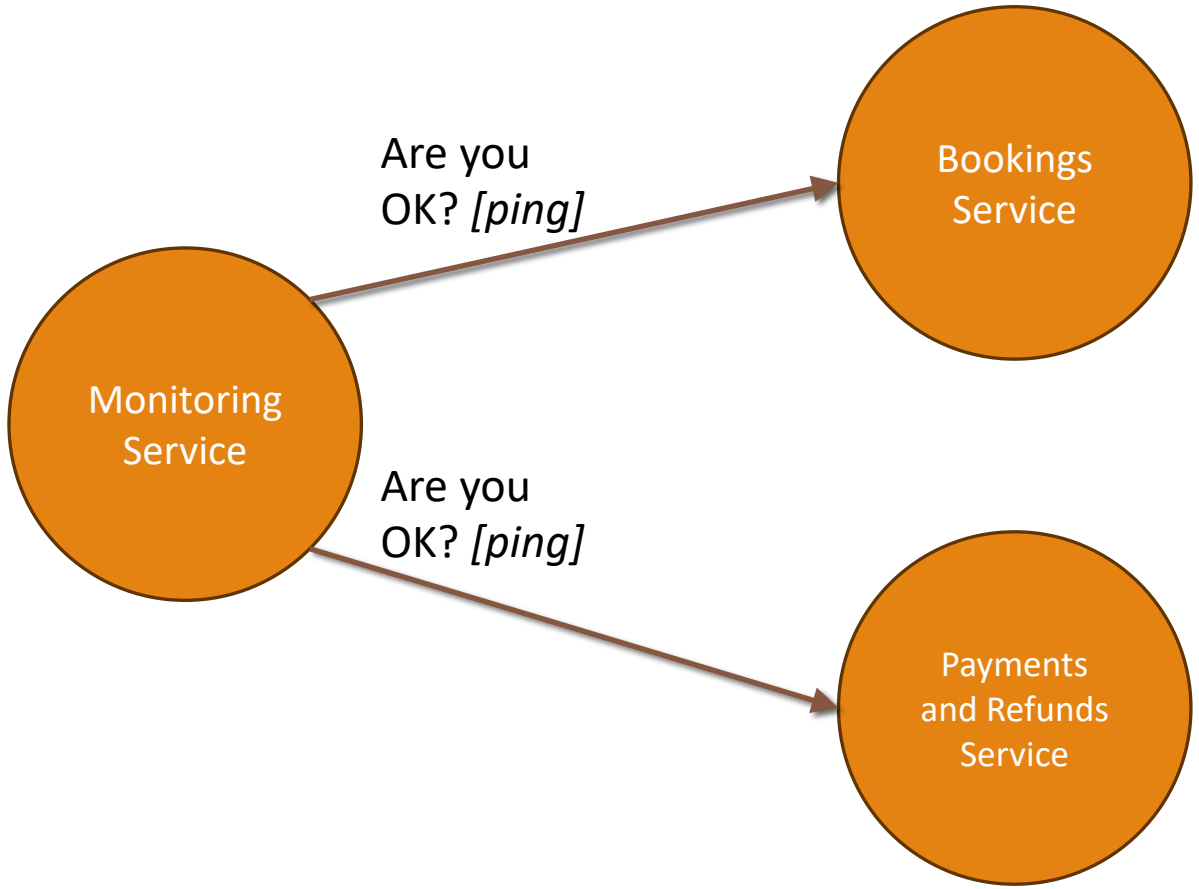
- *"A tactic is a design decision that influences the achievement of a quality attribute response – tactics directly affect the system's response to some stimulus."*
  – [Software Architecture in Practice, Len Bass, Paul Clements and Rick Kazman, 3rd Edition, Chapter 4]

Architectural Tactics for Availability

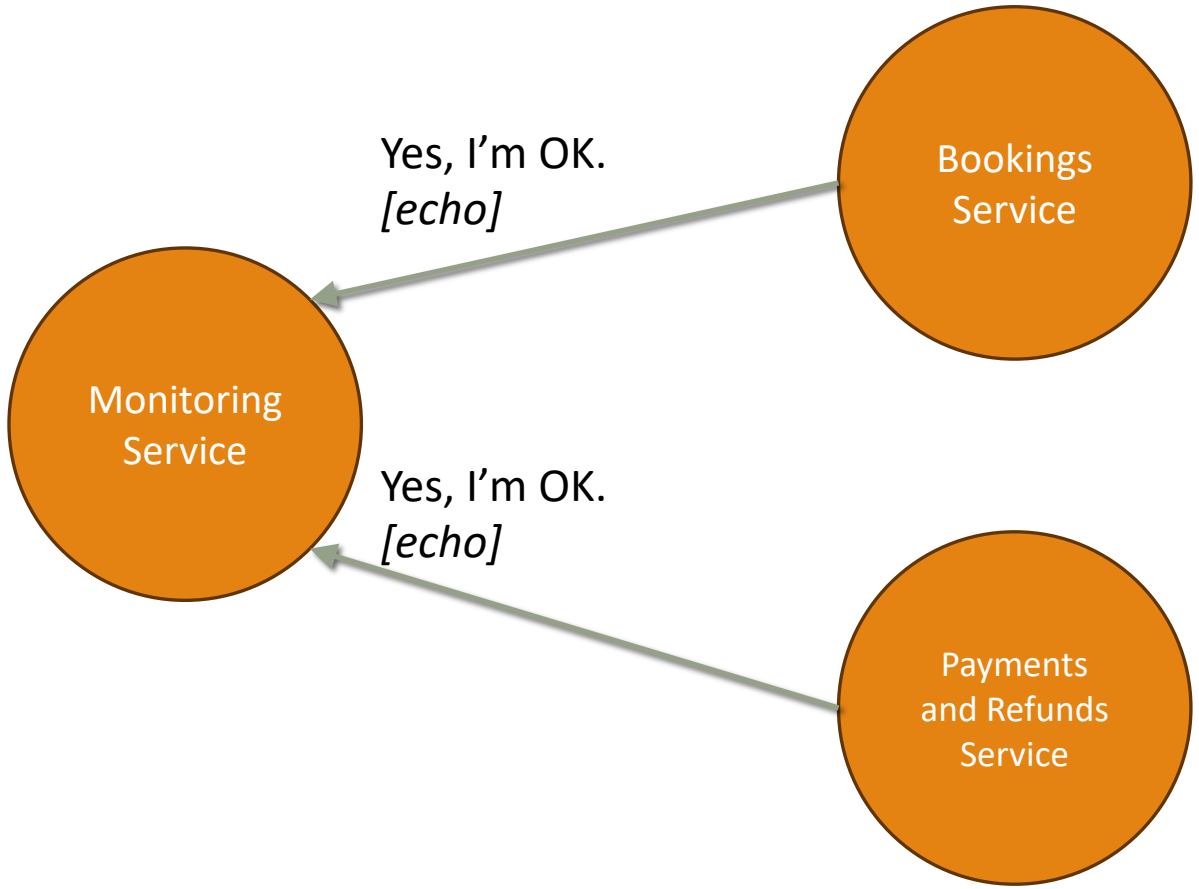
Tactics to *Detect* Failures

- Ping/Echo – Send a ping to a live component, and ask the component to respond with an echo
- Heartbeat – Ask the live component to keep sending a message to inform about its liveness
- A special component in the system, called a *Monitor*, keeps track of the availability of other components

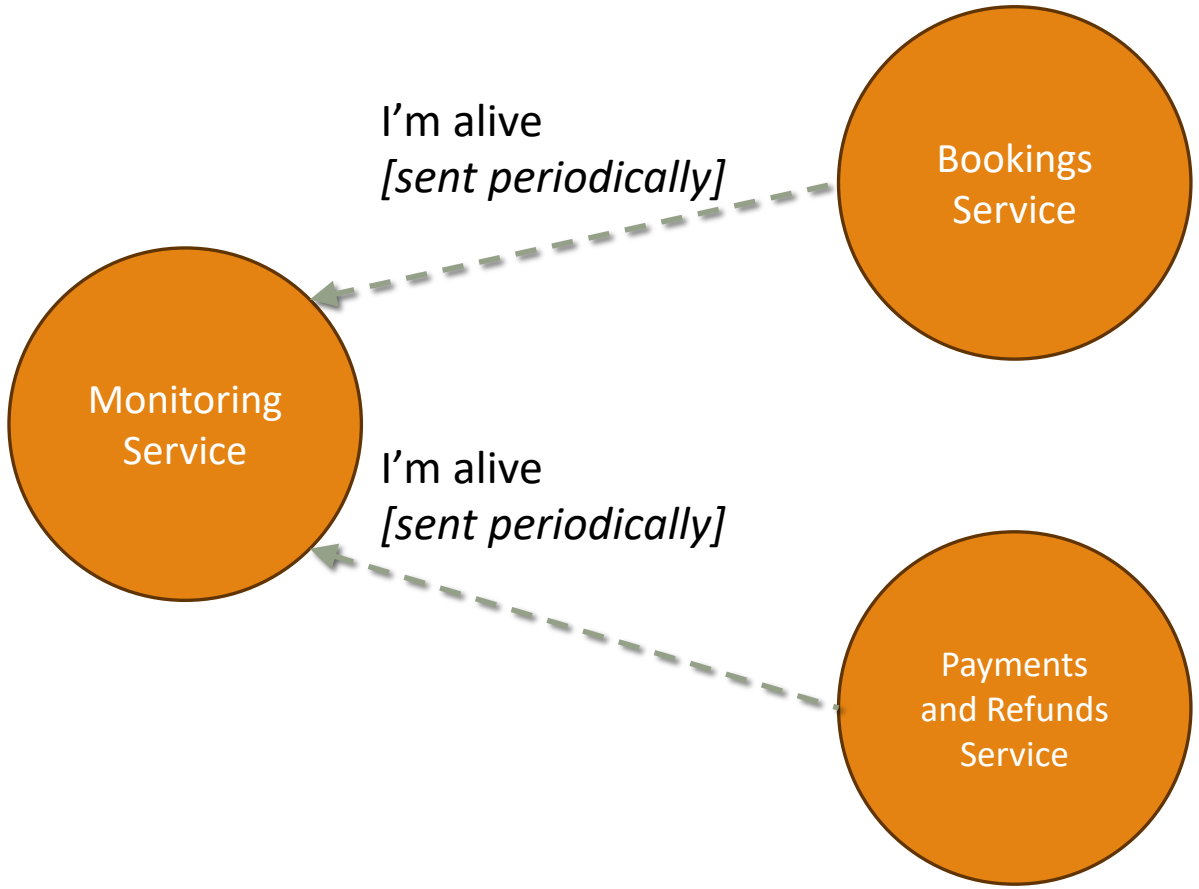
Ping/Echo


Are you
OK? *[ping]*
Bookings
Service
Monitoring
Service
Are you
OK? *[ping]*
Payments
and Refunds
Service

Ping/Echo


Yes, I'm OK.
[echo]
Bookings Service
Monitoring Service
Yes, I'm OK.
[echo]
Payments and Refunds Service

Heartbeat


I'm alive
[sent periodically]
Bookings Service
Monitoring Service
I'm alive
[sent periodically]
Payments and Refunds Service

Architectural Tactics for Availability

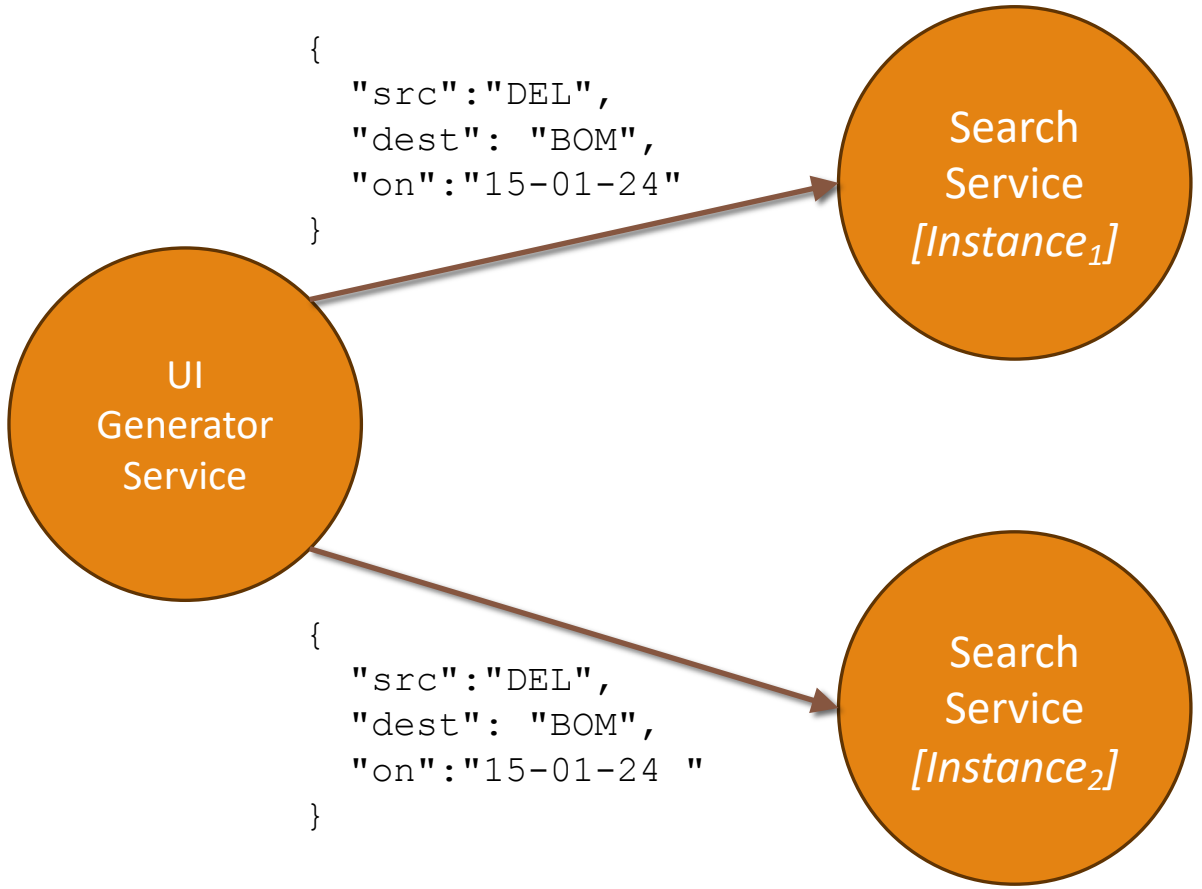
Tactics to *Detect* Failures

- Ping/Echo – Send a ping to a live component, and ask the component to respond with an echo
- Heartbeat – Ask the live component to keep sending a message to inform about its liveness
- A special component in the system, called a *Monitor*, keeps track of the availability of other components

Tactics to *Recover* from Failures

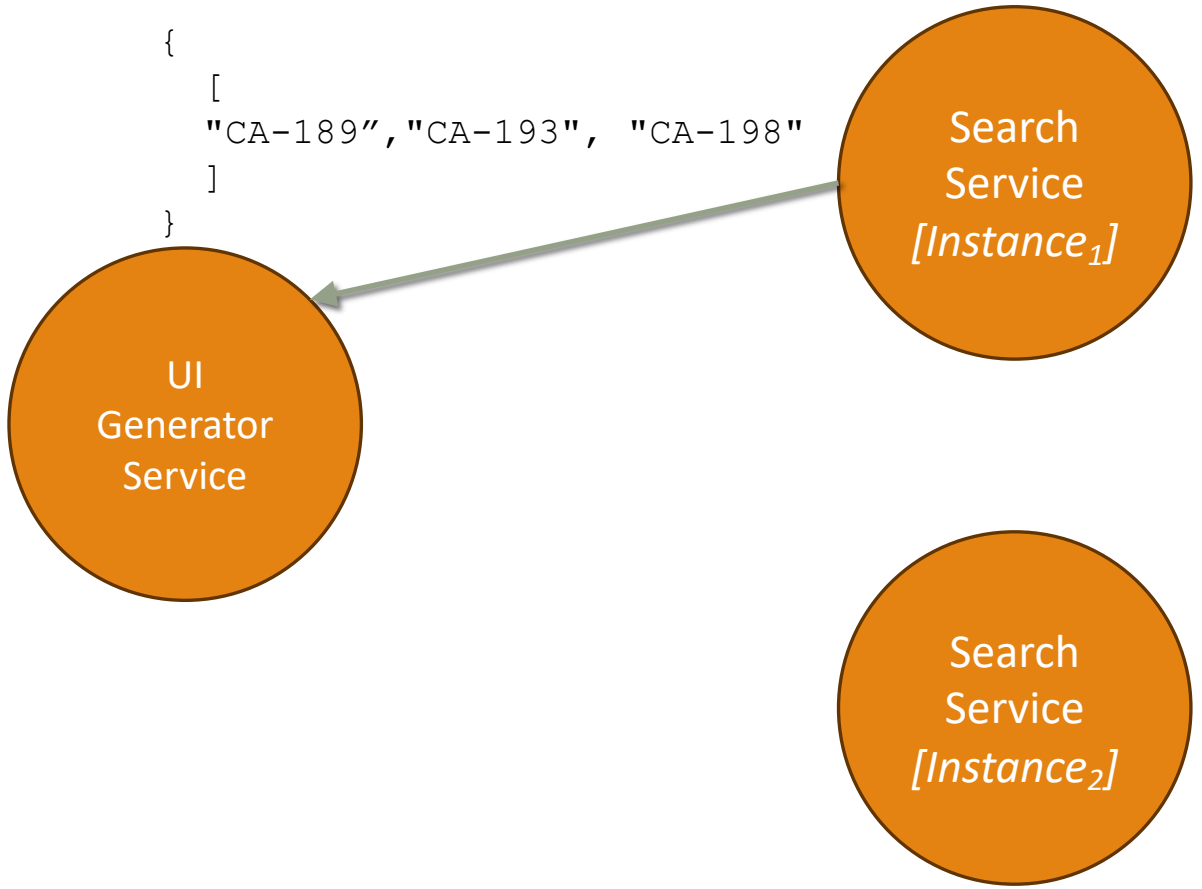
- Active Redundancy – Multiple “live” instances of a component; all receive every input request (all are in sync)
- Passive Redundancy – Multiple “live” instances of a component; some are active, others are synced periodically
- Cold Spare – New instance of a component is forked and made “live” when an instance dies
- Rollback Support – Build systems with “rollback” features (e.g., using Logs)

Active Redundancy



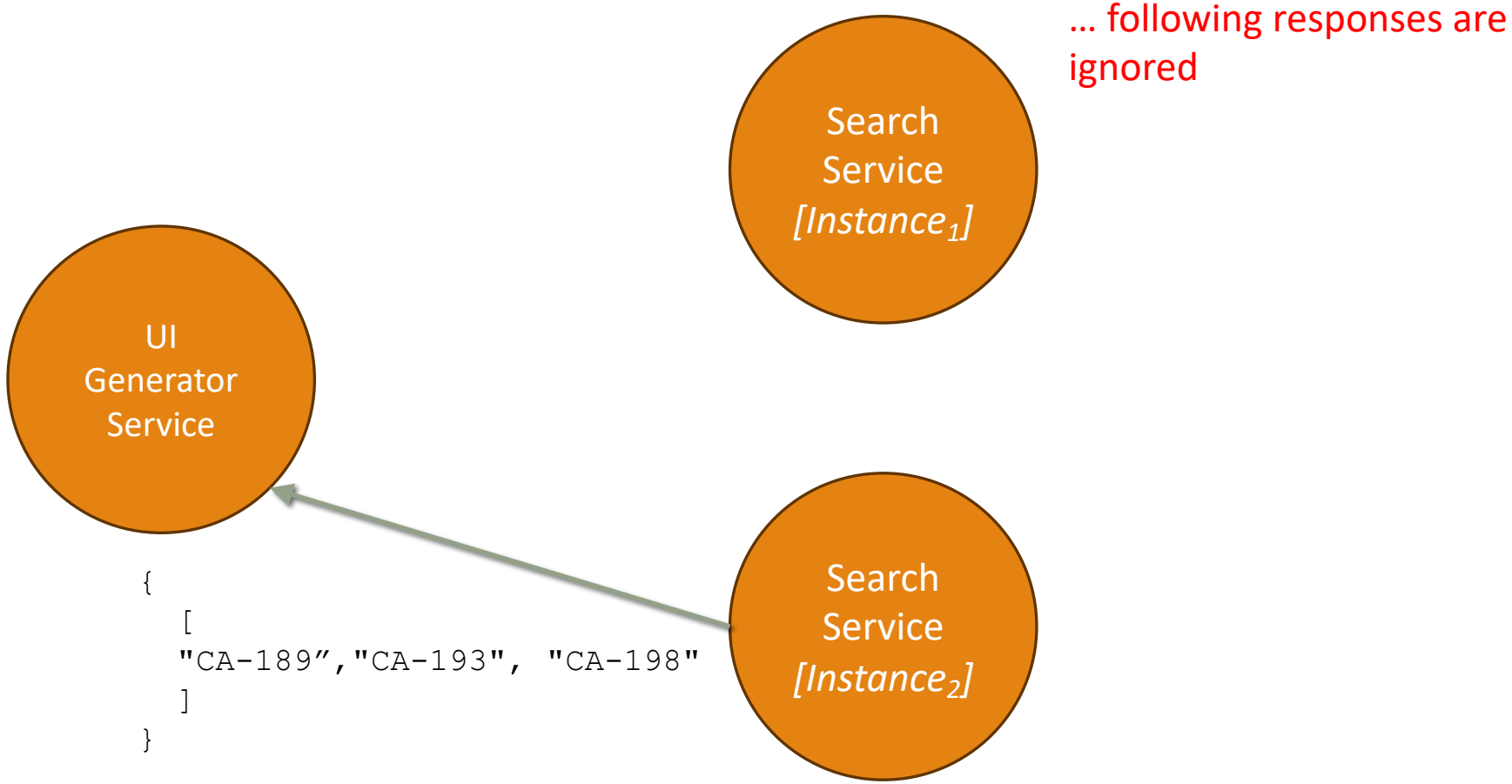
Requests are simultaneously sent to all “active” instances

Active Redundancy

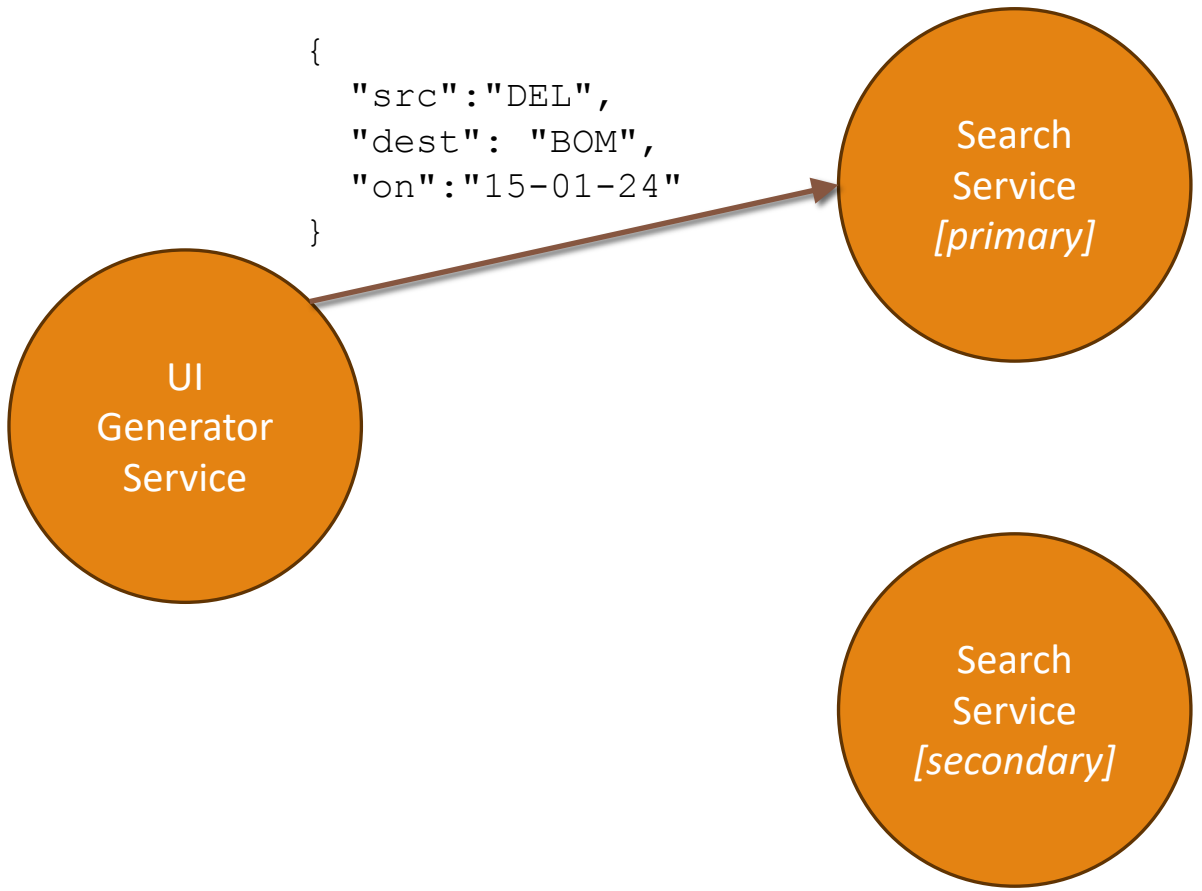


The first received response is taken …

Active Redundancy

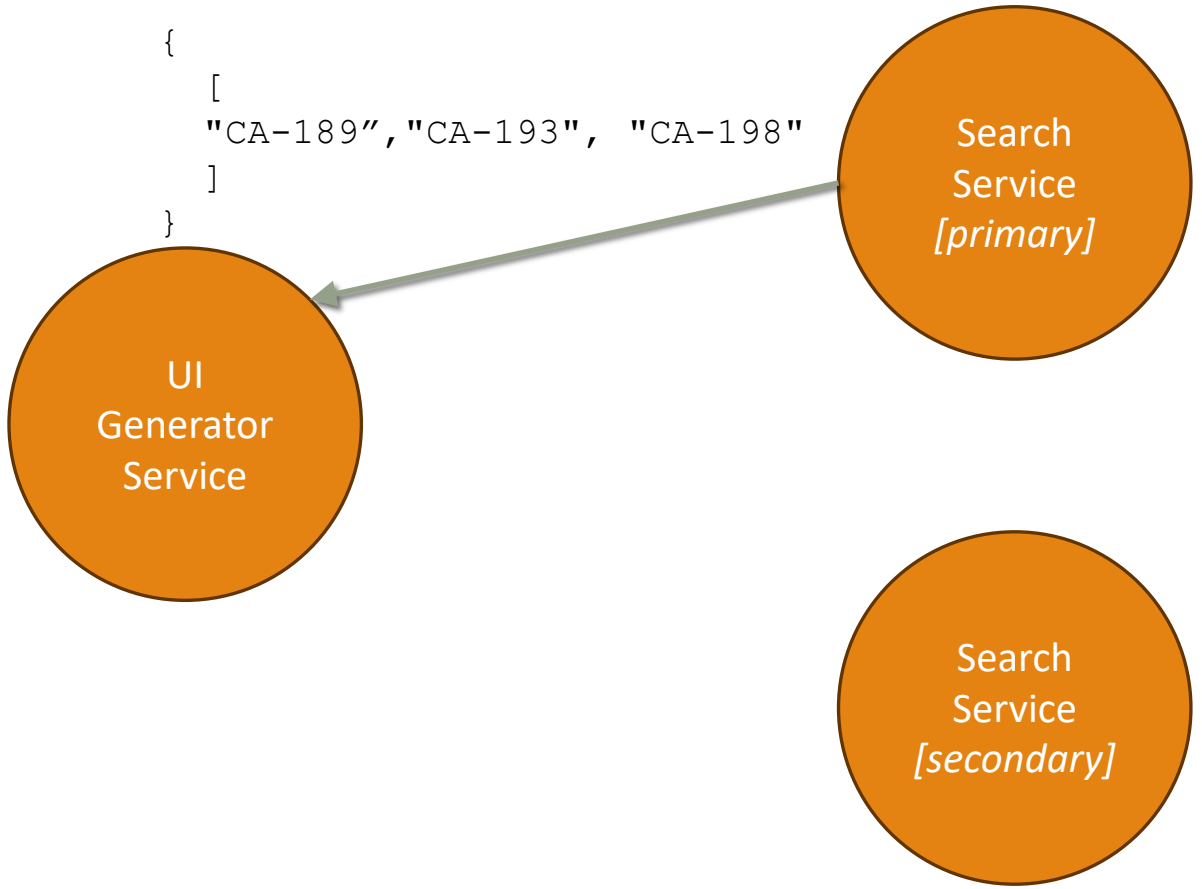


Passive Redundancy



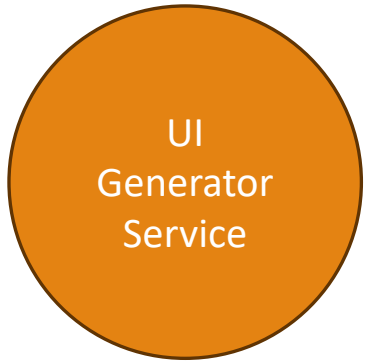
Requests are sent to the "primary" instance only

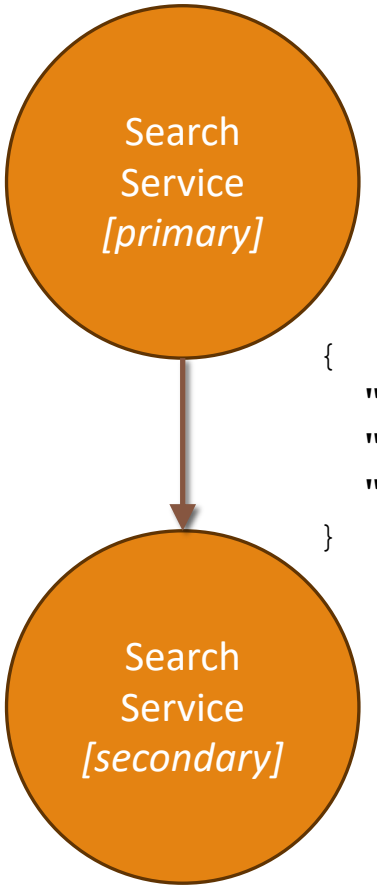
Passive Redundancy



The ”primary” instance responds to the requests

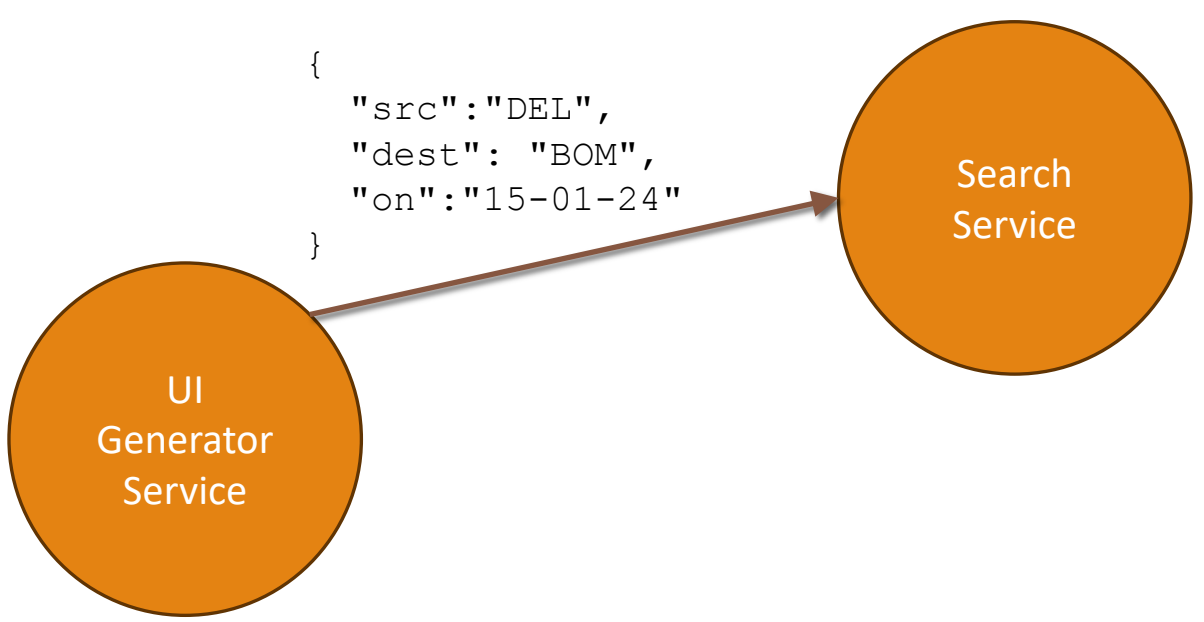
Passive Redundancy





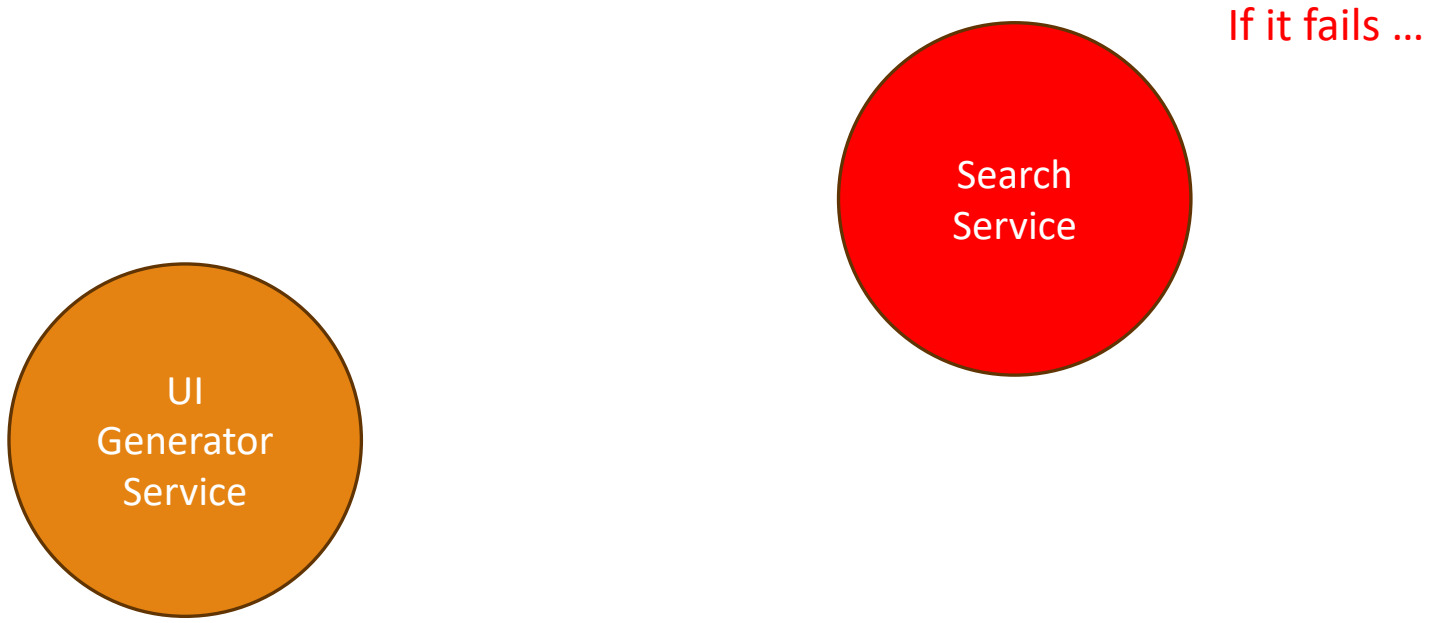
The ”primary” instance also communicates with secondary instances to keep them “fresh” (i.e., their states being close to the primary)

Spare

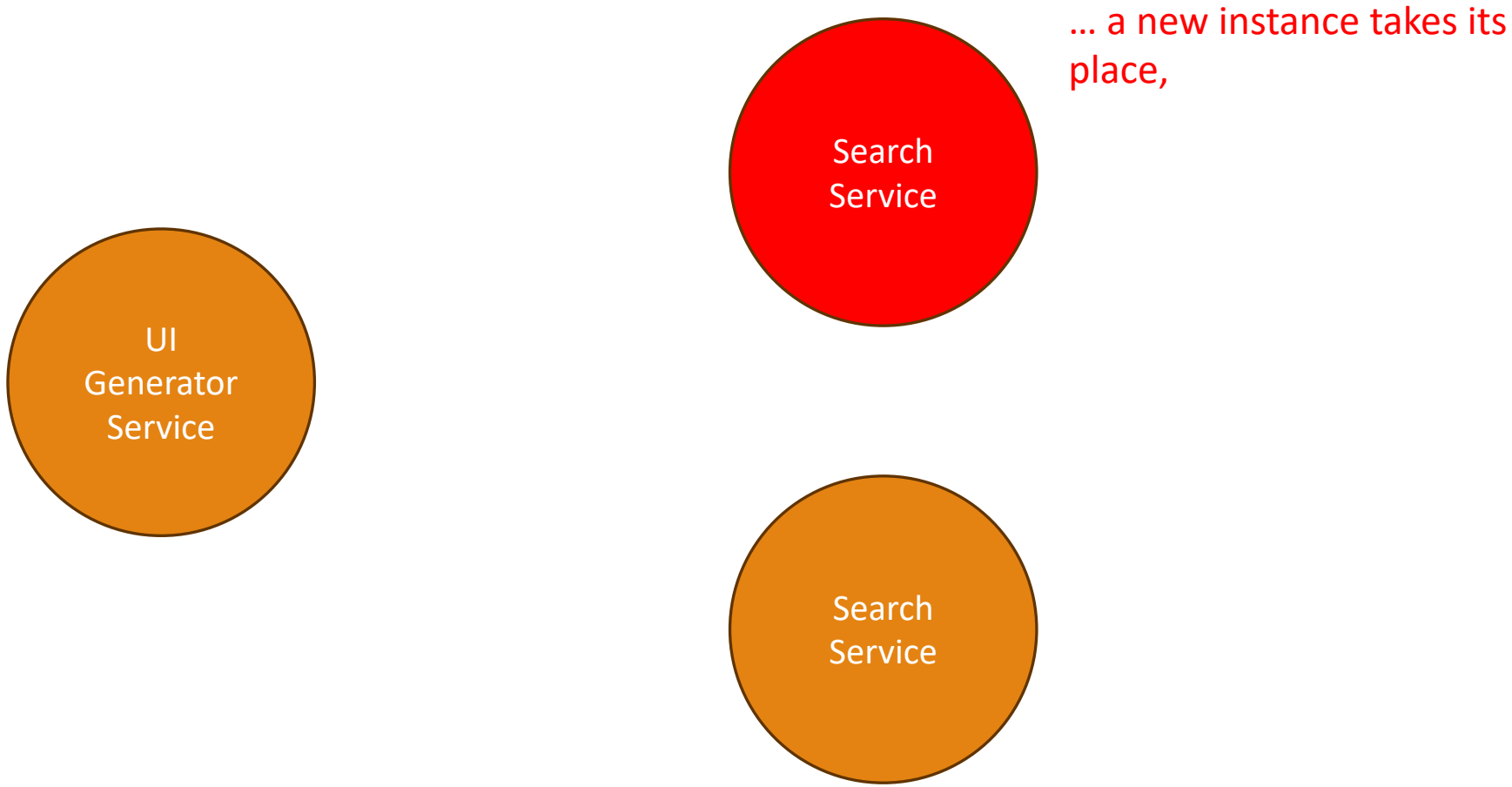


There is only one active instance

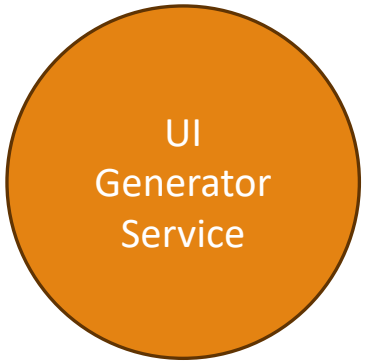
Spare



Active Redundancy



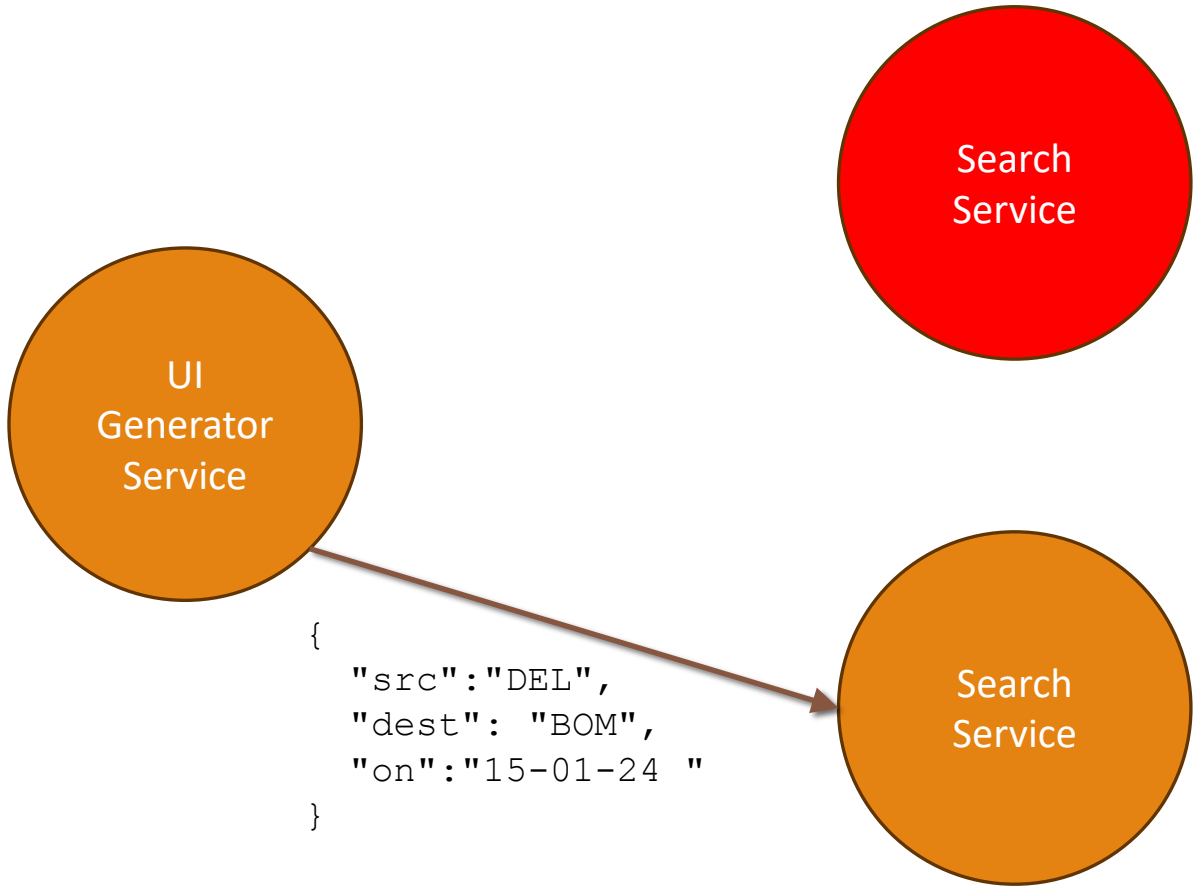
Active Redundancy



… a new instance takes its place,

As evident, its applicability is rather limited (e.g., in “state-less” scenarios, where instances work without maintaining any inherent state) !!

Active Redundancy



Some requests may be lost and have to be retried !!

Architectural Tactics for Availability

Tactics to *Detect* Failures

- Ping/Echo – Send a ping to a live component, and ask the component to respond with an echo
- Heartbeat – Ask the live component to keep sending a message to inform about its liveness
- A special component in the system, called a *Monitor*, keeps track of the availability of other components

Tactics to *Recover* from Failures

- Active Redundancy – Multiple "live" instances of a component; all receive every input request (all are in sync)
- Passive Redundancy – Multiple "live" instances of a component; some are active, others are synced periodically
- Cold Spare – New instance of a component is forked and made "live" when an instance dies
- Rollback Support – Build systems with "rollback" features (e.g., using Logs)

In addition, a way to *Prevent* Failures includes regular check-ups, and *scheduled maintenance*

Architectural Tactics for Performance

Tactics that attempt to *Control Demand for Resources*

- Restrict Requests – Use rate limiting to disallow requests beyond allowed limits (e.g., with API Keys)
- Limit Event Processing Rate – Buffer requests for deferred processing (e.g., using Queues)
- Prioritise Events – If resources are limited enforce priorities over requests
- Reduce Overheads – Reduce intermediate components for a particular requests (though it impacts Modifiability)

Tactics that attempt to *Manage Resources* in a better way

- Increase Resources – Add more instances of the same component (keeping Horizontal Scaling in mind)
- Increase Concurrency – Serve different requests simultaneously through different instances
- Caching – Use caching at various levels to reduce repeated computation

These tactics are intended to reduce the overall Response Time of the application

- The idea is to use the available resources effectively to do the same …
- … and keep the application flexible enough so that adding more resources in small denominations …
- … keep the Response Time under acceptable limits

Revisit – Modifiability Quality Attribute

Availability and Performance attributes directly affect the user

- For instance, users would always want systems with high Availability and Performance

Some Quality Attributes are important mostly for the developers and the development firm

- For instance, whether the application can be easily ported to a new platform or not (called *Portability*)

We will discuss arguably the most important Quality Attribute for the developers now – *Modifiability*

- As evident, Modifiability is a measure of the effort required to "change" the application
- Changes may be required due to many reasons – from requirement changes to bug fixes to technology updates
- If the application is "hard to change", it makes every change costlier (and riskier) for the developers

It may be noted that in general, achieving better levels of every Quality Attribute is usually not possible

- There are, thus, trade-offs between multiple Quality Attributes, and the Architect must make the right decision
- Modifiability usually has a trade-off with Performance (i.e., increasing one, may reduce the other)
- We will discuss one such example in a minute

Architectural Tactics for Modifiability

Increase Modularity

- Instead of building fewer, large-sized modules, create more, small-sized modules
- A positive side-effect of this tactic is that different modules may be assigned to different manpower
- A negative side-effect is that the complexity of the application increases due to more inter-module calls

Increase Cohesion

- Avoid doing too many things in one module – do one thing well rather than multiple things poorly

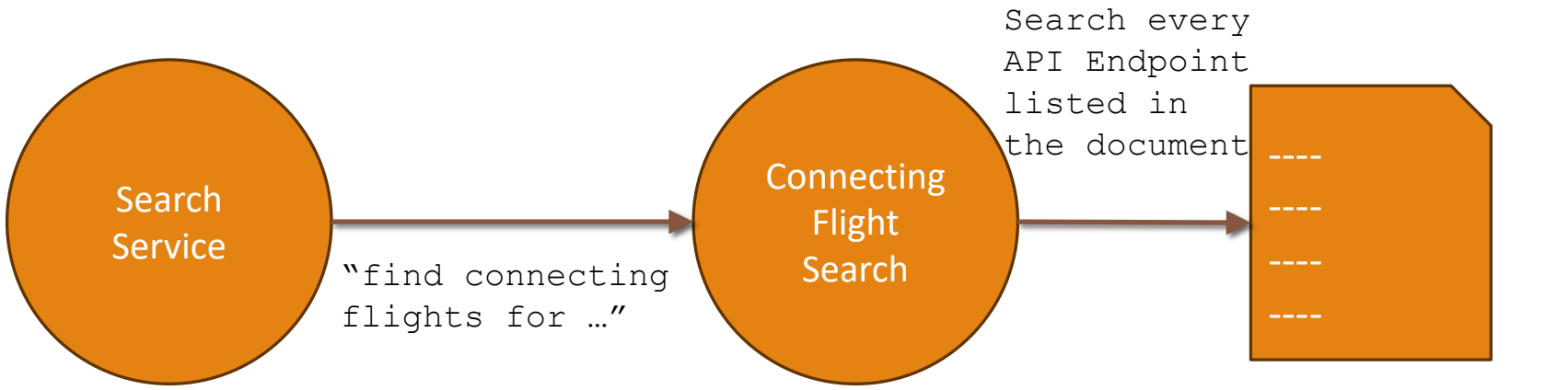
Decrease Coupling

- One way to do so is via intermediate components, that act as opaque layers between two or more components
- The intermediate component ensures that the overall flow is maintained in case of changes
- Note that inclusion of intermediate components affects the performance negatively – a trade-off example !!

Defer Binding

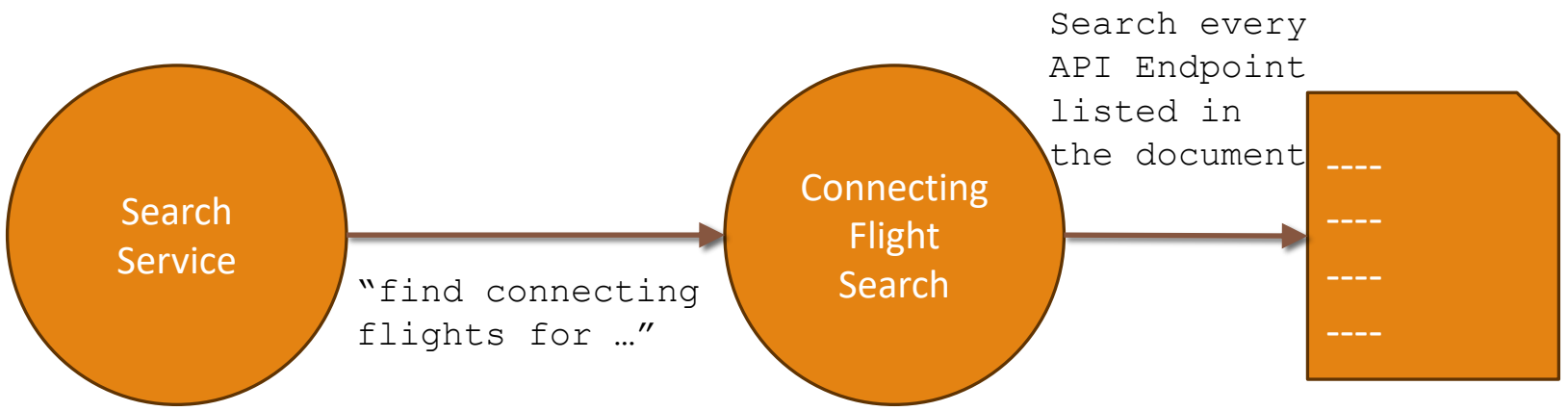
- Instead of supplying the code/logic, supply a location where the logic will be found (the logic itself, can change)

Defer Binding



List of External APIs provided by other airlines (subject to frequent changes)

Defer Binding



Instead of "hard coding" these URLs in source code, they are added to a configuration file, which can be manually updated by a person in Operations (without any knowledge of IT)

List of External APIs provided by other airlines (subject to frequent changes)

To sum up …

Different Quality Attributes may require different Architectural Tactics to achieve them in a system

The application of a Tactic may have side-effects

- For instance, a tactic for Availability may also increase the Reliability, but may decrease Performance

We choose Architectural Tactics based on the requirements and constraints

Homework

Think about the *Defer Binding* tactic

- What effect, if any, can it have on Availability and Performance attributes?

Further Reading

The book *Software Architecture in Practice* by Len Bass et al.

- While I am referring to the the 3rd edition of the book, I found out a source on the internet for the 2nd edition
- It should suffice to study that for most parts, including Architectural Tactics
- The link to the same is: https://people.ece.ubc.ca/matei/EECE417/BASS/index.html