
HELM
TiKV
KubeCon
CloudNativeCon
Europe 2023

KubeCon | CloudNativeCon

Europe 2023

Breakpoints in Your Pod

Interactively Debugging Kubernetes Applications

Daniel Lipovetsky, D2IQ

v1.2

About me

Photo: H. Lipovetsky

Daniel Lipovetsky

- Working on Kubernetes *cluster lifecycle* since 2016
- Contributor to Cluster API
- Staff Software Engineer @ D2IQ

@dlipovetsky on Kubernetes Slack and GitHub

About this presentation

- Introduction
- Debug Session (Demo)
- Challenges & Solutions
- Future Work
- Call to Action

Introduction

- Goal: Remote debug session, setting breakpoints in multiple Pods
- Prior Art: Squash
- Related: Telepresence
- Technical details
  - Cluster API
  - Ephemeral containers

Cluster API

A lot to learn by setting breakpoints

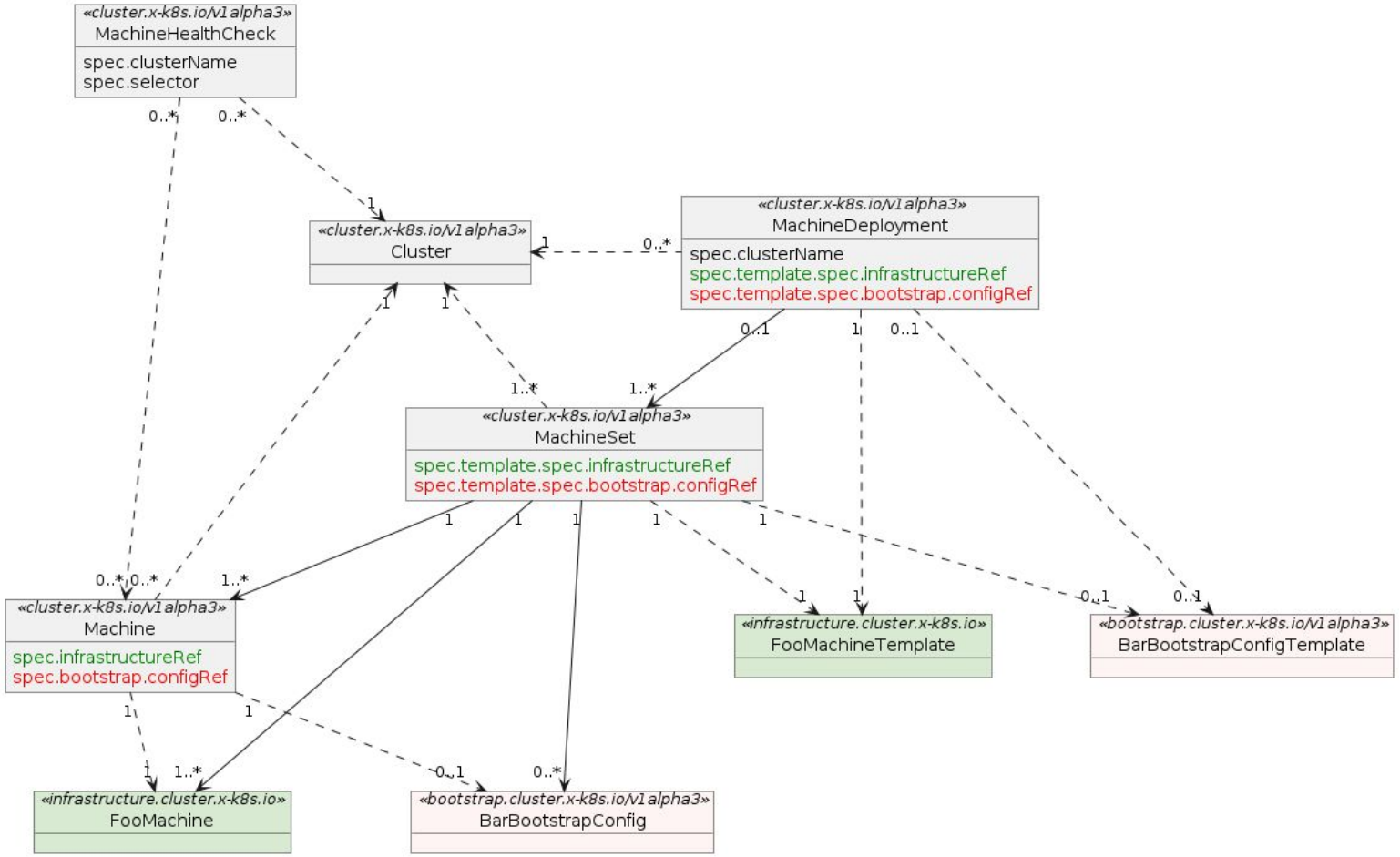
- Complex system that manages Kubernetes clusters
- 4+ Pods, 14+ Controllers

A representative example

- Has most of the challenges you can expect when you debug a Kubernetes application
- Used in production all over the world

Cluster API

Relationships between a few Cluster API Custom Resources



Source: https://cluster-api.sigs.k8s.io/developer/crd-relationships.html

Ephemeral containers

- Containers you can create in a running Pod
- Designed for investigation and debugging
  - Can use their own Security Context
  - Can share process namespace of another container in the Pod
- Watch the KubeCon EU 2022 talk by Alan Alpar, *Seeing is Believing*

Debug Session (Demo)

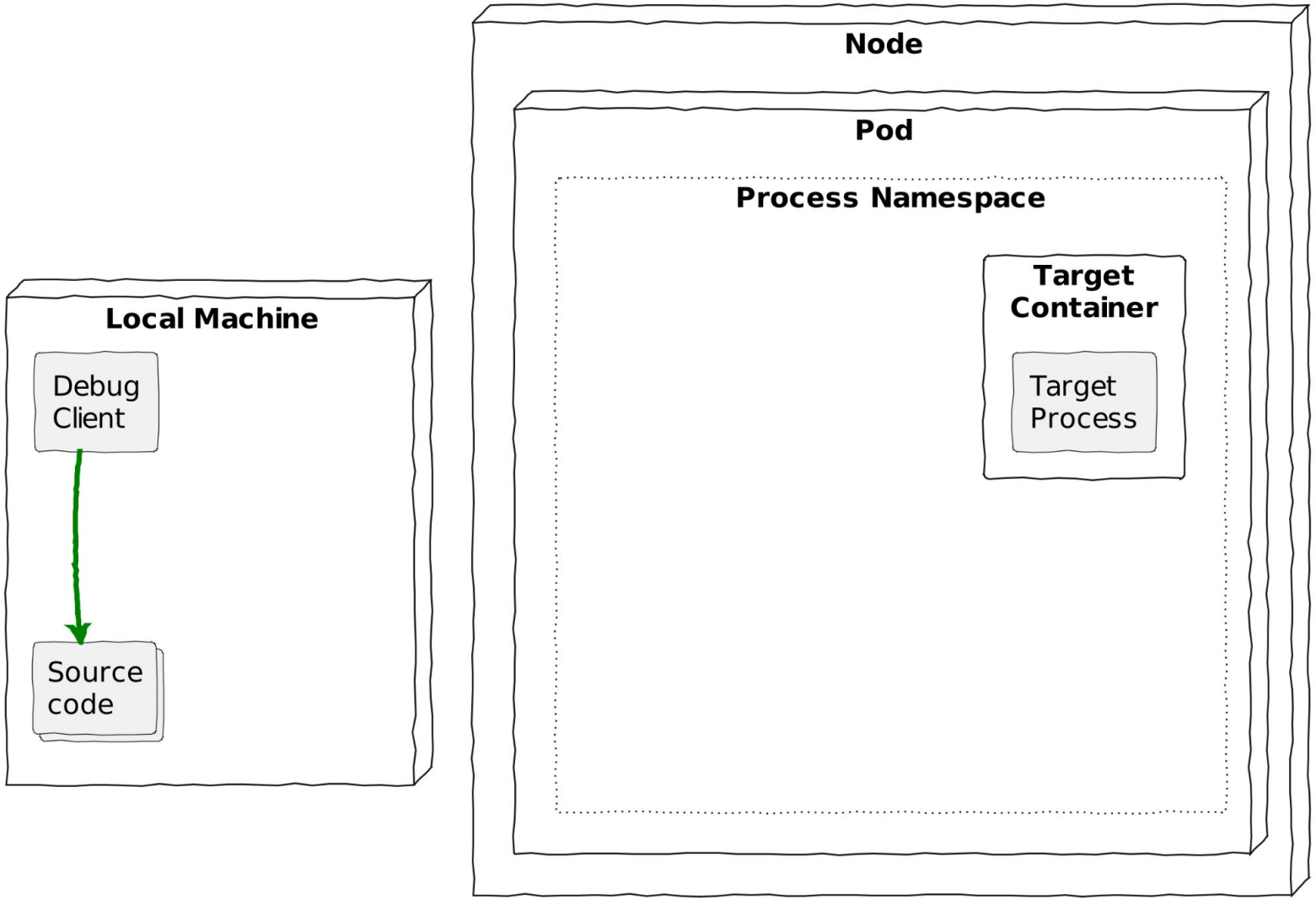
- One VS Code session
- Debugging processes in 3 Pods
- Breakpoints help us follow along as Cluster API creates a new Machine

Reproduce this demo yourself. Learn more at
https://github.com/dlipovetsky/kubecon-eu-2023-demo

Challenges & Solutions

1. No debugger in the Pod
2. Debugger cannot attach
3. No debug info
4. Cannot reach debugger server
5. Source code not found
6. Other, less common challenges

Starting point



Challenge 1: No debugger in the Pod

- Debugger must run in the process namespace of the target process
- There is no debugger executable in the Pod

Challenge 1: No debugger in the Pod

Solution

1. Build an image with our debugger
2. Create an ephemeral container in the Pod

Challenge 1: No debugger in the Pod

Solution

1. Build an image with our debugger

```
FROM golang:1.20.3 AS build-dlv
RUN CGO_ENABLED=0 \
    go install
github.com/go-delve/delve/cmd/dlv@v1.20.2
FROM alpine:3.17.2
COPY --from=build-dlv /go/bin/dlv /usr/local/bin/dlv
RUN apk add binutils elfutils
```

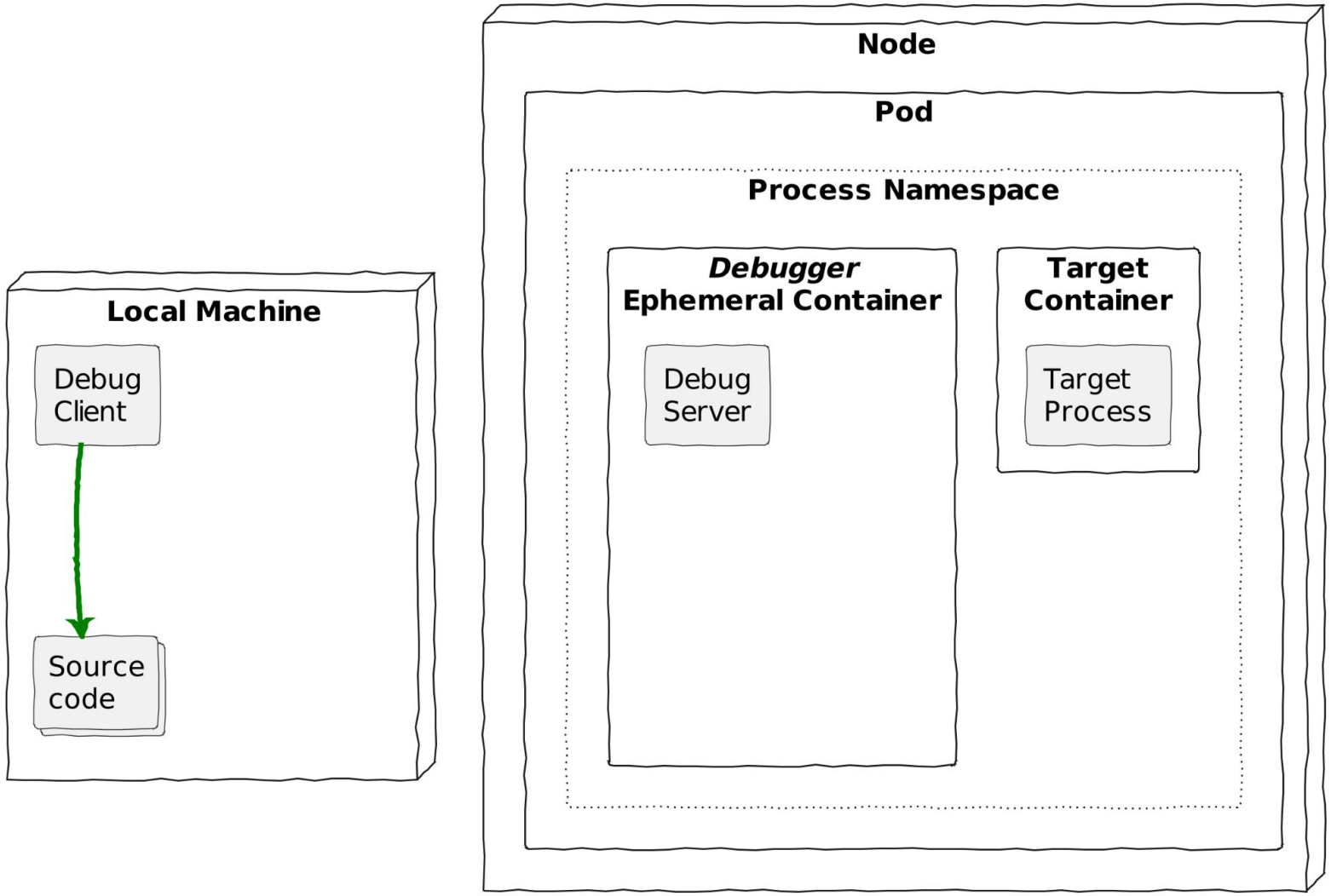
Challenge 1: No debugger in the Pod

Solution

2. Create an ephemeral container using the debugger image to the Pod

```
> kubectl debug example-pod \
    --image=docker.io/dlipovetsky/debugger:v1.0 \
    --container=debugger \
    --target=example-container \
    --share-processes=true \
    -- sleep infinity
```

Challenge 1: No debugger in the Pod



Challenge 2: Debugger cannot attach


```
could not attach to pid 1: operation not permitted
```

- The Pod defines a restrictive Security Context
- The debugger container needs
  - the SYS_PTRACE capability to attach to the target process and read the executable
  - to run as root, or with the user ID of the target process

Challenge 2: Debugger cannot attach

Solution

1. Give the debugger SYS_PTRACE capability
2. Run debugger as root (or as the user of the target process)

Challenge 2: Debugger cannot attach

Solution

1. Give the debugger SYS_PTRACE capability

```
securityContext:
  capabilities:
    add:
    - SYS_PTRACE
```

Challenge 2: Debugger cannot attach

Solution

2. Run debugger as root (alternatively, as the user/group of the target process)

```
securityContext:
  runAsNonRoot: false
  runAsUser: 0
  runAsGroup: 0
```

Challenge 2: Debugger cannot attach

☝️ If you continue to see the same error, check if the Yama Linux Kernel Module is denying the ptrace system call. *You will need access to the Node.*

```
> cat /proc/sys/kernel/yama/ptrace_scope
1
```

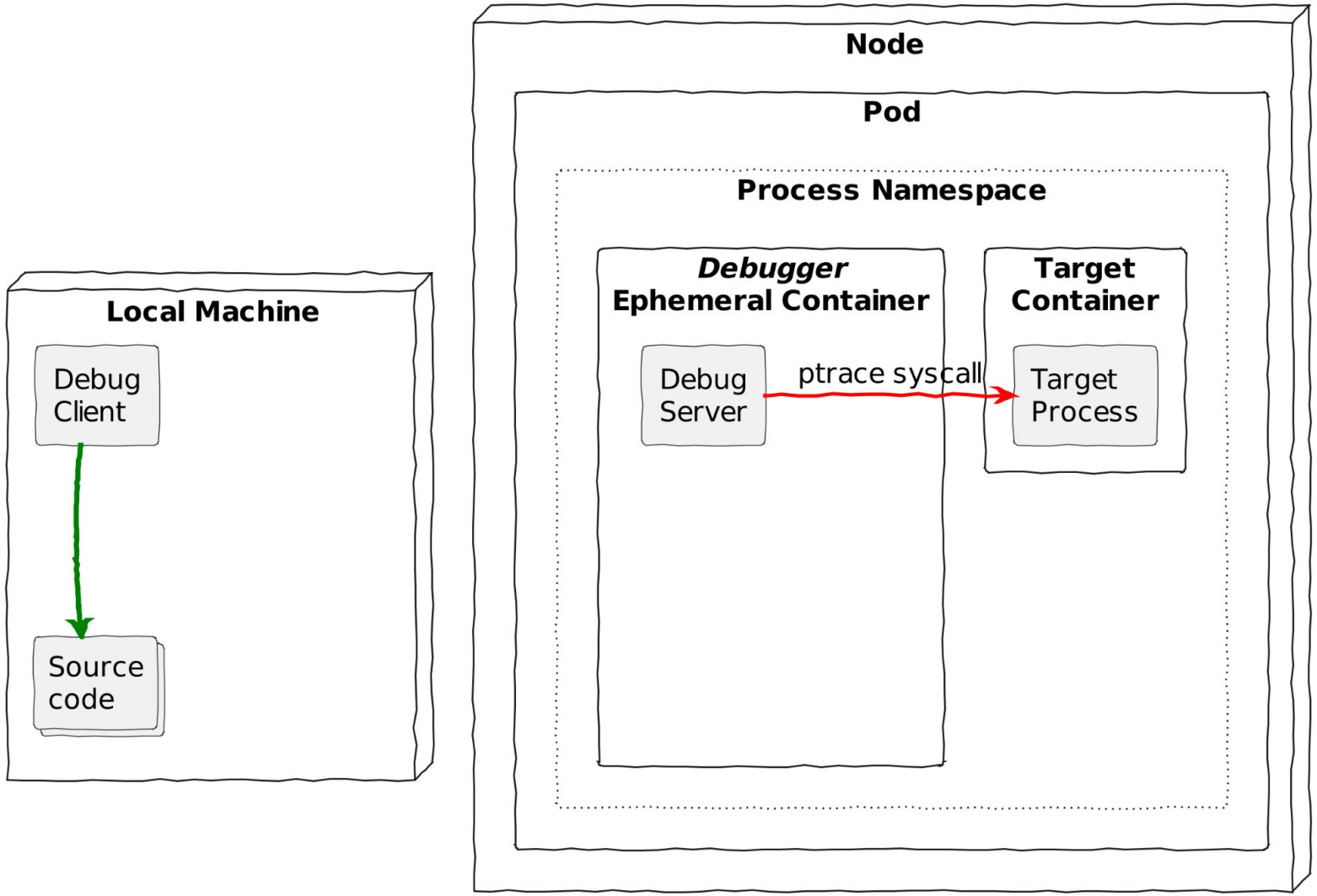
Configure Yama to allow the ptrace system call. *You will need privileged (superuser) access to the Node.*

```
> echo 0 | sudo tee
/proc/sys/kernel/yama/ptrace_scope
```

Challenge 2: Debugger cannot attach

- ☝️ You cannot (yet) set the security context with `kubectl debug`. You must patch the Pod's *ephemeralcontainers* subresource.
- ☝️ You cannot (yet) patch the *ephemeralcontainers* subresource with `kubectl patch`. You must either send an HTTP request, as described by Ivan Velichko, or patch `kubectl`.

Challenge 2: Debugger cannot attach



Challenge 3: No debug info


```
could not attach to pid 1: could not open debug info
```

- Debugger needs *debug info* to connect machine code to source code
- Some projects remove it to save space

Challenge 3: No debug info

Solution

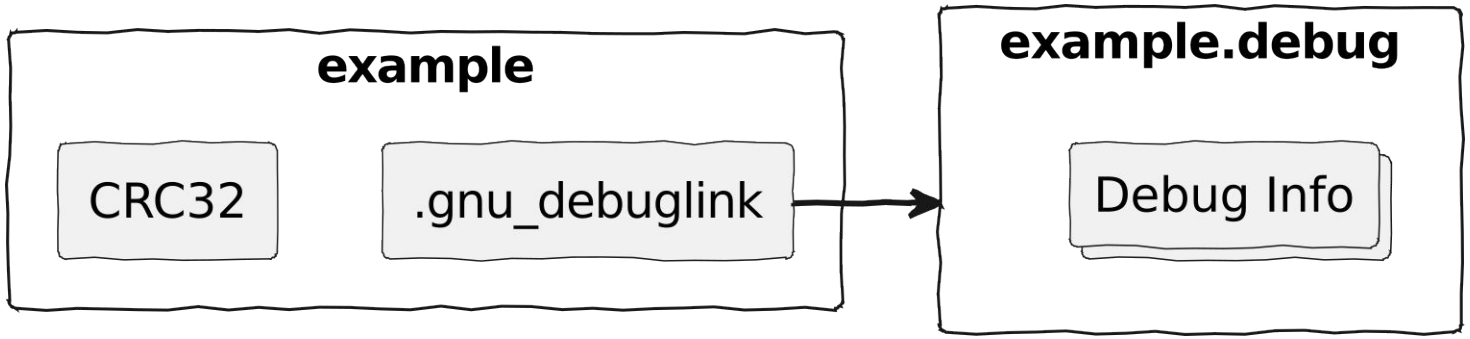
1. Get or create published debug info
2. Copy debug info to ephemeral container
3. Link executable to debug info

Challenge 3: No debug info

Solution

1. Get or create published debug info

```
> eu-strip example -f example.debug
```



Challenge 3: No debug info

Solution

2. Copy debug info to ephemeral container

```
> kubectl cp \
    --container=debugger \
    example.debug \
    example-pod:/
```

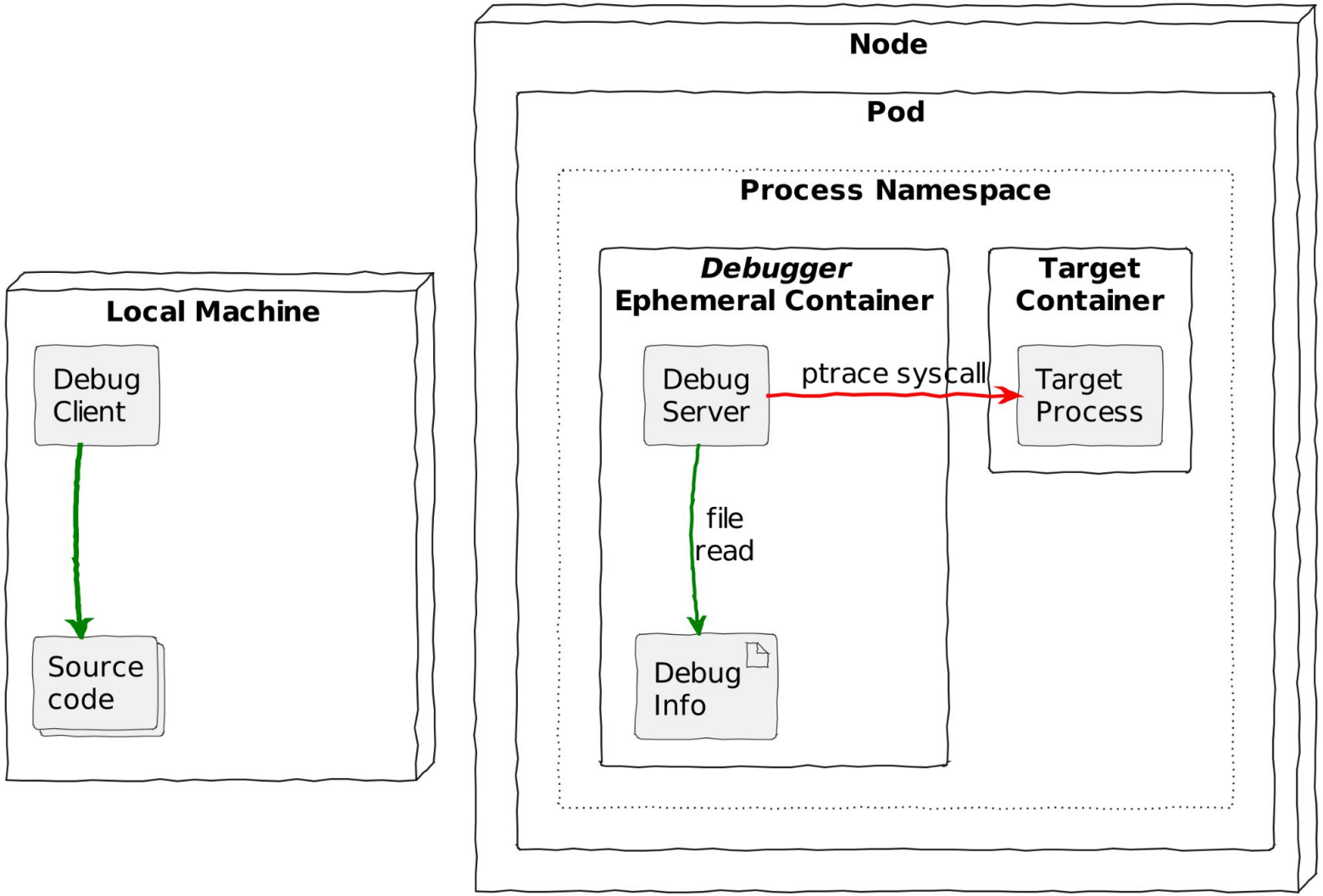
Challenge 3: No debug info

Solution

3. Link executable to debug info

```
> objcopy /proc/1/root/example \
    --add-gnu-debuglink example.debug
```

Challenge 3: No debug info



Challenge 4: Cannot reach debugger server

- Debugger client uses TCP to connect to server
- The ephemeral container does not expose any port

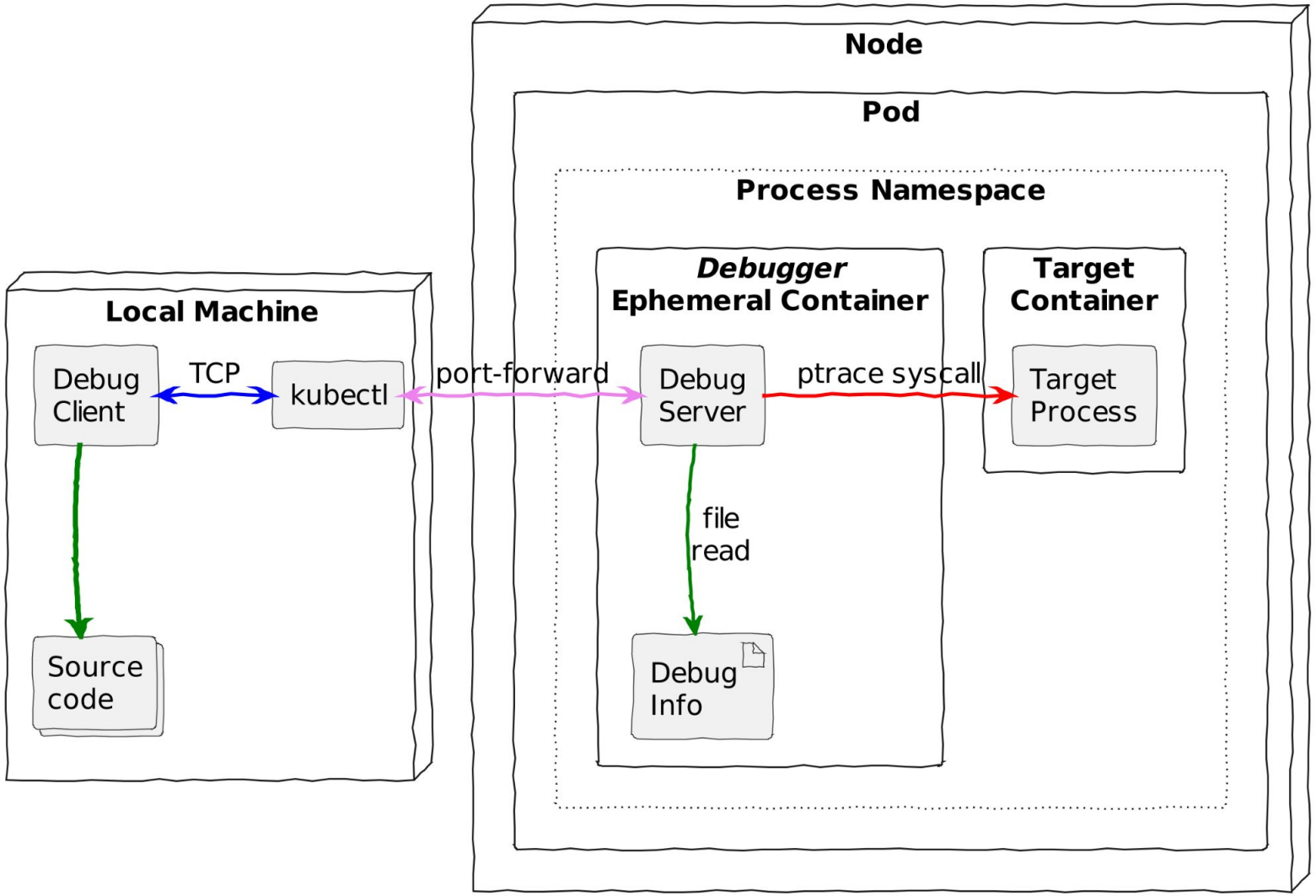
Challenge 4: Cannot reach debugger server

Solution

- Tunnel TCP using SPDY, via Kubernetes API server and kubelet

```
> kubectl port-forward example-pod \
    $DEBUG_CLIENT_PORT:2160
```

Challenge 4: Cannot reach debugger server



Challenge 5: Source code not found


```
could not find file /path/to/source/file.go
```

When you set a breakpoint, the debugger tries to find a match for the source code file and line number in the debug info.

If the debug info uses different paths, the debugger cannot find a match.

Challenge 5: Source code not found

Solution

1. Inspect the source code paths in the debug info.
2. Make a map from local source code paths to those in the debug info code.

Challenge 5: Source code not found

Solution

1. Inspect the source code paths in the debug info.

```
> readelf --debug-dump=decodedline example
```

Challenge 5: Source code not found

Solution

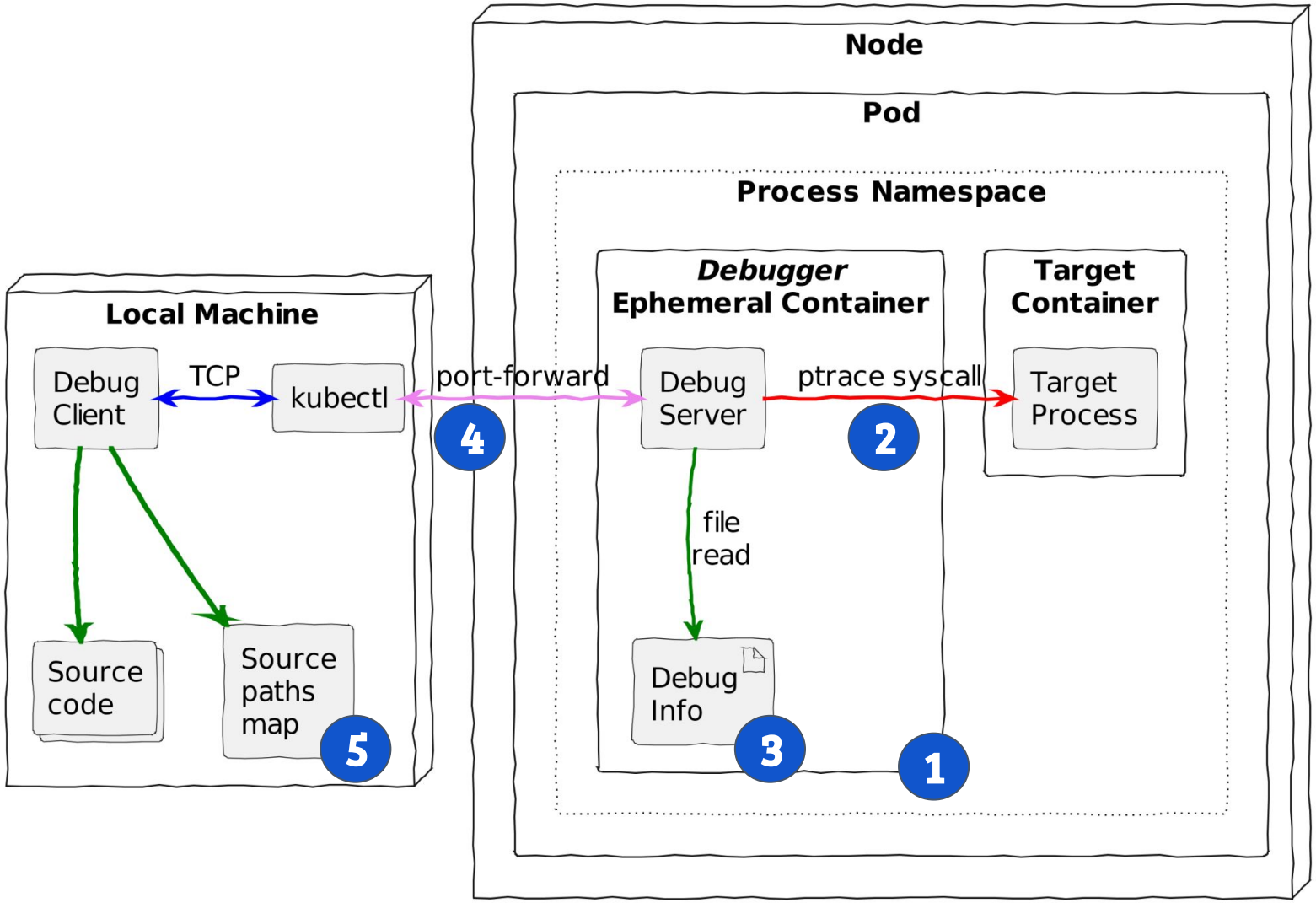
2. Make a map from local source code paths to those in the debug info code. For example:

from `/home/dlipovetsky/example-project/example.go`

to `github.com/dlipovetsky/example-project@v1.0/example.go`

Putting it all together

① Create debugger container
② Give container SYS_PTRACE capability
③ Create and upload debug info
④ Create tunnel for debugger client
⑤ Map local paths to paths in debug info



Other, less common challenges

Seen while stopped at a breakpoint:

- The kubelet terminates the target process, because it did not respond to liveness probes
- The target process terminates itself once you continue, because it lost a leader election
- The target process stops receiving the requests you want to investigate, because it did not respond to readiness probes

Other, less common challenges

Solutions

- Solutions depend on the application, and the debugger.
- I disabled leader election, and removed the liveness and readiness probes.
- I could have used logpoints (where the debugger logs an evaluated expression, instead of pausing execution).

Future Work

- Improve `kubectl debug`. The KEP roadmap include support for "profiles" that can, for example, set the right security context.
- Define a standard for distributing the debug info of containerized applications.

Call to action

- Set breakpoints in your Kubernetes applications.
- Teach others!

Questions? Want to collaborate?

@dlipovetsky on Kubernetes Slack

or find me at the D2IQ booth tomorrow

Thank you!

KubeCon | CloudNativeCon
Europe 2023

🥰 My wife & son

🤦 My D2IQ colleagues, especially software engineers Dimitri Koshkin & Shalin Patel

🙏 Derek Parker and Alessandro Arzilli for Delve

🙏 Lee Verberne for the Ephemeral Containers KEP

🙏 Kovid Goyal for kitty terminal

🙏 Fernand Galiana for k9s

Please scan the QR Code above
to leave feedback on this session