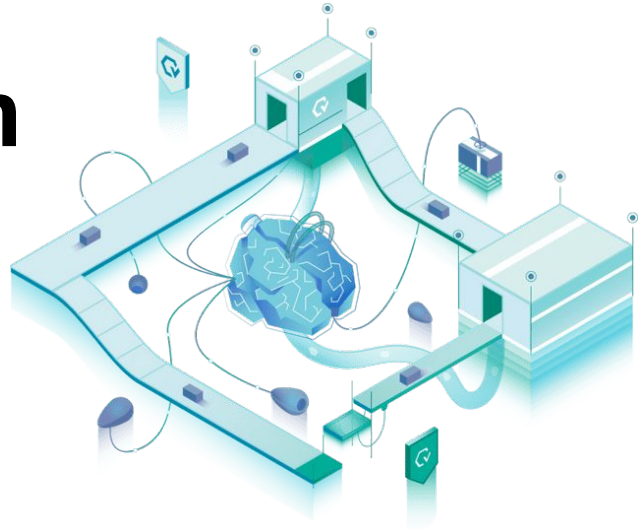
A Better Way to Manage Stateful Systems:

Design of a Stateful system for Robust Deployment and Observability



SREcon22 Asia/Pacific
7–9 December, 2022

Kazuki Higashiguchi, @hgsgtk
Sr. Site Reliability Engineer

Autify

What this talk is about

- Lessons learned from a project building and operating a stateful WebSocket server
- Design considerations of stateful systems maintaining long-living states

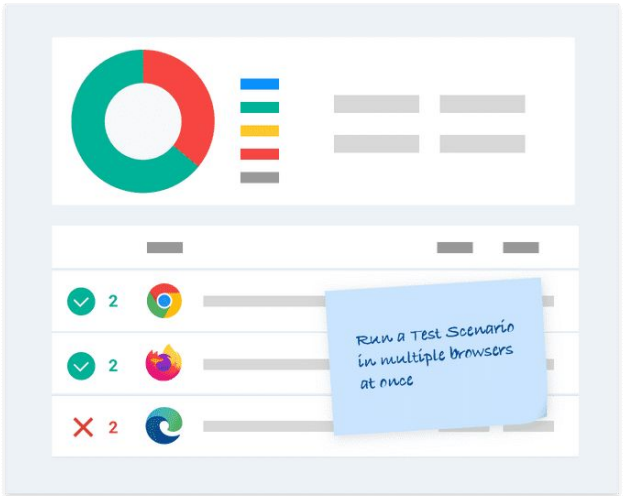
Topic	Technique
 Deployment	Implement zero-downtime automated deployment pipeline
 Observability	Make invisible internal states inside of an app observable

Case study - E2E test automation service for web apps

Build tests easily with no-code

Cross-browser testing

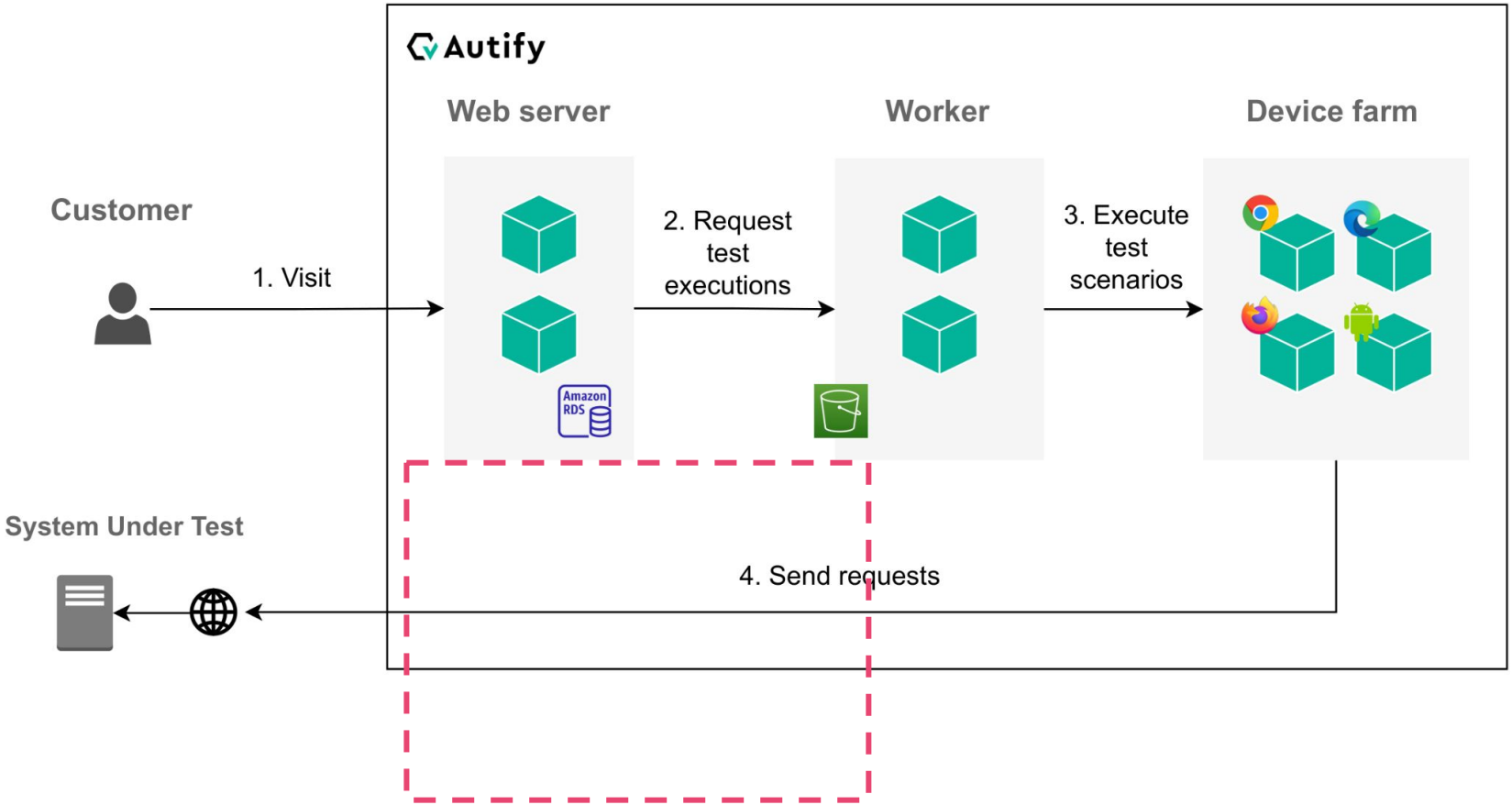
Parallel execution



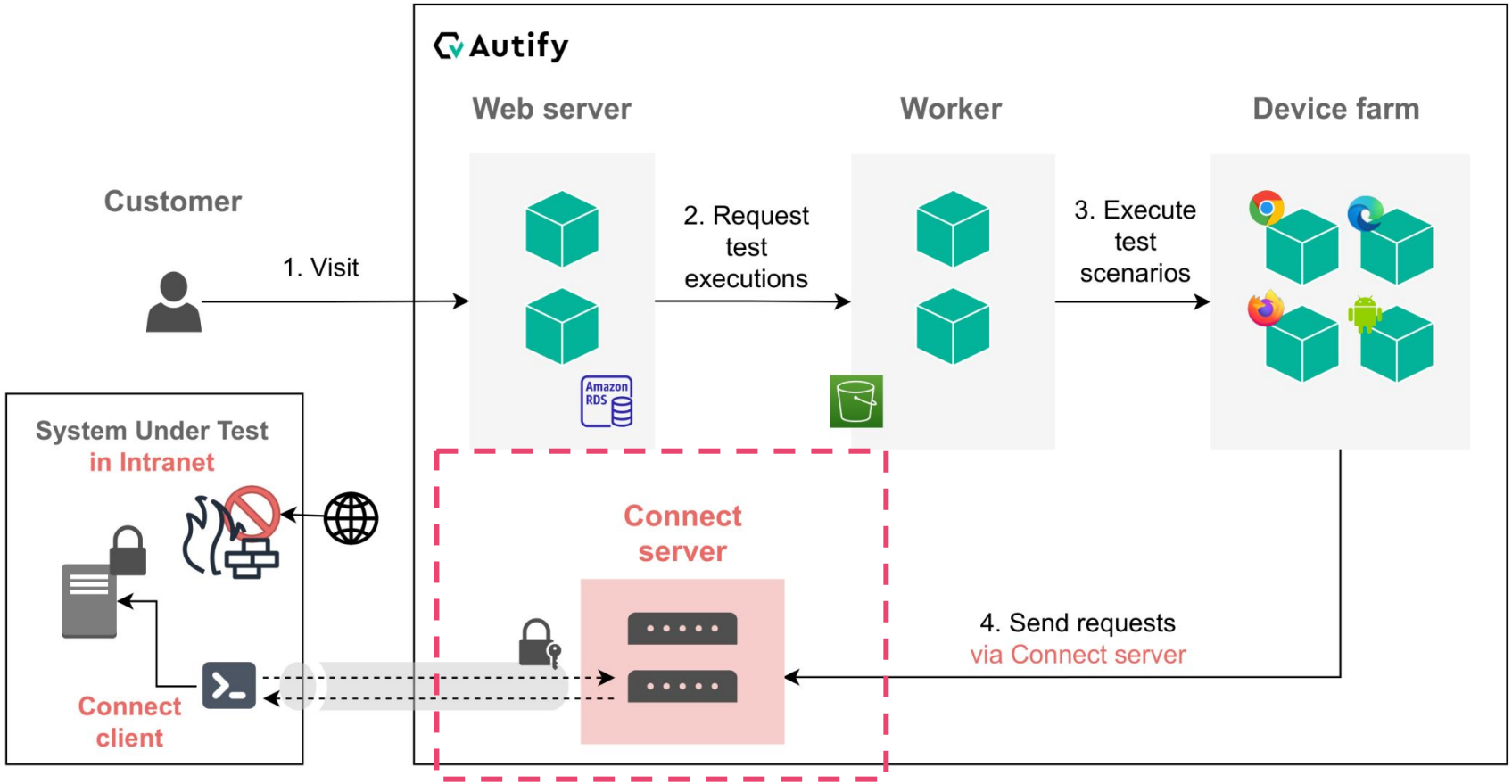
Case study - Primary infrastructure components

- Web server
  - Management console (e.g., create/edit test scenarios, run tests, view test results)
- Test execution engine (Worker)
  - Facilitate test executions, persist test results data
- Test execution environment (Device farm)
  - Browsers and devices running tests
- Connect server / client
  - Establish a secure bidirectional tunnel connection with customers' private networks

Case study - Test execution journey

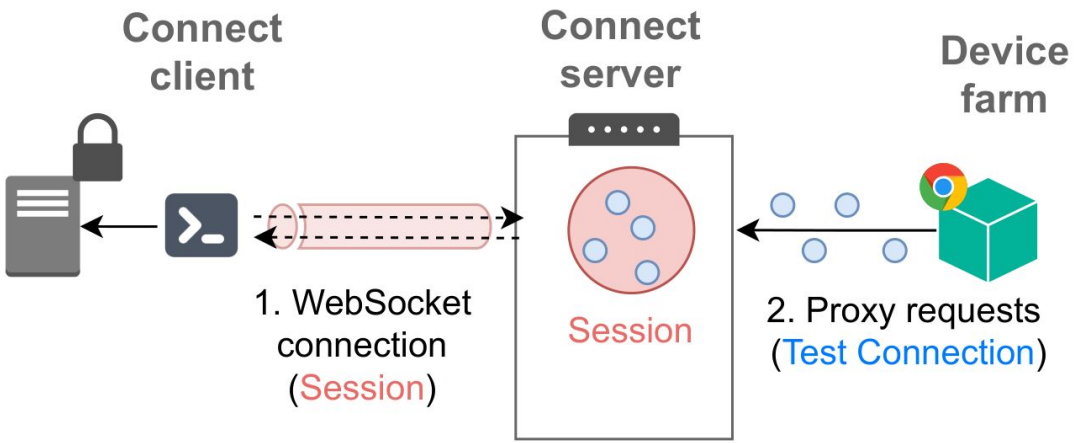


Case study - A stateful system in Autify



Case study - States in Connect server

1. Use WebSocket to establish a bidirectional tunnel connection (Session)
    a. Long-living connection between Connect server and Connect client
2. Proxy server to transfer requests (Test Connection) from Device farm
    a. Proxy HTTP(S) requests over Session



Design of a stateful system for...

Robust Deployment

Automated zero-downtime deployment for a stateful system

Why zero-downtime?

Background

Actively developed

- Go application developed by Autify
- Frequently deployed into production

High stability is required

- Used by customers right now to test applications on their private networks
- Reliable test execution infrastructure is essential for our business

Solution

Automation

- To keep deployment frequency
- Minimal errors

Zero-downtime

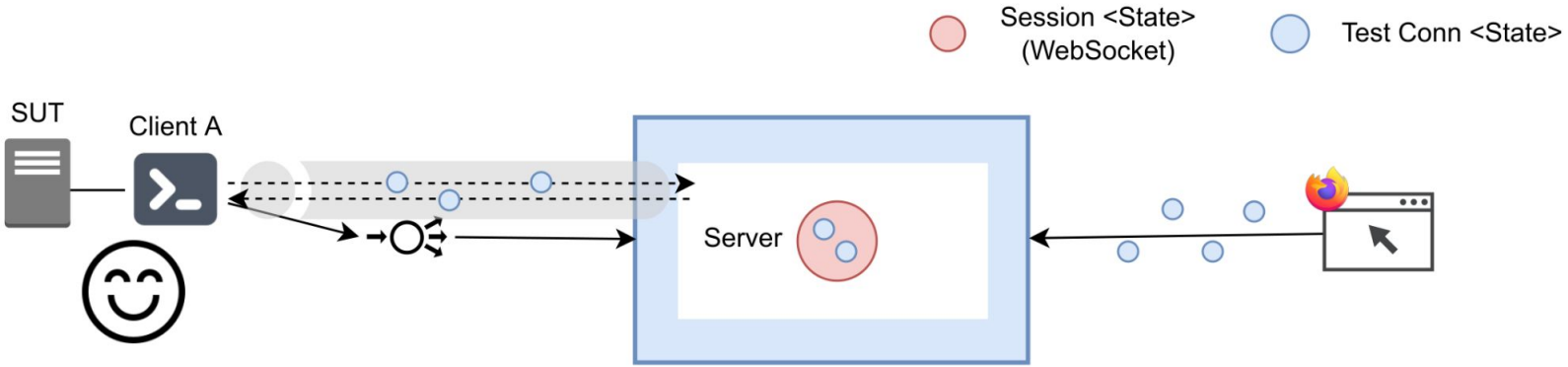
- To make this stateful system reliable

Blue-Green Deployment

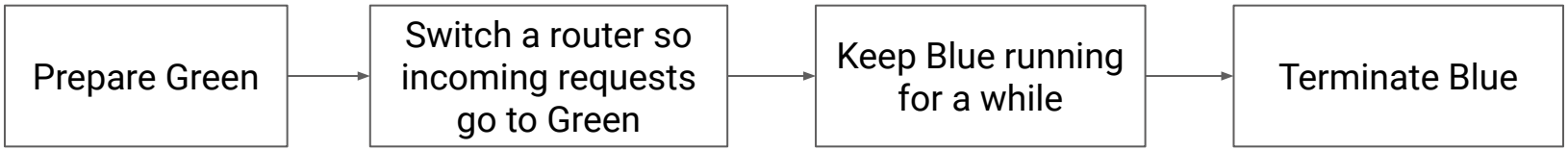
Rolling Update …etc

A failure story - with a Blue-Green deployment

A stateful server is running in Blue environment…

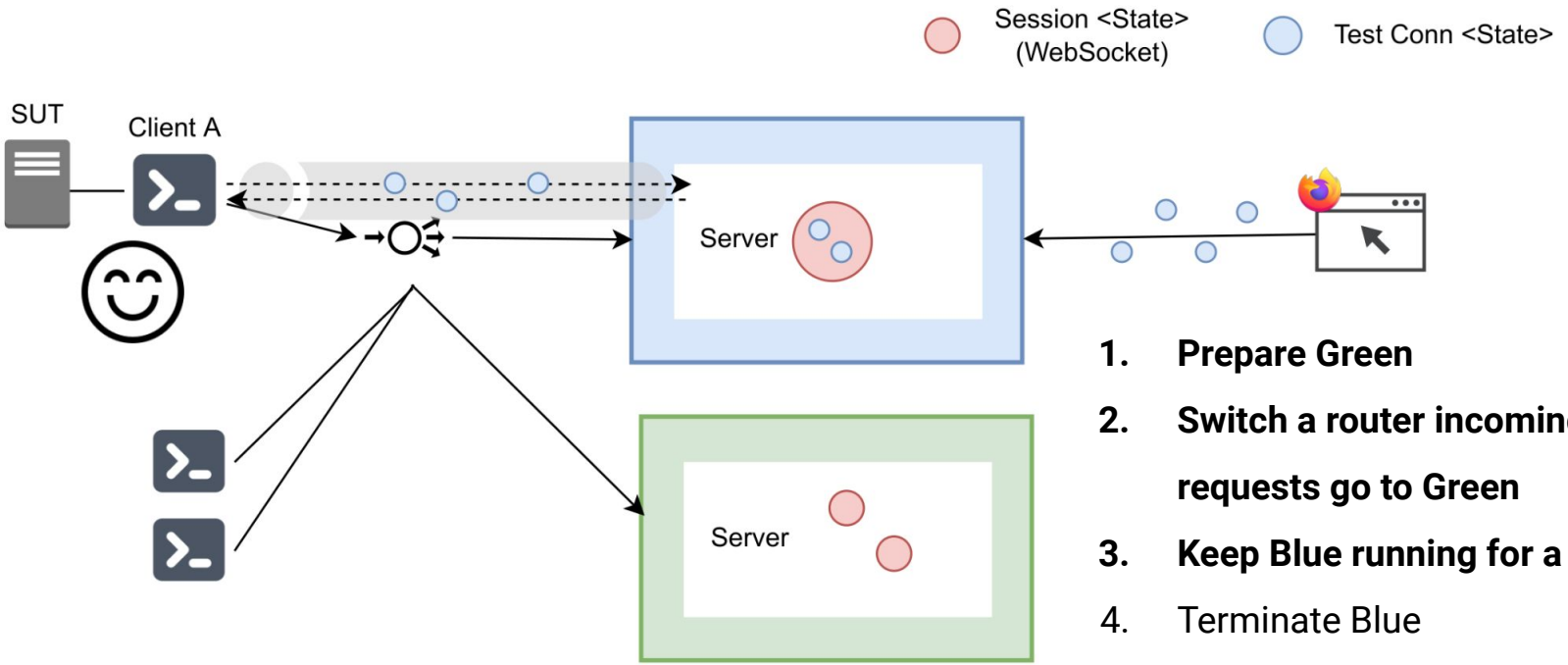


Steps in a typical Blue-Green deployment



A failure story - with a Blue-Green deployment

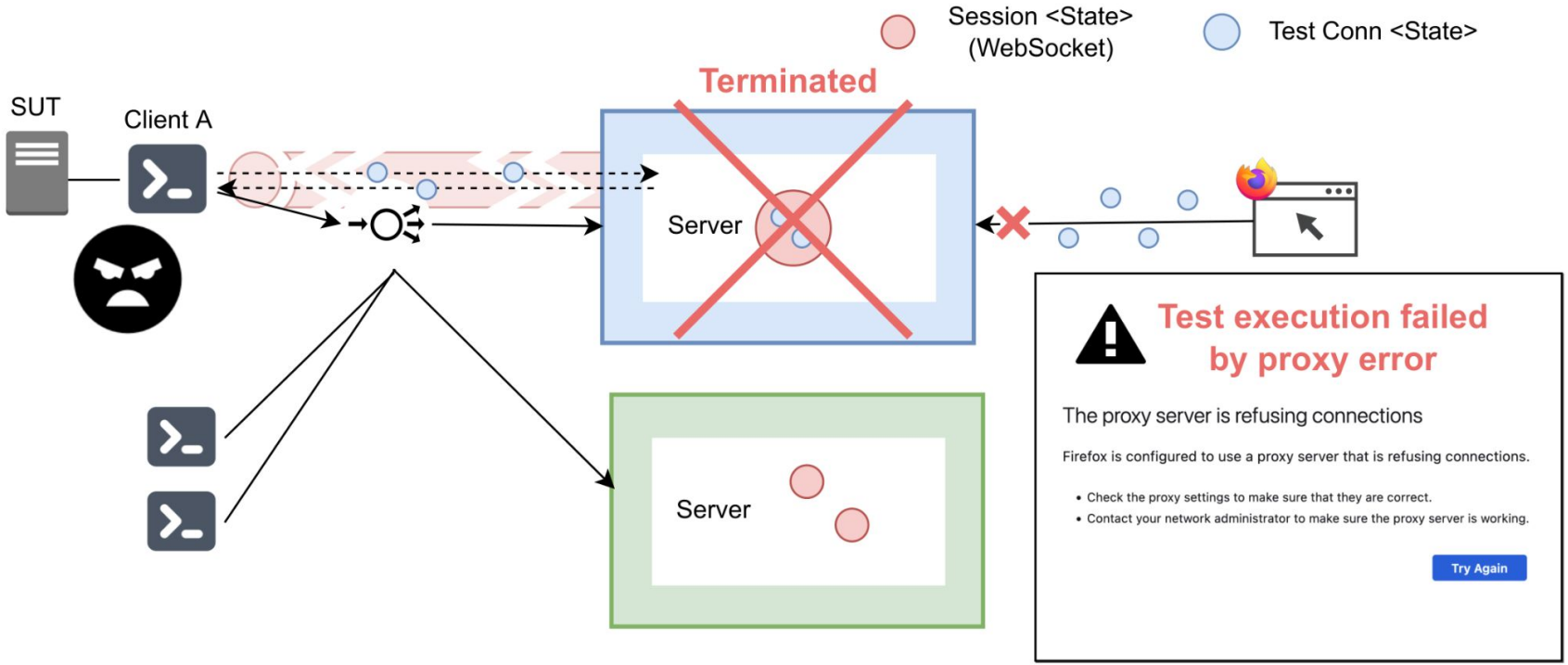
Start a new Blue-Green deployment…



1. **Prepare Green**
2. **Switch a router incoming requests go to Green**
3. **Keep Blue running for a while**
4. Terminate Blue

A failure story - with a Blue-Green deployment

Terminating a busy Blue server causes test execution errors



How to avoid errors during deployment

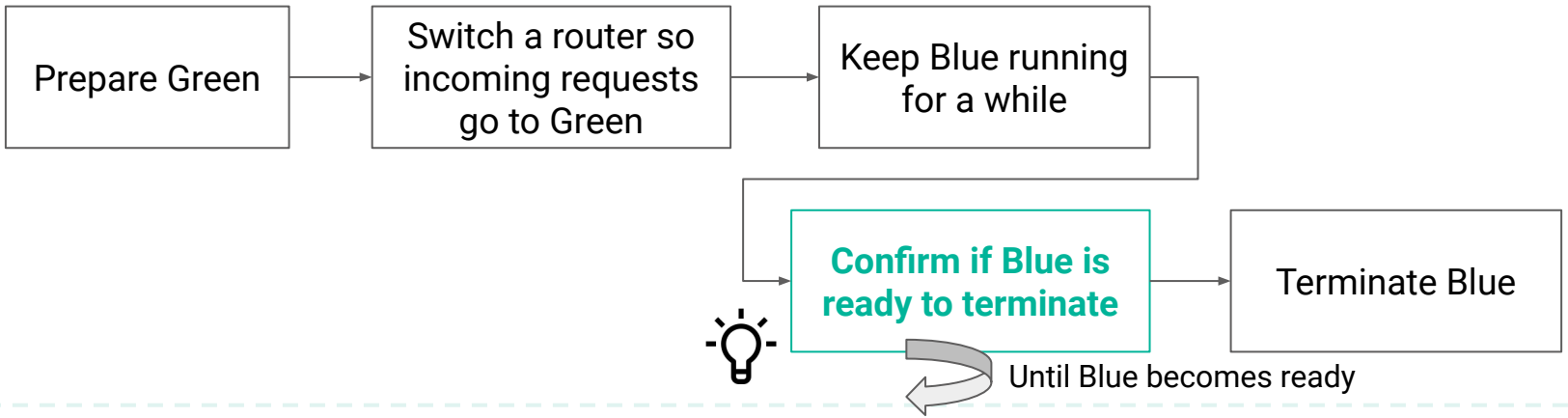
Symptom

Terminating Blue servers causes errors

Why?

Blue server is not ready to terminate if it has busy states internally

Solution - Verification step to confirm if Blue is ready to terminate



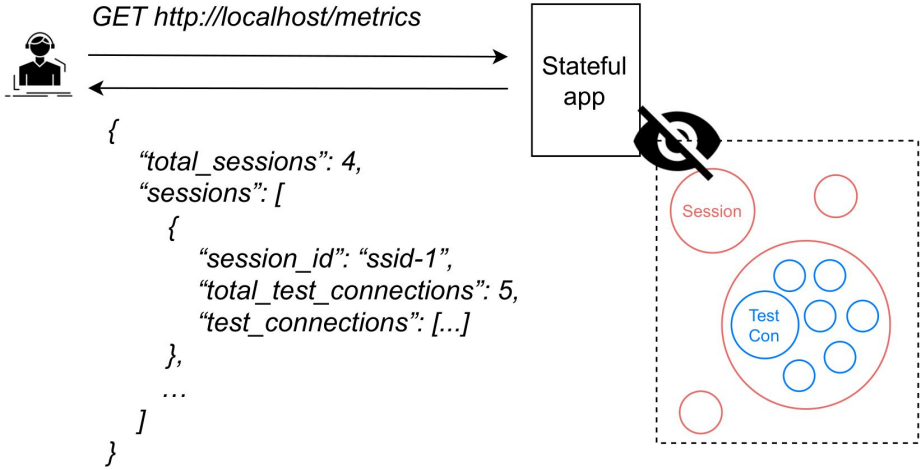
Candidates to see if Blue is ready to terminate

Possible solutions	Effectiveness
 Log monitoring Search for application logs	 • +1 Help to understand and measure how apps are behaving • -1 Just an event measured at some point. Not real-time status of app
 Infrastructure metrics Watch CPU usage, Network I/O, etc	 • -1 Mixed causes other than app • -1 Difficult to see what the application is currently doing
 See internal states inside of app Get data about internal states via app	 • +1 Single source of truth • +1 Real-time data • -1 Effort to design and implement codes to make apps observable

Design tips for making stateful apps observable (1)

HTTP endpoint to show metrics of internal states

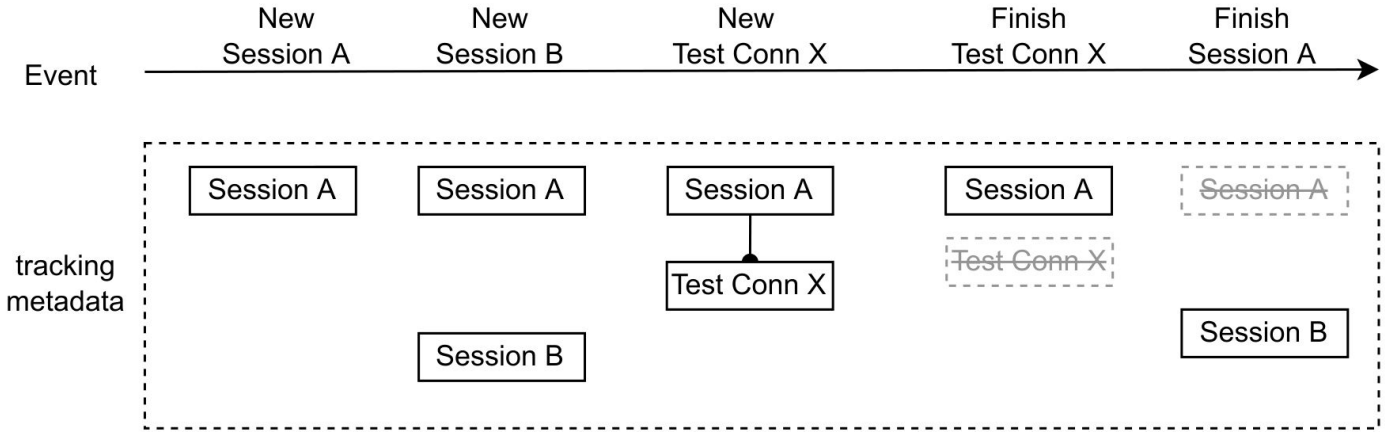
- e.g., “GET /metrics”
- Similar patterns, Health Endpoint pattern
- Easy to use from external programs



Design tips for making stateful apps observable (2)

Design stateful app to be able to see necessary details in each state

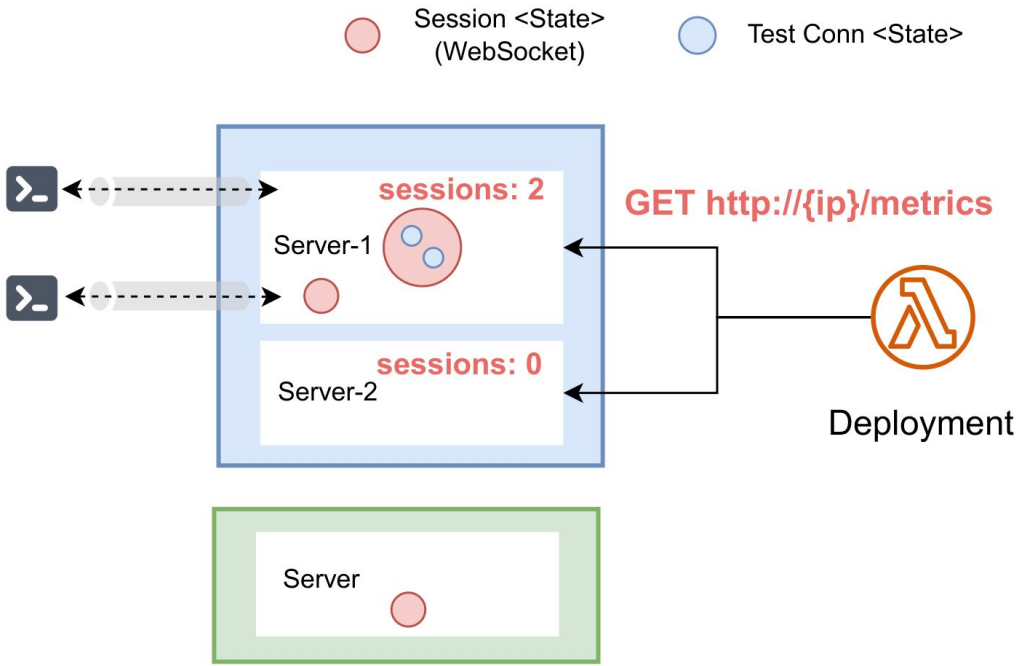
- To build metrics of internal states anytime
- e.g., Store metadata of ongoing states in-memory
- It should be considered at an earlier stage to avoid a huge rewrite



Verification steps to confirm if Blue is ready to terminate

Get metrics of internal states via API

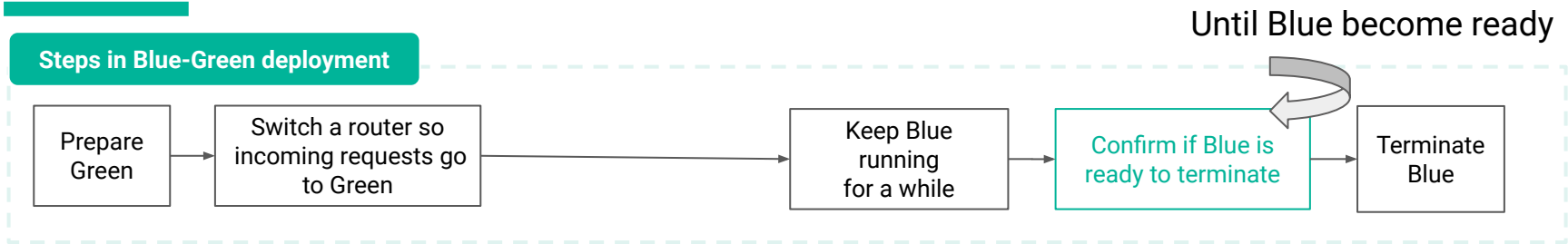
Determine if each server is ready to terminate



If the number of sessions is 0, it is ready

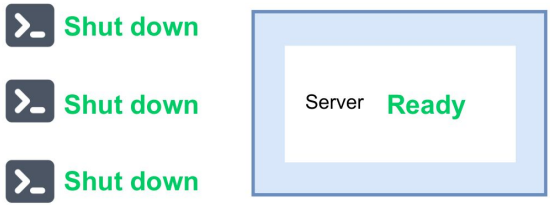
- Server-1: Not ready to terminate
- Server-2: Ready to terminate

Still not enough to address long-living states



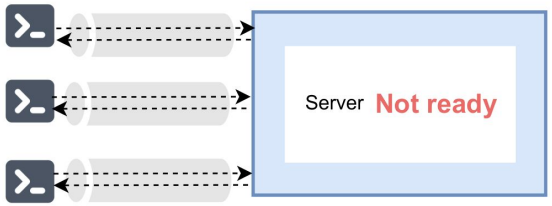
Happy path

Every session in a blue server shuts down when starting a deployment. Deployment finishes successfully.



Unhappy path

Every session in a blue server keeps running. We've waited for hours, but Blue is not ready to terminate yet…



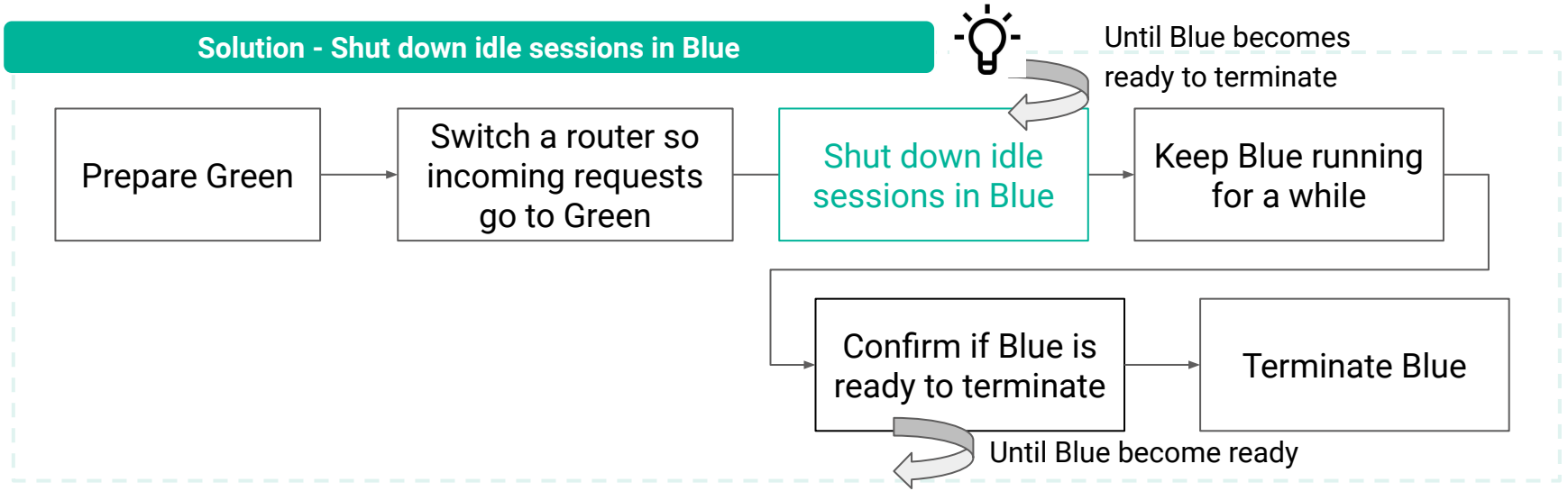
Any ways to address unhappy path?

Symptom

Long-living existing sessions makes terminations of Blue wait for a long time

Why?

Each session keeps living until a client disconnects a WebSocket connection



Shut down idle sessions in Blue - 1st step

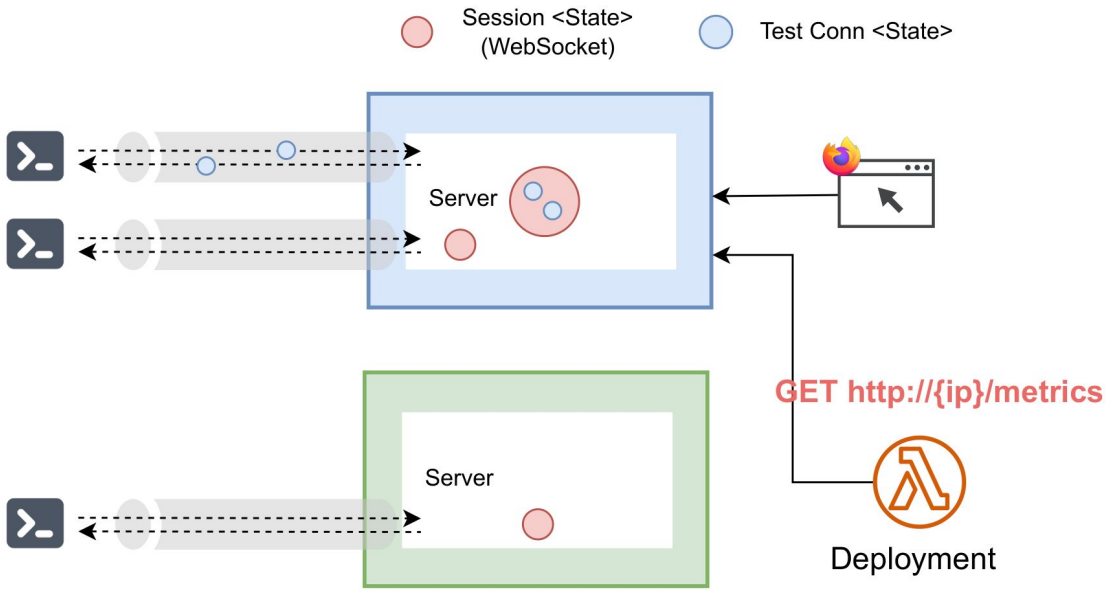
Get metrics of internal states via API

Determine if each session is idle

Shut down idle sessions

Clients reconnect to Green

Get detailed metrics of each session via API



Shut down idle sessions in Blue - 2nd step

Get metrics of internal states via API

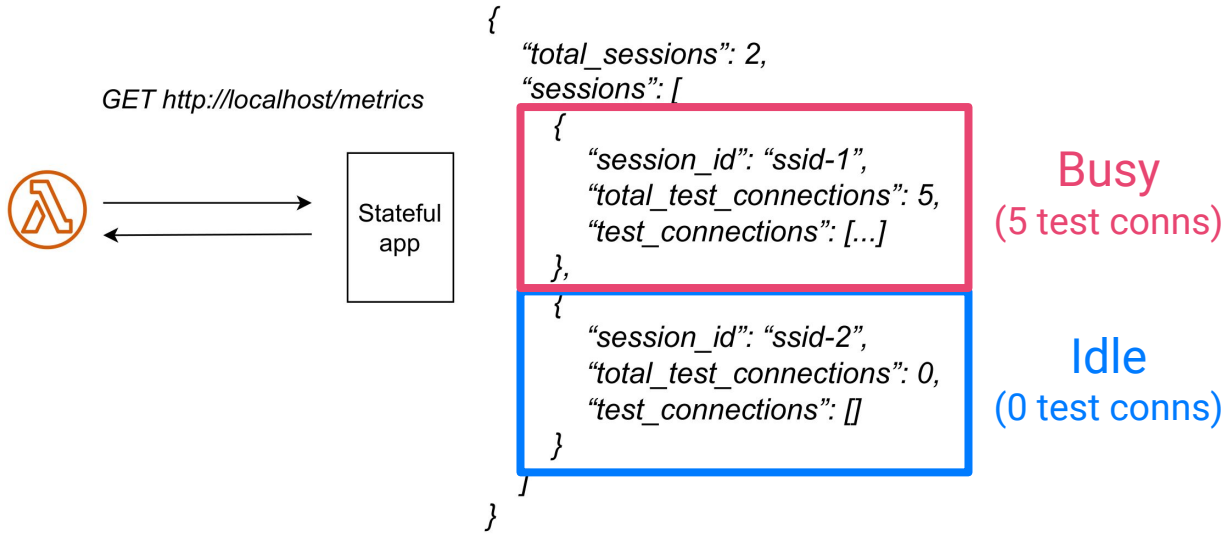
Determine if each session is idle

Shut down idle sessions

Clients reconnect to Green

Implement the business logic to determine if each session is idle

a. If no Test Connection, it is idle
b. Else, it is busy (must not be shut down)

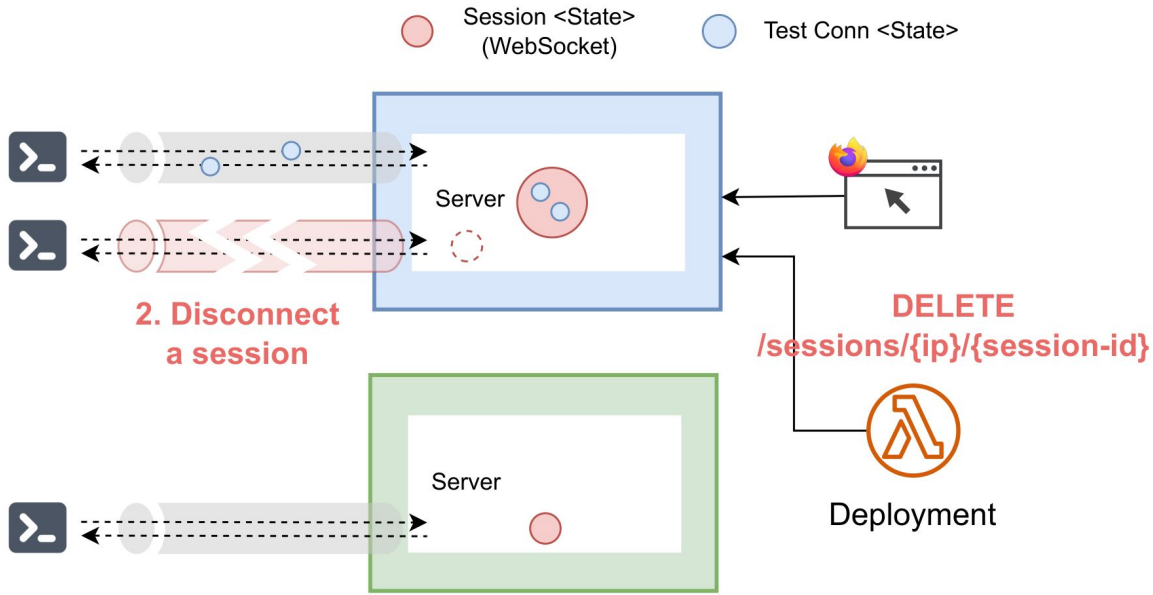


Shut down idle sessions in Blue - 3rd step

Shu down idle sessions by calling API to control each session

Implement HTTP endpoint to control internal states

- Deployment process sends requests “DELETE /sessions” to close sessions
- Server accept requests and closes idle sessions

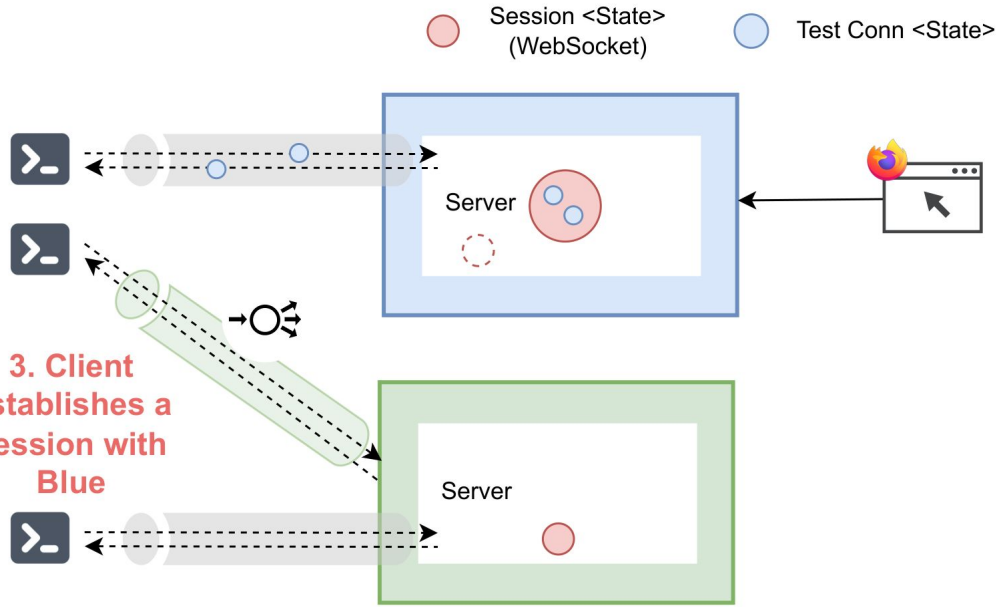


Shut down idle sessions in Blue - 4th step

Clients reconnect to Green following by the router navigation

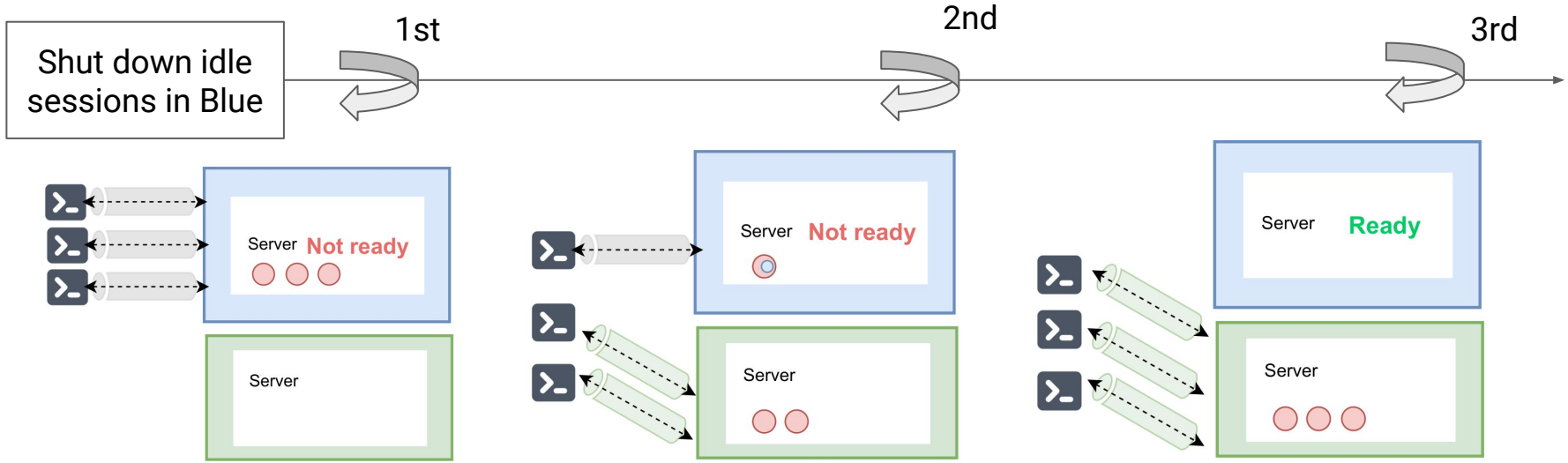
Client application should have an automatic reconnection

- In case a server is replaced



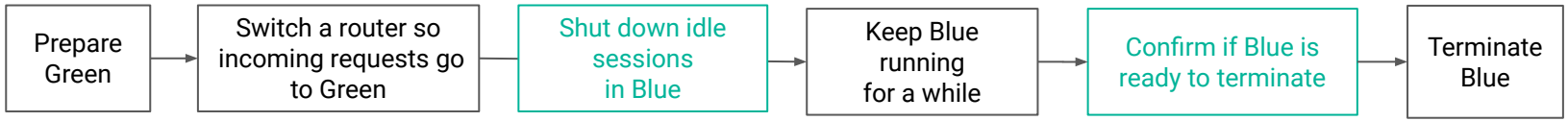
Succeeded to address unhappy path

- Existing clients can now be safely directed to the Green in sequence
  - It is easier to find the safe time to disconnect on a per-session basis than on a per-server basis
- Deployment time has been mitigated



Implementation with AWS

Steps in Blue-Green deployment

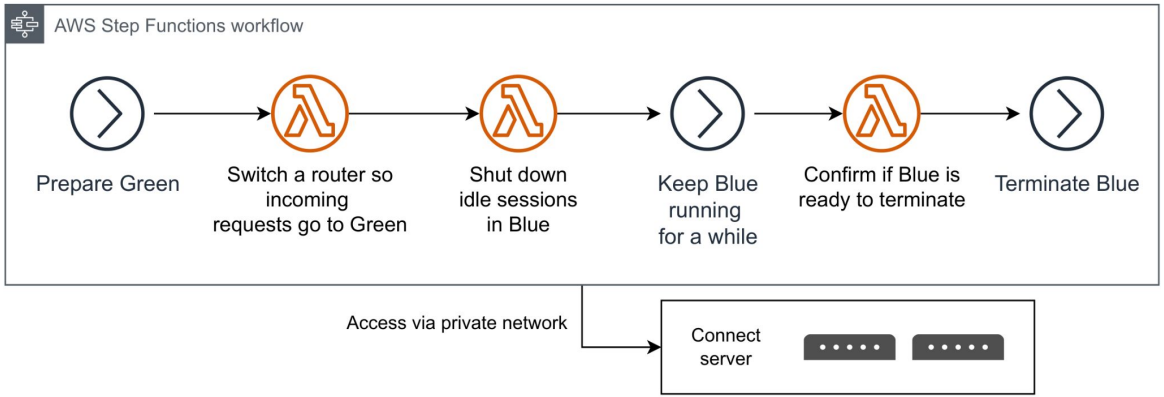


Develop in-house deployment pipeline

- No managed AWS service was a fit

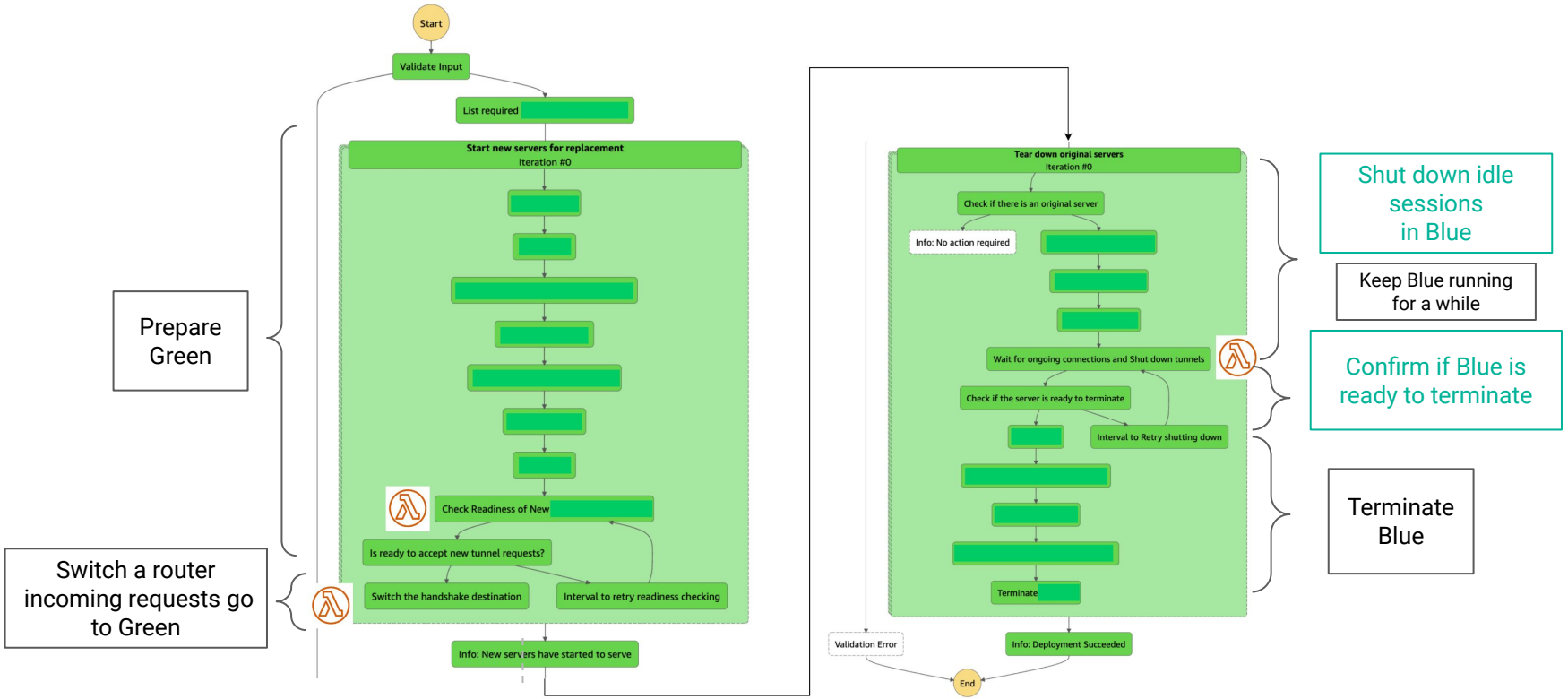
AWS Step Functions*1 **and Lambda**

- +1 Serverless
- +1 Fully customizable
- -1 Takes time to build



*1 AWS Step Functions is a visual workflow service helping developers to build automated processes

Deployment workflow with AWS Step Functions



Lessons we learned

- Long-living states make it difficult to terminate an old server safely
- Blue-Green deployment with two steps controlling internal states
  - 1. Shut down idle states (Sessions; WebSocket conns)
  - 2. Verify Blue servers are ready to terminate before terminating them
- Design a stateful app to be observable and be able to control its internal states since an earlier stage
- Pros/Cons
  - +1 Zero-downtime, no errors during deployment
  - +1 Mitigate deployment time by shutting down idle states gradually
  - -1 Time to build a deployment pipeline

Design of a stateful system for...

Observability

A measure of how well you can understand and explain any situation your system can get into, no matter how novel or bizarre.

A difficult question to answer in stateful systems (1)

Difficult questions

How can we see the actual usage of each stateful server?

- Meaningful data is inside of internal states
- *e.g., The number of test connections each server addresses in its internal state*

Cause

Low Visibility

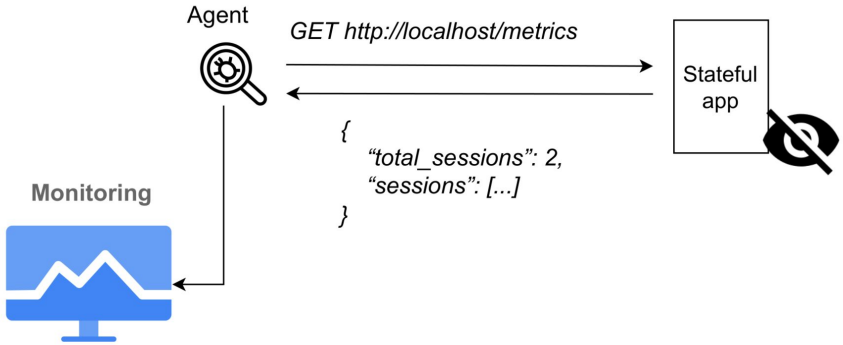
- Inside of app is invisible from external
- Not persistent

Solution

Custom metrics visualizing internal states

Custom metrics to overview inside of a stateful app

- Get metrics data via API to show metrics of internal states
- Record as a custom metrics
  - e.g., DataDog custom metrics from datadog-agent
- Visualize meaningful data inside of an app
  - E.g., number of test connections -> how busy each server is
- Set alerts
  - Define what makes a metric health versus unhealthy



A difficult question to answer in stateful systems (2)

Difficult questions

Can you search all events and user requests related to a suspicious state?

- Logic to handle internal states would be complex and core part of app
- *e.g., Session "A" seems to be disconnected at unintentional timing. Is it ok?*

Cause

Difficult to search from each state

- No searchable key by default

Solution

Searchable logs and traces by each state identifier

Searchable logs by each state level

Techniques

- Structured logging
- Add an identifier of state to the key of log entries
  - e.g., *session_id*, *test_conn_id*

Effect

- Likely to be uniquely identifiable
  - e.g., Session_id > test_conn_id

```
| jq 'map(select( .session_id == "ssid-1" ))'

| jq 'map(select( .test_conn_id == "id-1" ))'
```

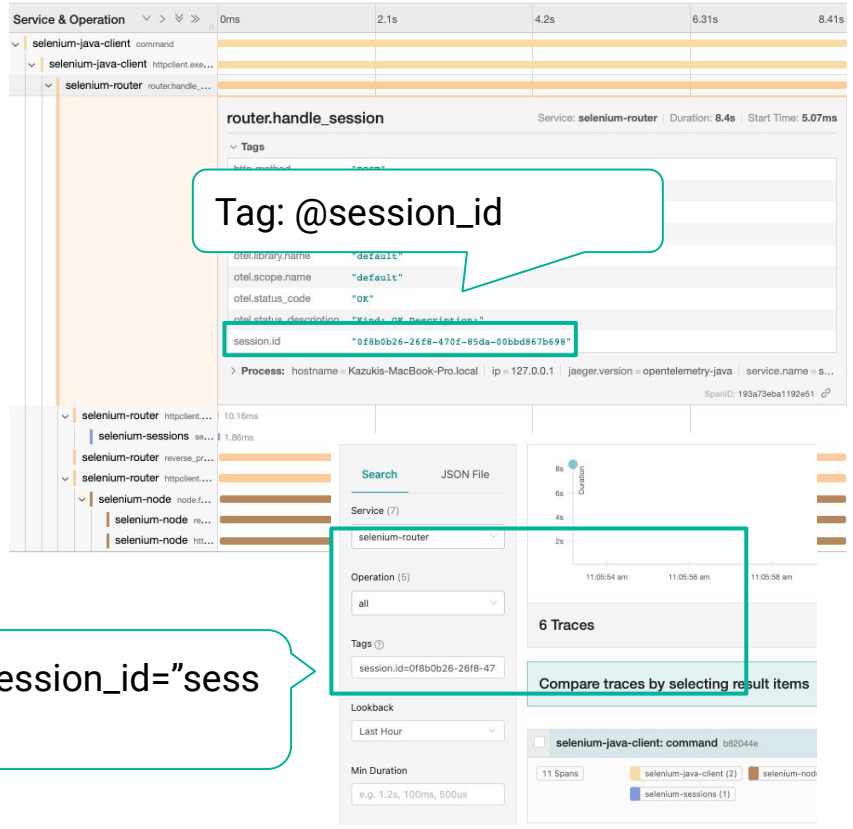
{“ts”: “...”, "level": "info", "session_id": "ssid-1", "msg": "..."}
{“ts”: “...”, "level": "info", "session_id": "ssid-2", "msg": "..."}
{“ts”: “...”, "level": "warn", "session_id": "ssid-1", "msg": "..."}
{“ts”: “...”, "level": "info", "session_id": "ssid-1", "test_conn_id": "id-1", "msg": "..."}
{“ts”: “...”, "level": "info", "session_id": "ssid-1", "test_conn_id": "id-2", "msg": "..."}
{“ts”: “...”, "level": "info", "session_id": "ssid-2", "test_conn_id": "id-1", "msg": "..."}
{“ts”: “...”, "level": "info", "session_id": "ssid-1", "msg":"..."}

Traces - OSS case study - Selenium Grid 4

- Selenium Grid 4
  - A server that makes it easy to run tests in parallel on multiple machines
  - Instrumented with tracing using OpenTelemetry

Techniques

- Add a state identifier into span attributes *1
- Traces can be searched by “*session_id*”
  - (WebDriver) Session id is a unique identifier to handle a browser for testing



*1 Key-value pairs providing additional information about each span

Autify

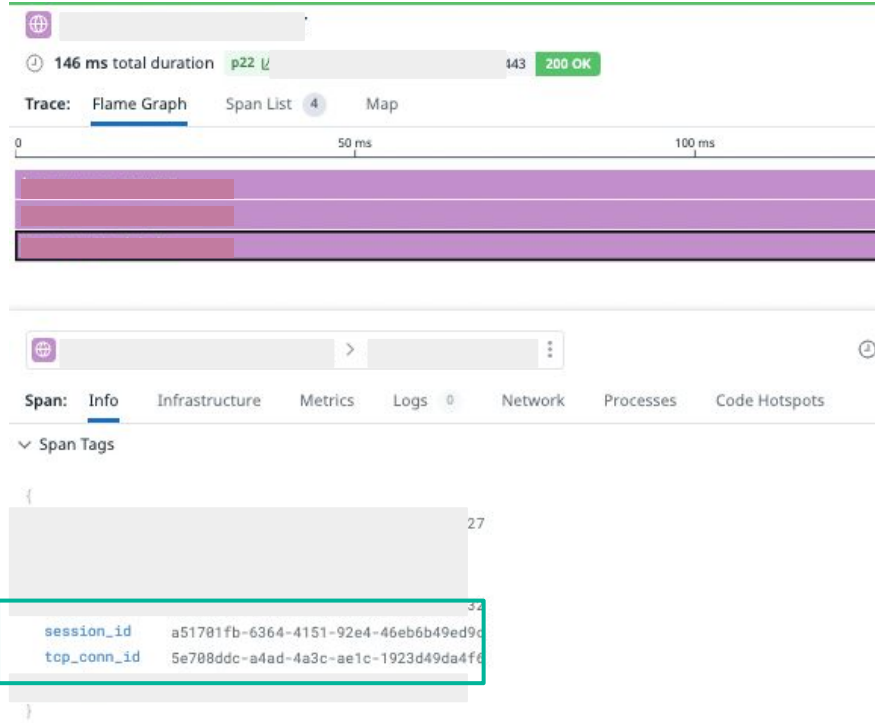
Traces - Custom Instrumentation to search for traces

Provide context into spans by adding an identifier of states into span tags

Traces have contexts pointing internal states

@tags.session_id=”session_id”

@tags.test_conn_id=”test_conn_id”



Lessons we learned

- Building a robust deployment makes us to consider observability of stateful systems
  - Solutions for building a robust deployment can be used to build metrics data
- Improve metrics, logs, and traces in case something unknown happens around the core stateful part

Questions? >>>

Kazuki Higashiguchi

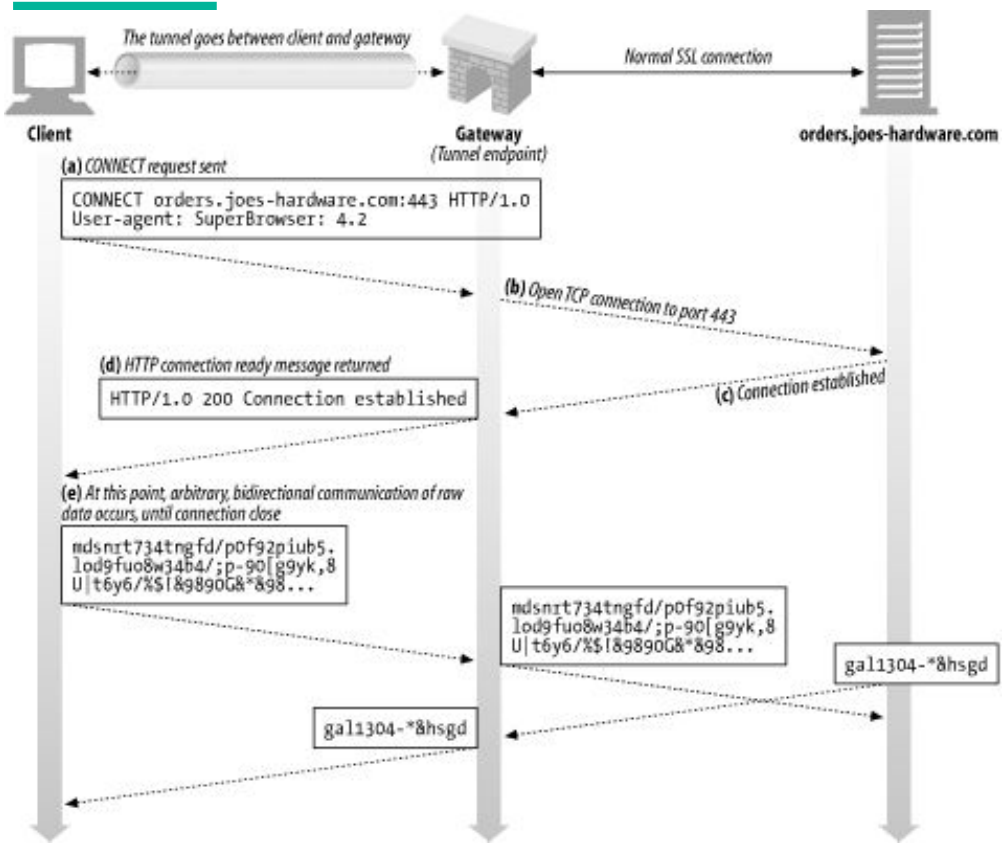
@hgsgkt

Appendix A

HTTP Tunneling over WebSocket

https://speakerdeck.com/hgsgtk/http-tunneling-in-go

HTTP Tunnel with CONNECT method



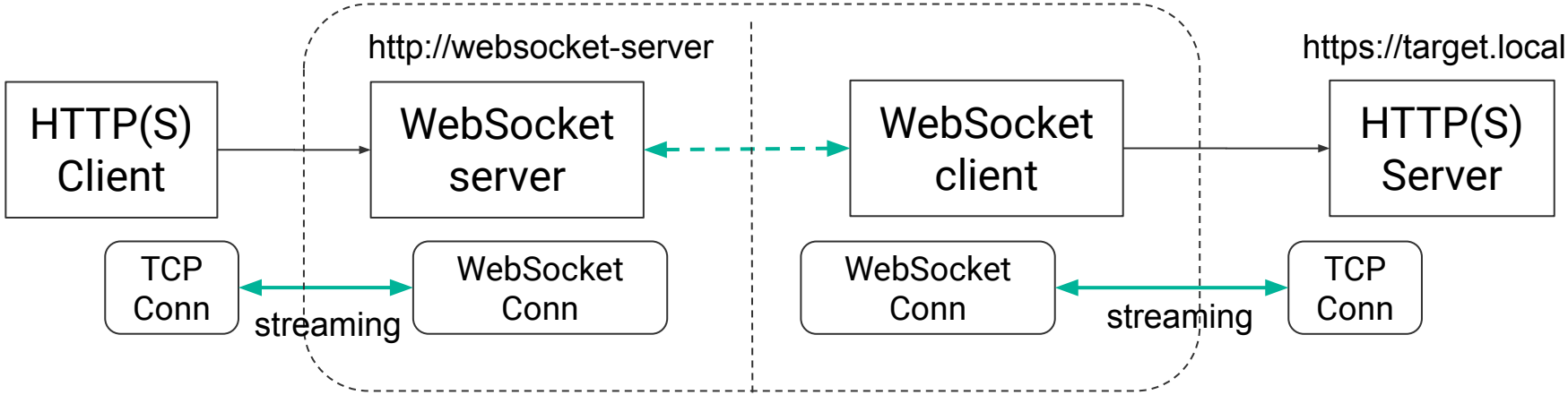
1. Client sends CONNECT request
2. Gateway opens TCP connection to the server
3. Gateway returns HTTP ready message to the client
4. Start bidirectional communication of raw packets of data

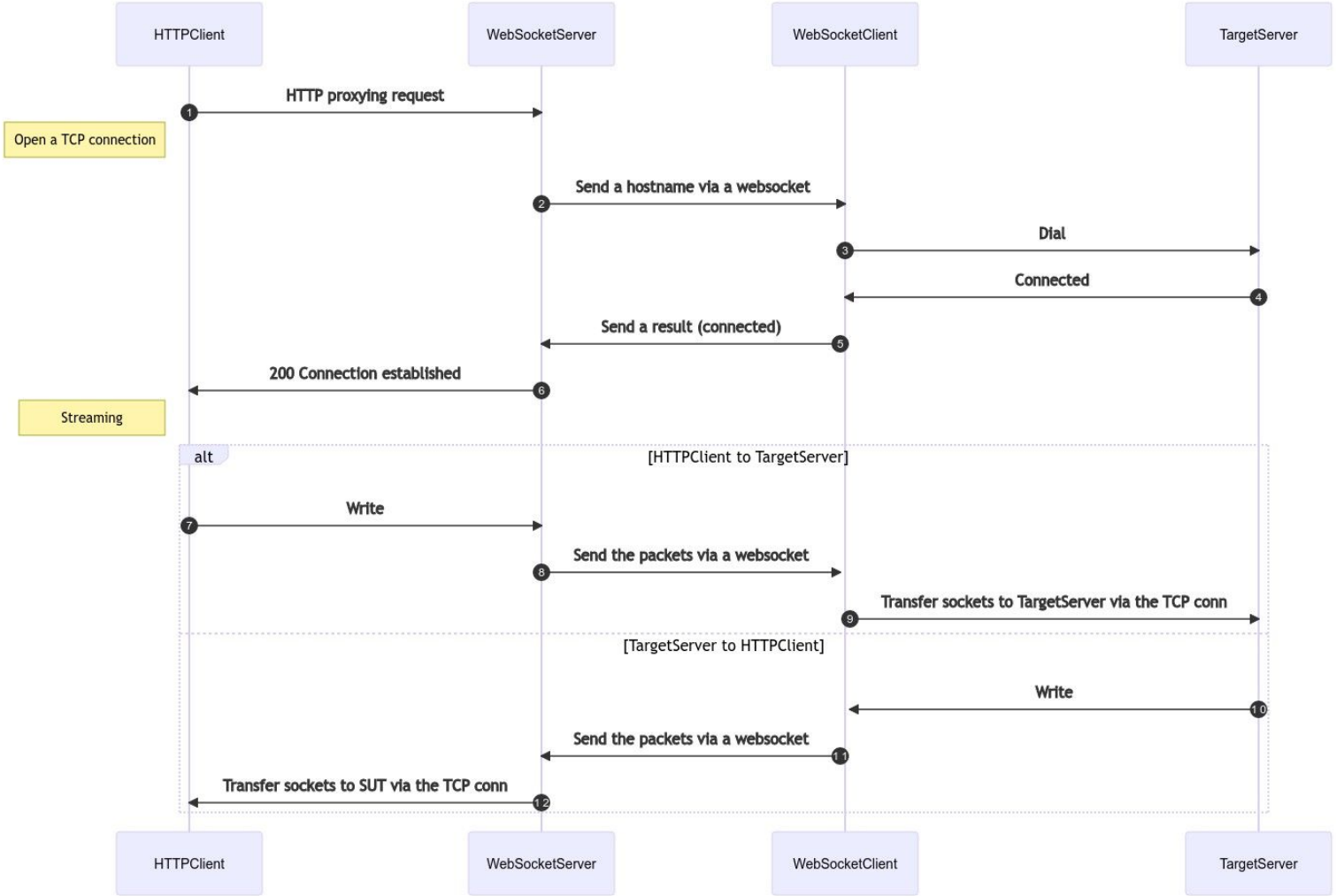
Quoted from [HTTP: The Definitive Guide / 8.5 Tunnels]

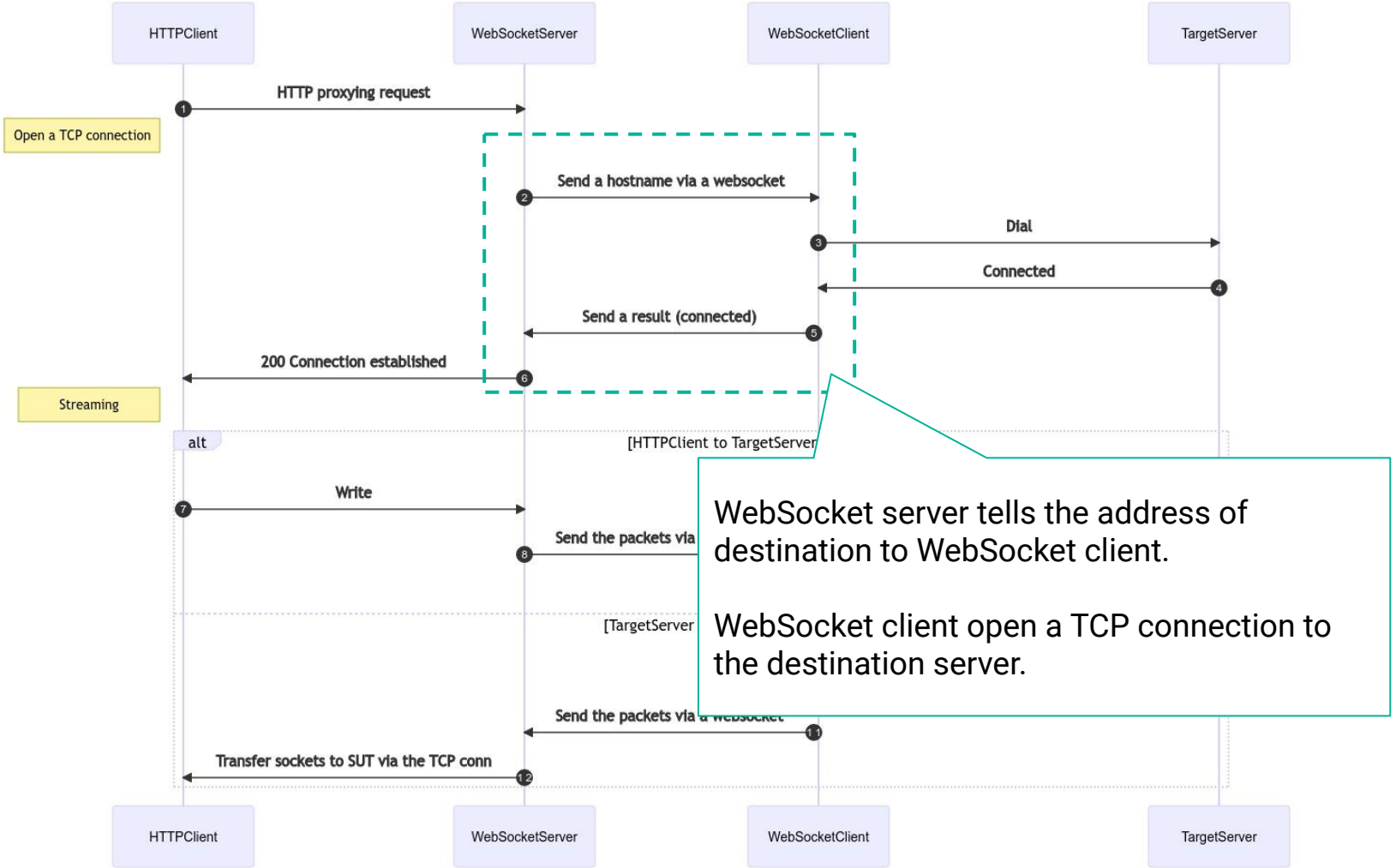
HTTP Tunneling over WebSocket

WebSocket server and WebSocket client jointly act as a proxy(gateway).

$ curl -Lv -x http://websocket-server https://target.local

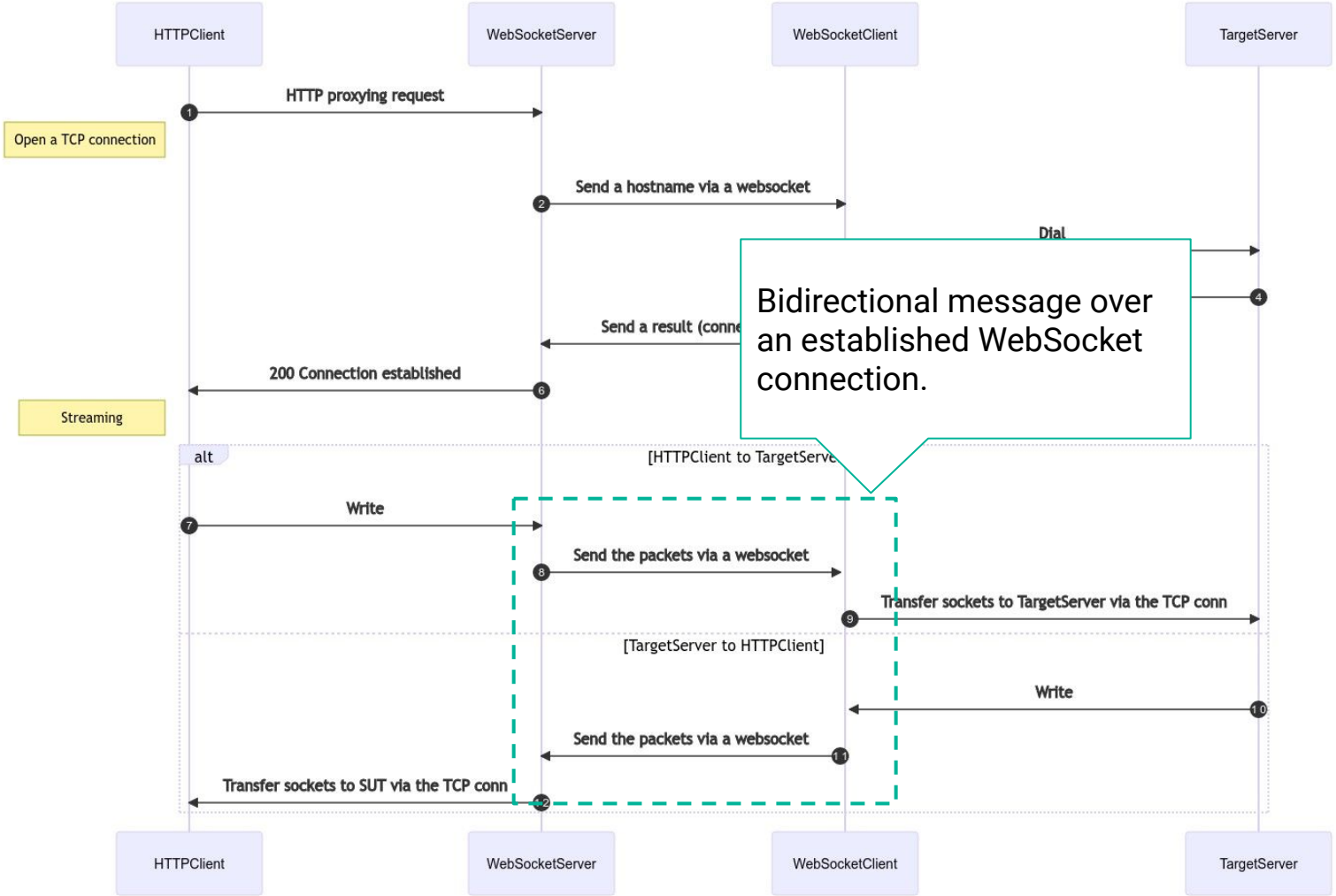



HTTPClient
WebSocketServer
WebSocketClient
TargetServer
HTTP proxying request
1
Open a TCP connection
Send a hostname via a websocket
2
Dial
3
Connected
4
Send a result (connected)
5
200 Connection established
6
Streaming
alt
[HTTPClient to TargetServer]
Write
7
Send the packets via a websocket
8
Transfer sockets to TargetServer via the TCP conn
9
[TargetServer to HTTPClient]
Write
10
Send the packets via a websocket
11
Transfer sockets to SUT via the TCP conn
12
HTTPClient
WebSocketServer
WebSocketClient
TargetServer



WebSocket server tells the address of destination to WebSocket client.

WebSocket client open a TCP connection to the destination server.

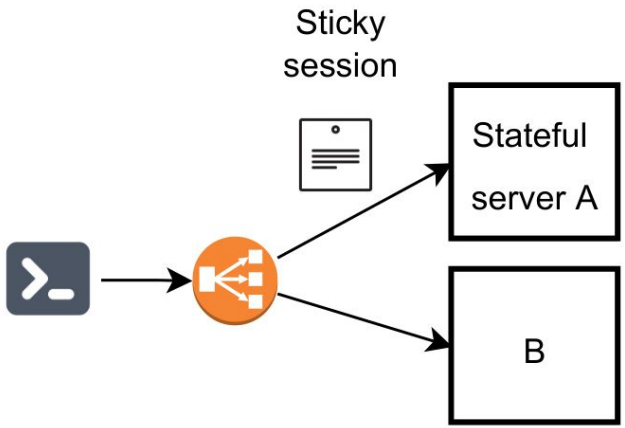

HTTPClient
WebSocketServer
WebSocketClient
TargetServer
HTTP proxying request
1
Open a TCP connection
Send a hostname via a websocket
2
Dial
4
Send a result (conne
200 Connection established
6
Streaming
Bidirectional message over an established WebSocket connection.
alt
[HTTPClient to TargetServe
Write
7
Send the packets via a websocket
8
Transfer sockets to TargetServer via the TCP conn
9
[TargetServer to HTTPClient]
Write
10
Send the packets via a websocket
11
Transfer sockets to SUT via the TCP conn
12
HTTPClient
WebSocketServer
WebSocketClient
TargetServer

Appendix B

Infrastructure patterns of routing stateful servers

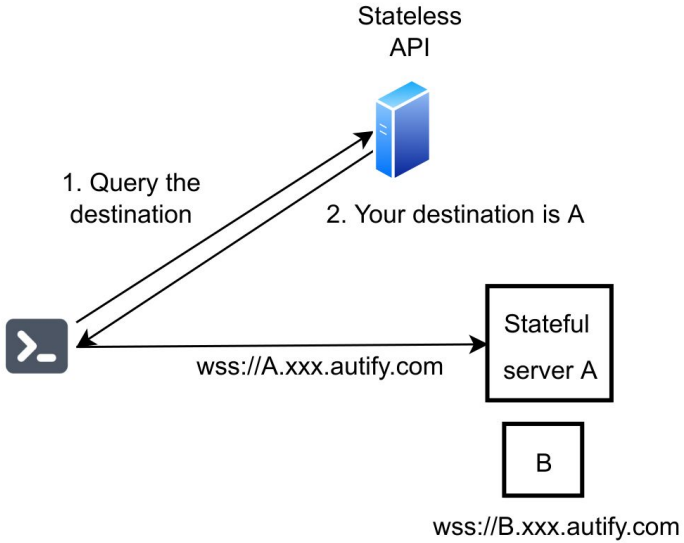
Infrastructure patterns of routing stateful servers

1



Load balancing with Sticky session
+1 Simple
+1 Little or no implementation required

2



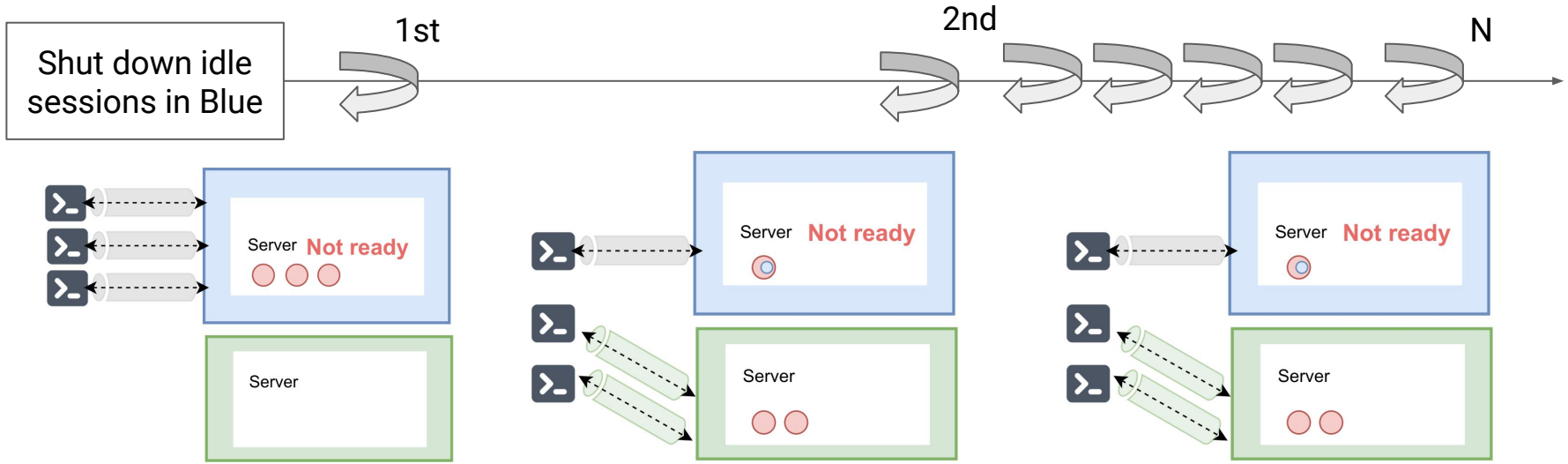
Another API tell clients the addr
+1 Address flexible server requirements (e.g., enterprise customer)
+1 Each server has its own URL, making reconnection easy

Appendix C

If one session is busy in more long term...

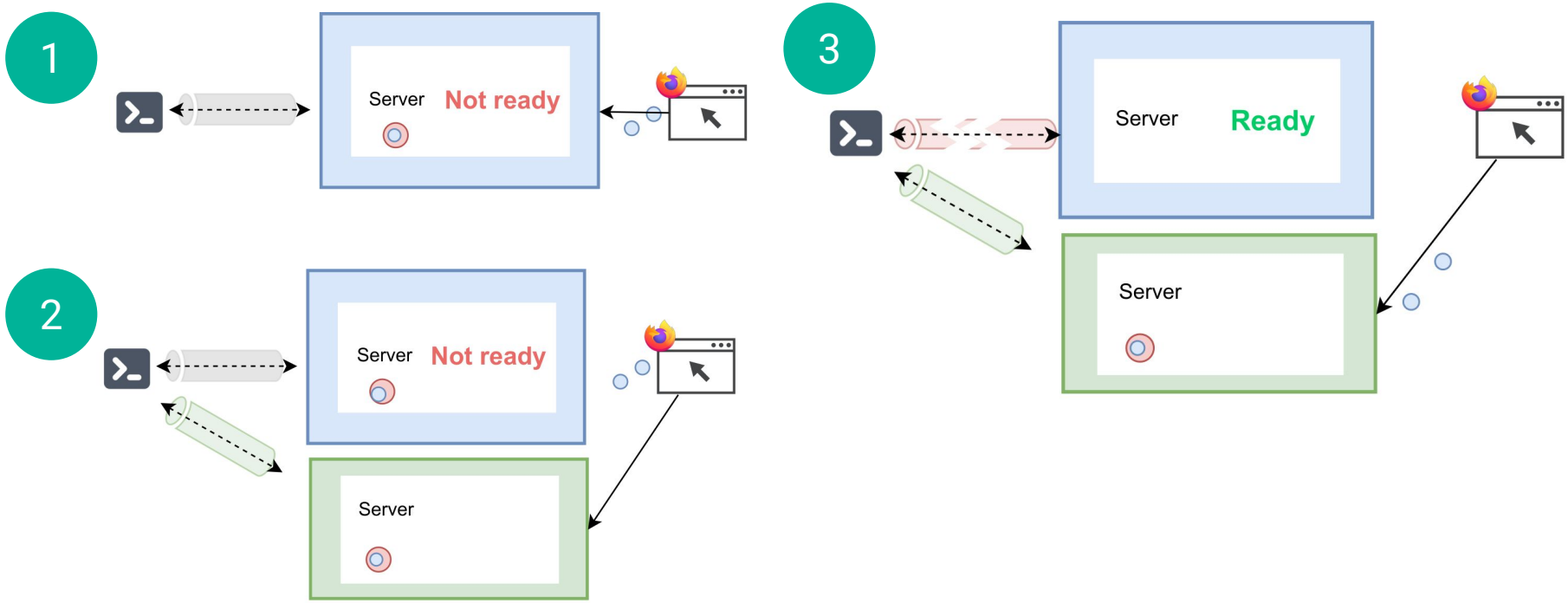
Unhappy path again if one session is busy more long term…

Theoretically, if a single session is used forever and uninterruptedly, the solution “Shut down idle sessions in Blue” will not be sufficient.



A possible design

Make the network route redundant by having two sessions at the same time



A possible design: Pros/Cons

Pros

- Faster deployment even when some sessions are busy for a long time

Cons

- More complex design and error handling
  - Lots of edge cases (e.g., network interruption)

Our current decision: Go with the original design

- Browser automation does not always make network requests (e.g., scrolling, find an element)
- Enough easy to find the safe time to disconnect on a per-session
- Even if this becomes necessary, the original design will still need to be

Appendix D

Infrastructure patterns between browsers and proxies

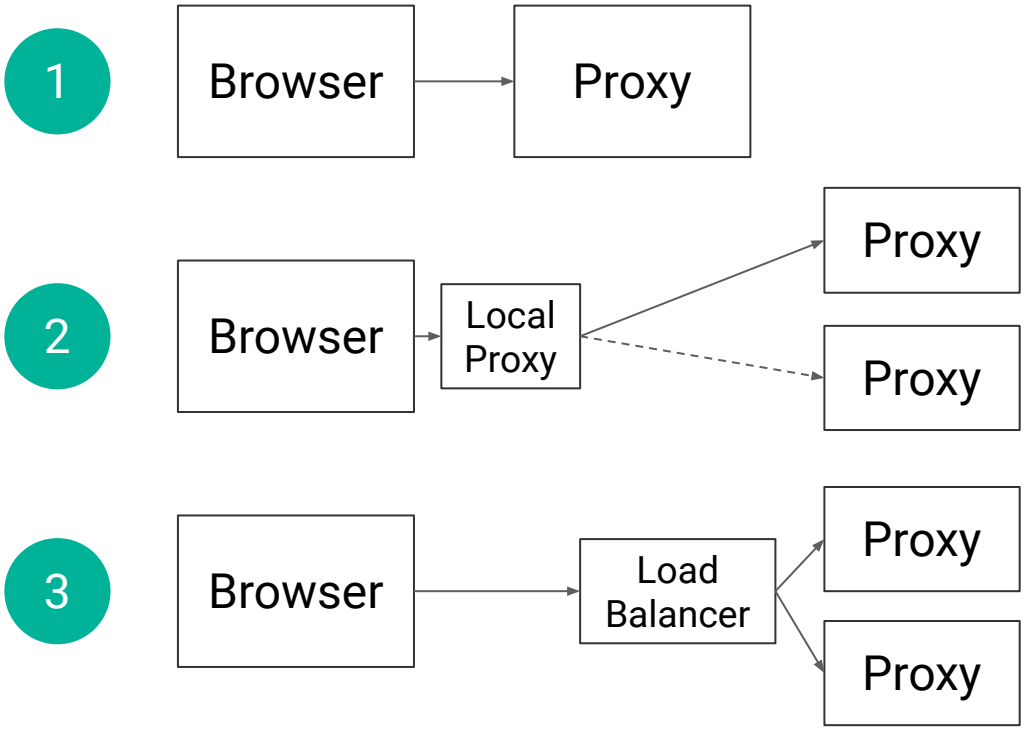
Modern browsers’ specification

- Proxy address is specified at startup
  - Changing addresses after startup is tricky.

e.g., Chrome

./Google\ Chrome --no-sandbox --proxy-server={proxy-host}:{proxy-port}

Infrastructure design patterns



Pros/Cons

+1 Straightforward
-1 No way to switch a proxy server after launching a browser

+1 Can switch a target proxy by a local proxy behavior
-1 Communicating address changes to the Local proxy could be complicated

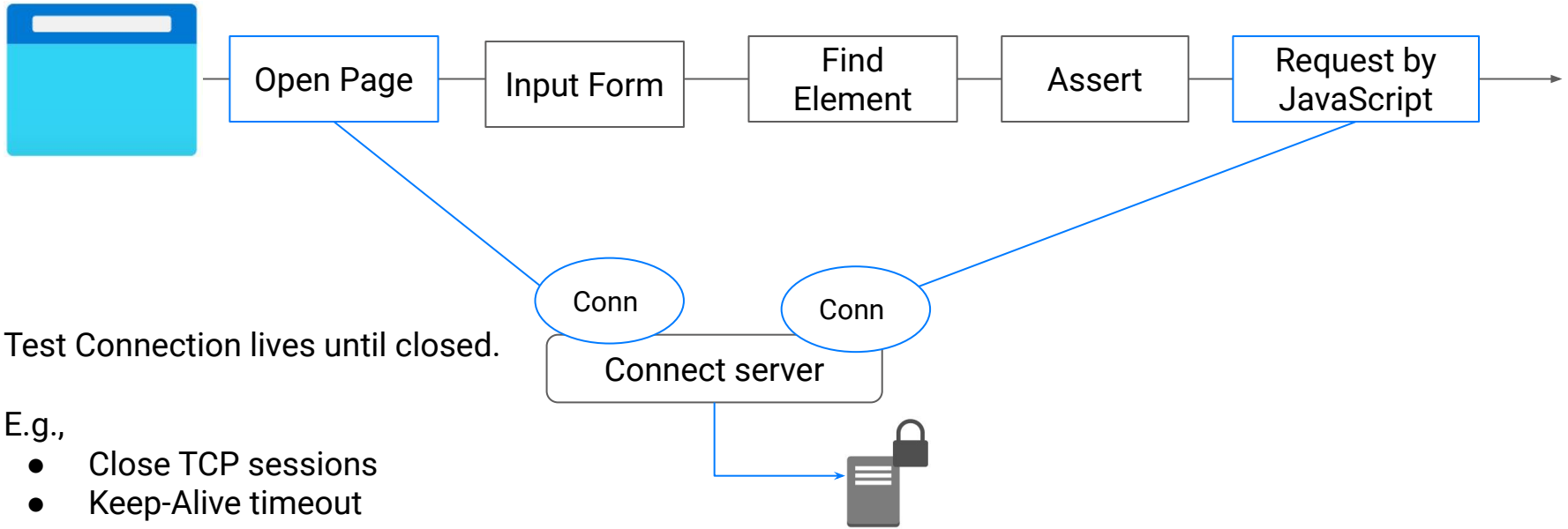
+1 Can switch a target proxy by load balancing
-1 It depends on the URL of the load balancer itself.

Appendix X

Supplementary materials

Network request during browser operation - Simple

A test Scenario with a simple web application



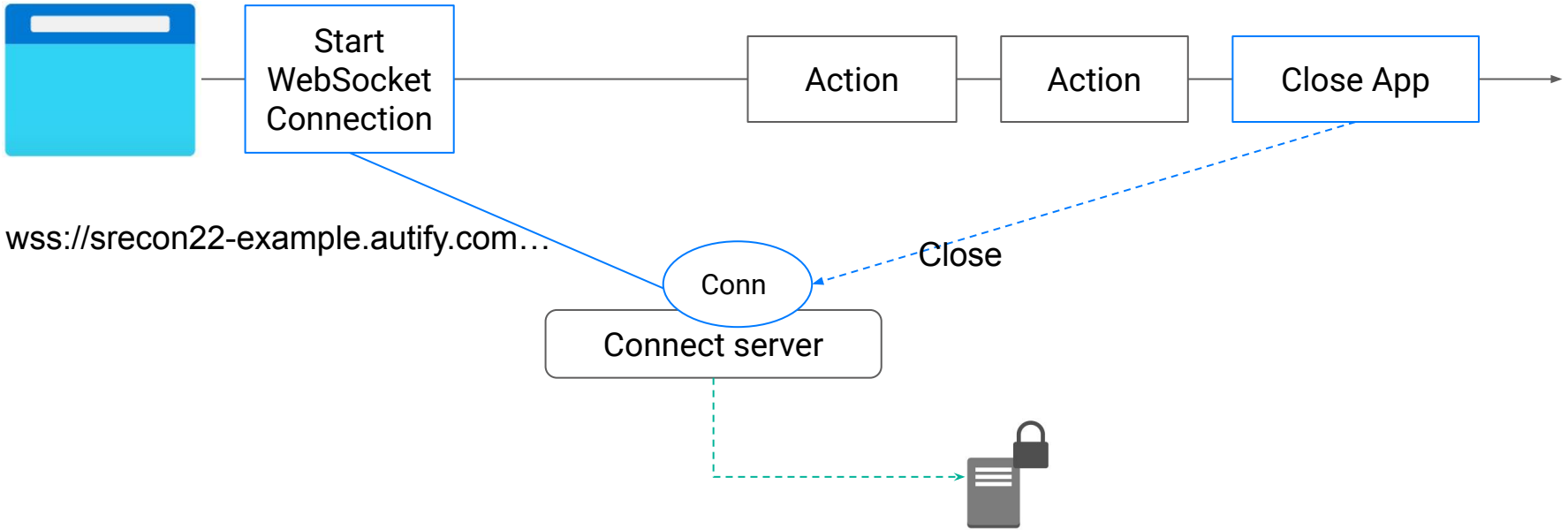
Test Connection lives until closed.

E.g.,

- Close TCP sessions
- Keep-Alive timeout

Network request during browser operation - WebSocket

A frontend application using WebSocket



References

- HTTP Tunneling in Go by Kazuki Higashiguchi, Autify
- Book “Observability Engineering” by Charity Majors, Liz Fong-Jones, George Miranda
- Book “Operations Anti-Patterns, DevOps Solutions” by Jeffery Smith
- Book “Practical Monitoring” By Mike Julian
- Observability Whitepaper by CNCF TAG for Observability
- Observability in Selenium Grid by Selenium
- Blue Green Deployment by martinfowler.com