SQUASH

Debugger for microservices

Idit Levine

solo.io

About me

Idit Levine
Founder and CEO of solo.io


@Idit_Levine

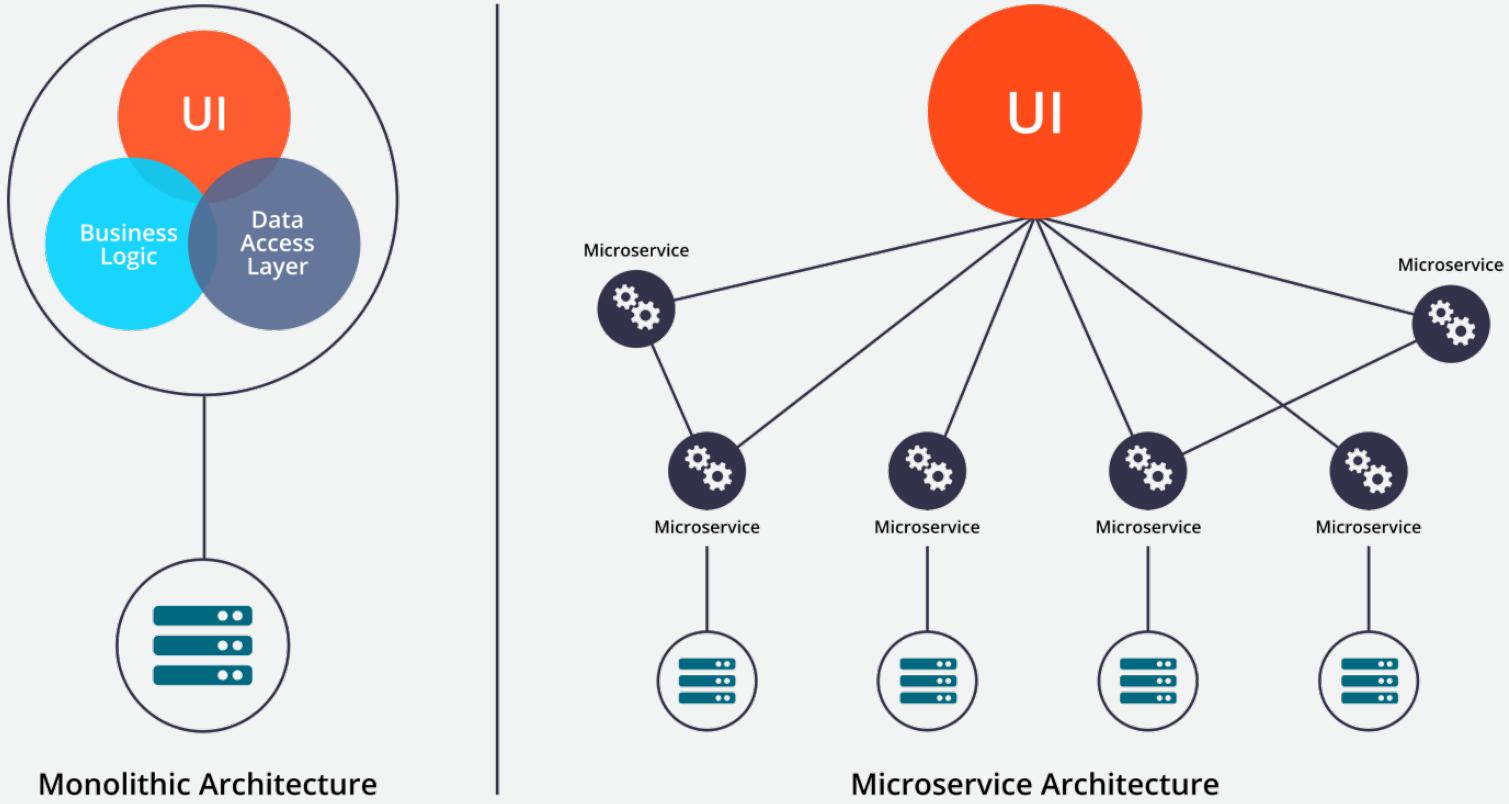

@ilevine

solo.io

The problem:

Debugging microservices applications is hard

The problem



A monolithic application consists of **a single process**

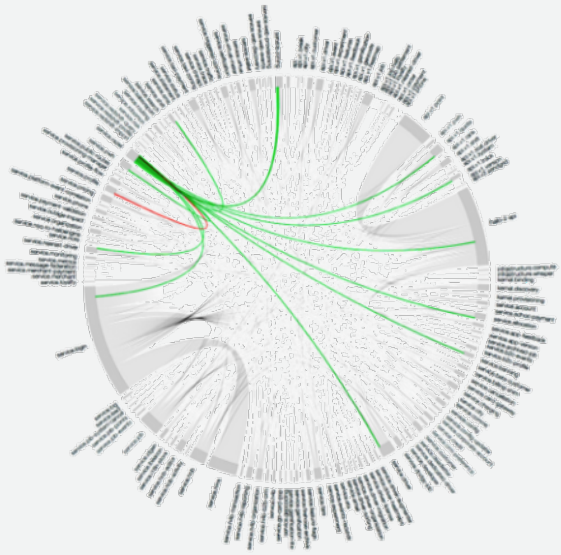
An attached debugger allows viewing the complete state of the application during runtime

A microservices application consists of potentially **hundreds of processes**

Is it possible to get a complete view of the state of a such application?!

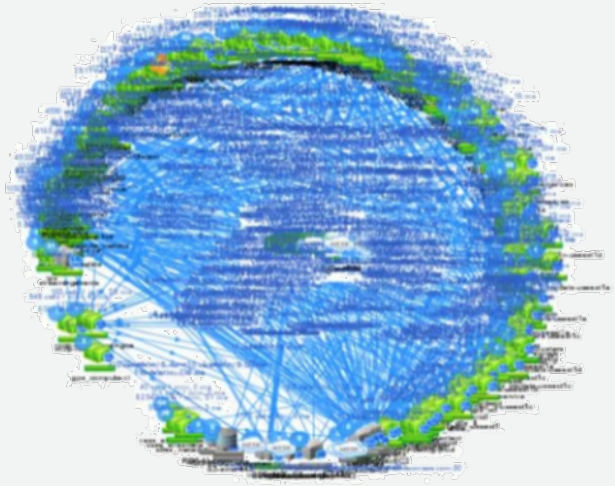
The problem

450+ microservices



HAIL O

500+ microservices



NETFLIX

500+ microservices

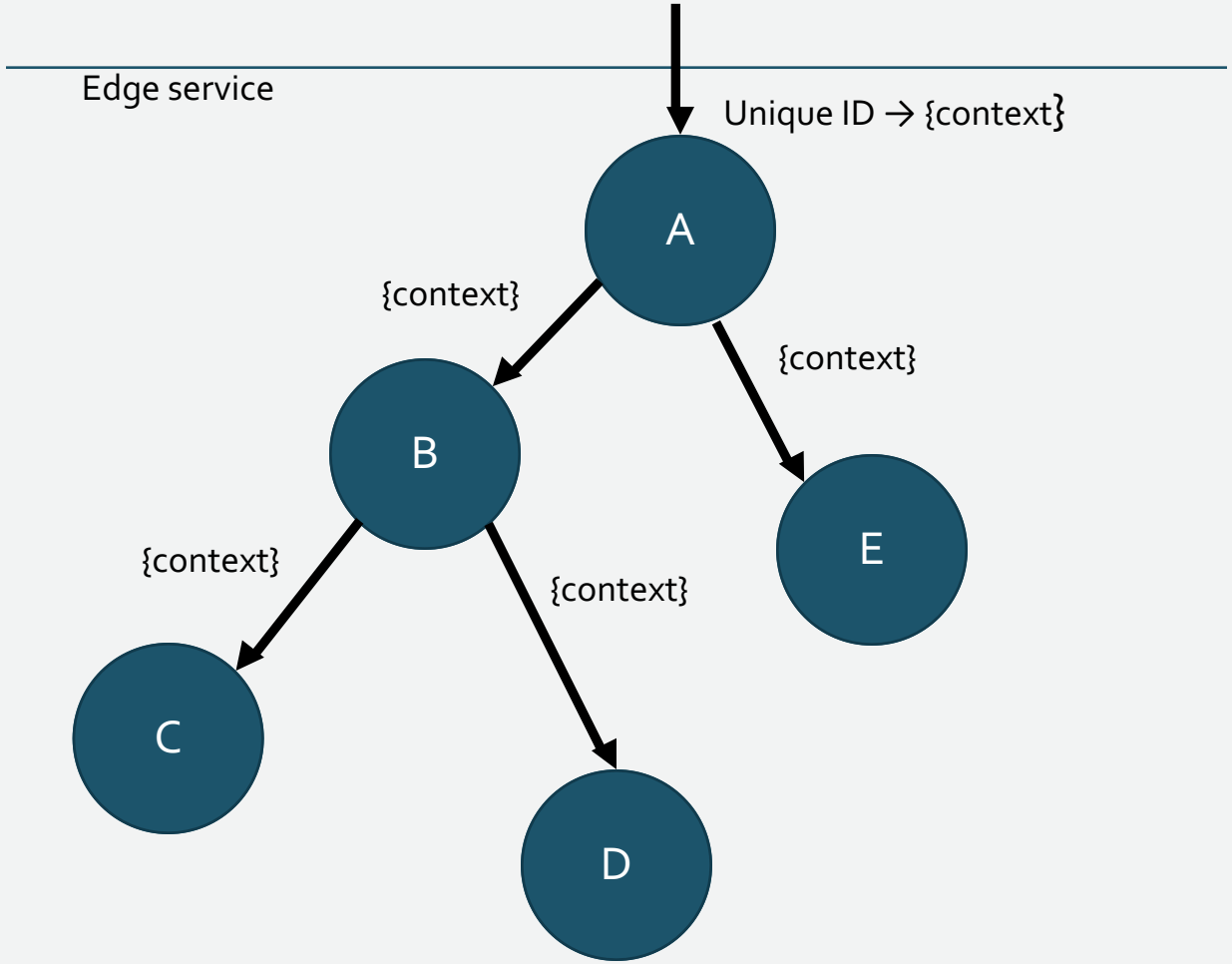
The problem


Honest Status Page
@honest_update
Follow
We replaced our monolith with micro services so that every outage could be more like a murder mystery.
4:10 PM - 7 Oct 2015
3,028 Retweets 2,476 Likes
20
3.0K
2.5K

Solution I

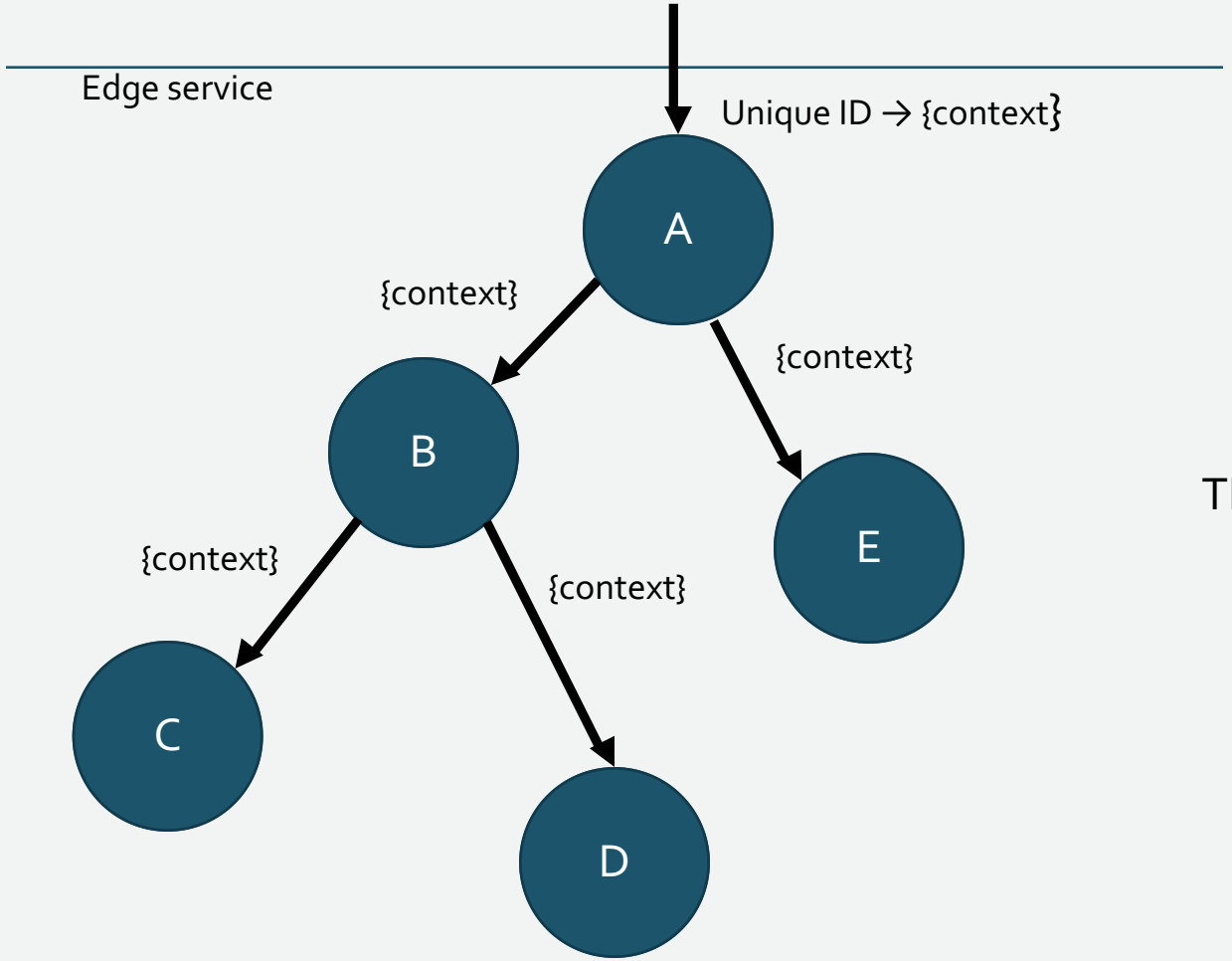
OpenTracing

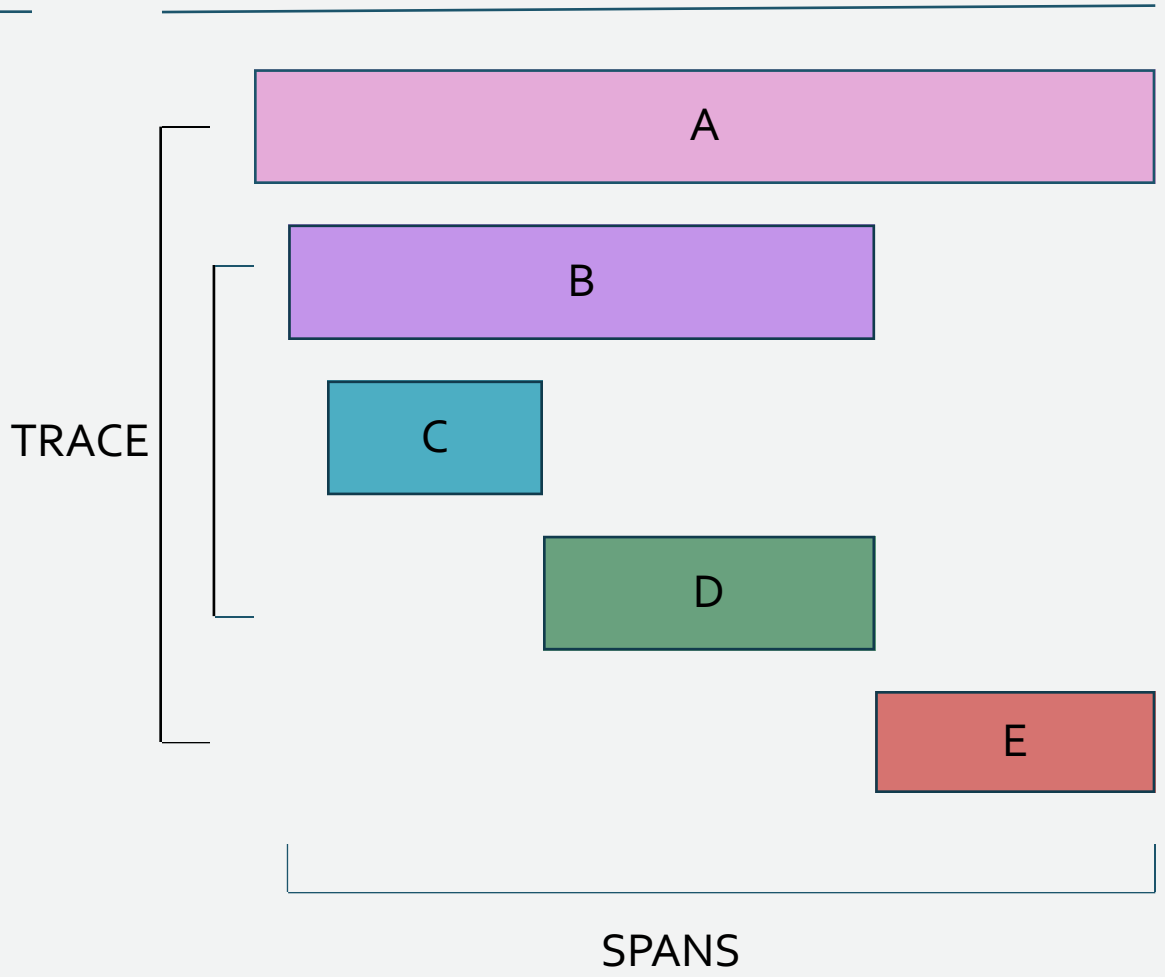
OpenTracing



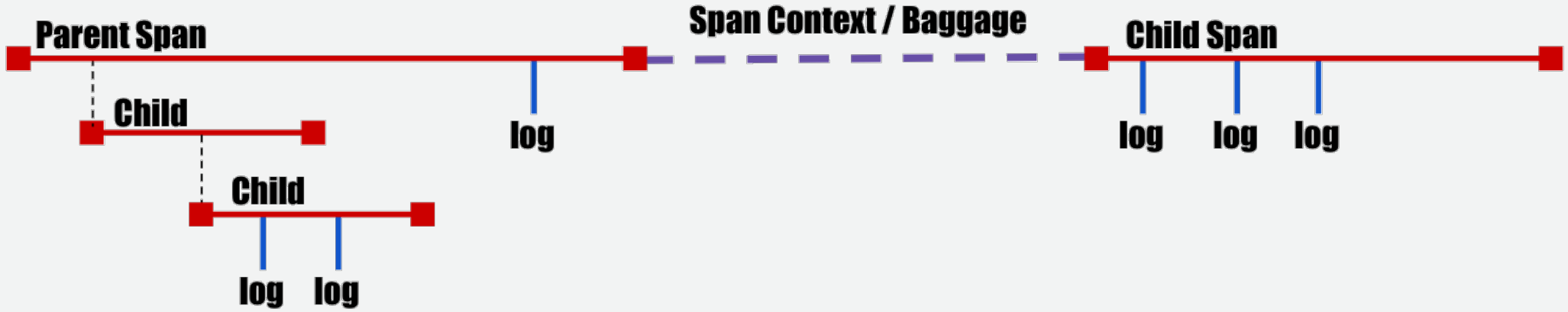
1. assign a ***unique identifier*** to each request at the edge service
2. store it in a ***context*** object, along with other metadata
3. ***propagate the context*** across process boundaries (in-band)
4. ***baggage*** is arbitrary K/V
5. capture timing, events, tags and collect them out-of-band (async)
6. re-assemble the call tree from the storage for the UI

OpenTracing





OpenTracing Architecture



spans - basic unit of timing and causality. can be tagged with key/value pairs.
logs - structured data recorded on a span.

span context - serializable format for linking spans across network boundaries.
carries baggage, such as a request and client IDs.

tracers - anything that plugs into the OpenTracing API to record information.
zipKin, jaeger & lightstep. but also metrics (Prometheus) and logging.

OpenTracing

OpenTracing is a consistent, expressive, vendor-neutral APIs for popular platforms, OpenTracing makes it easy for developers to add (or switch) tracing implementations with an O(1) configuration change.

CLOUD NATIVE
COMPUTING FOUNDATION

OpenTracing Demo

OpenTracing uses

logging - easy to output to any logging tool, even from OSS components.

metrics/alerting - measure based on tags, span timing, log data.

context propagation - use baggage to carry request and user ID's, etc.

critical path analysis - drill down into request latency in very high fidelity.

system topology analysis - identify bottlenecks due to shared resources.

OpenTracing limitations

openTracing does not provide ***run-time debugging***

openTracing requires ***wrapping and changing the code***

no holistic view of the application state – can ***only see what was printed***

the ***process*** (repeatedly modify the application and test) ***is expansive***

Impossible to change variable values in runtime

logging and printing results in ***performances overhead***

Solution II

Squash

Squash brings the power of modern popular debuggers to developers of microservices apps that run on container orchestration platforms.

Squash bridges between the orchestration platform (without changing it) and IDE.

With Squash, you can:

- Live debugging cross multi microservices
- Debug container in a pod
- Debug a service
- Set breakpoints
- Step through the code
- View and modify values of variables
- and more ...

Squash Demo

Squash Architecture

Squash server: holds the information about the breakpoints for each application, orchestrates and controls the squash clients. Squash server deploys and runs on Kubernetes

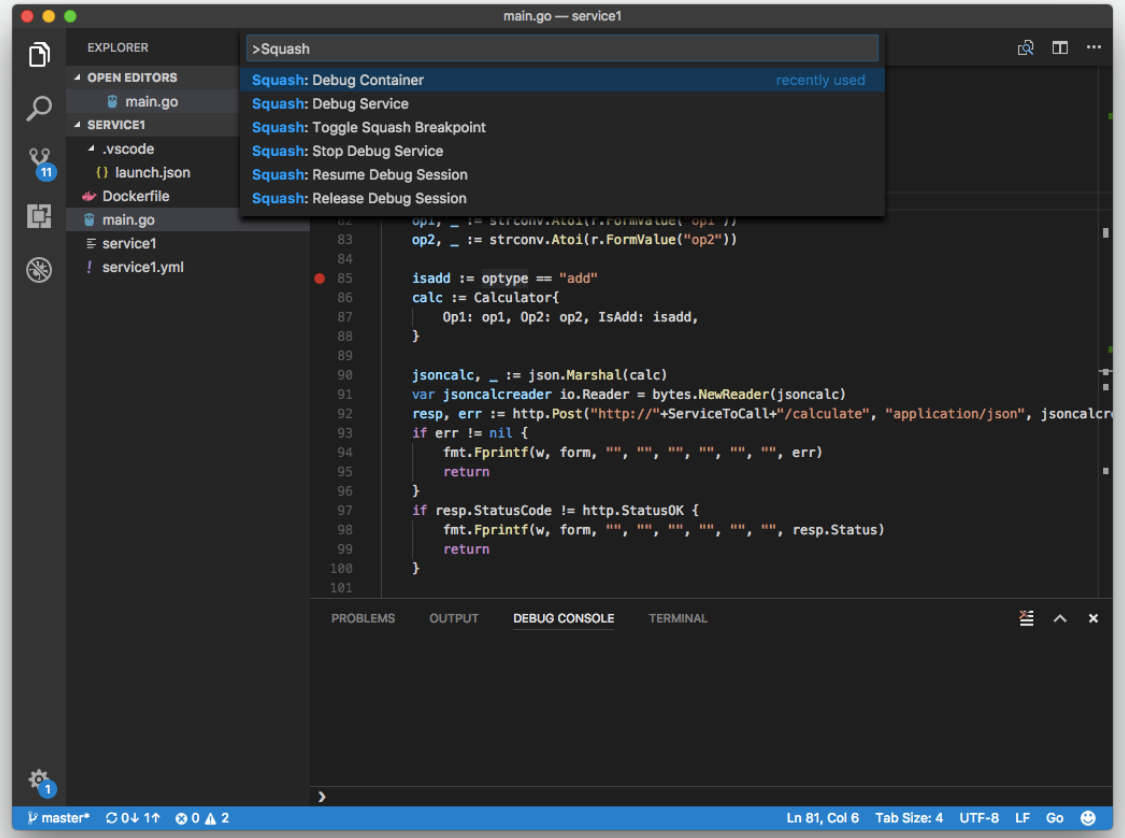
Squash client: deploys as daemon set on Kubernetes node. The client wraps as docker container, and also contains the binary of the debuggers.

Squash User Interface: squash uses IDEs as its user interface. After installing the Squash extension, Squash commands are available in the IDE command palette.

What vegetable scares all the bugs? Squash!"

one of my 8-year old daughter's favorite riddles

Squash Architecture: vs code extension



vs code extension ➜ ***kubectl*** to present the user pod/container/debugger options

vs code extension ➜ ***Squash server*** with debug config (pod/container/debugger /breakpoint) ➜ waits for debug session

vs code extension ➜ connects to the debug server & transfers control to the native debug extension.

Squash Architecture: Squash Server

vs code extension ➜ ***Squash server***

Squash server ➜ relevant ***Squash client*** with debug config (pod/container/debugger /breakpoint)

Squash server ➜ waits for debug session

Squash Architecture: Squash Client

Squash server ➜ ***Squash client***

Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ return debug session.

Squash Architecture: Squash Client

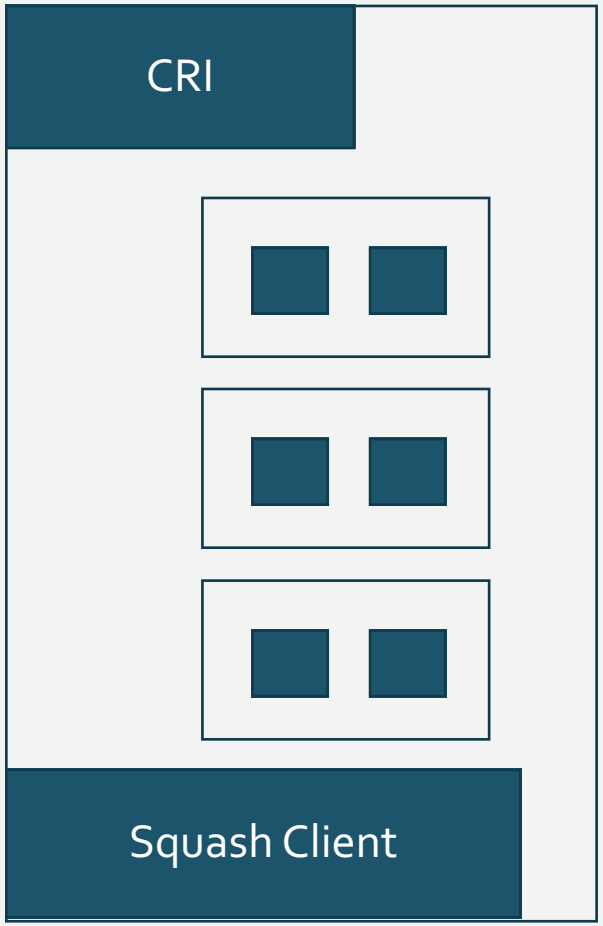
Squash server ➜ ***Squash client***

Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ return debug session.

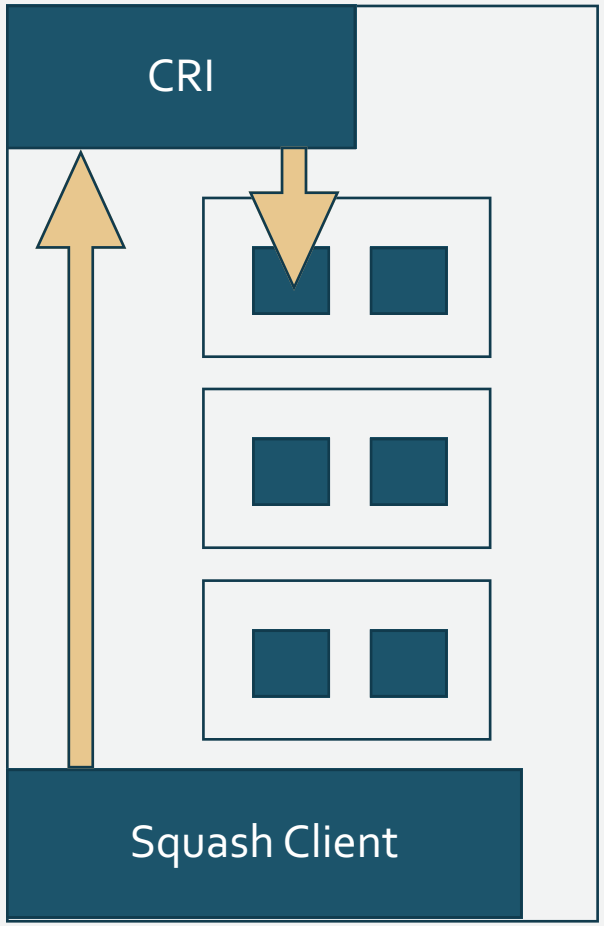
obtain host pid



Squash Client runs at the host namespace – we need to translate the pid of the process (application that run in the container) to the host pid namespace to allow debugger to attach.

- It is not going to be always container of docker type

obtain host pid



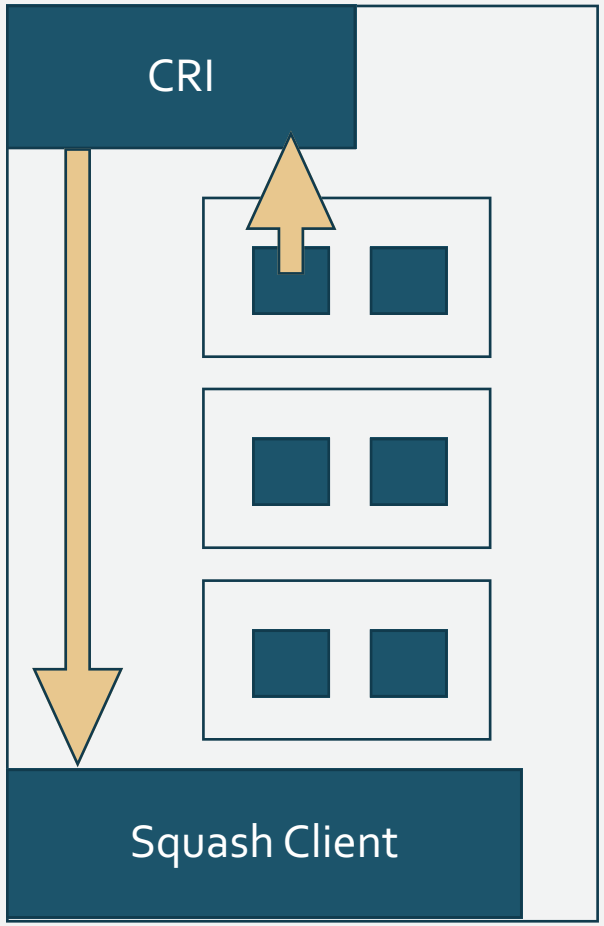
-> ls -l /proc/self/ns

total 0
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 net -> net:[4026532009]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 pid -> pid:[4026531836]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 pid_for_children -> pid:[4026531836]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 user -> user:[4026531837]
lrwxrwxrwx 1 idit idit 0 Dec 7 01:14 uts -> uts:[4026531838]

-> inod of mnt namespace (unique identifier to the container namespace)

via CRI api call ExecSyncRequest

obtain host pid



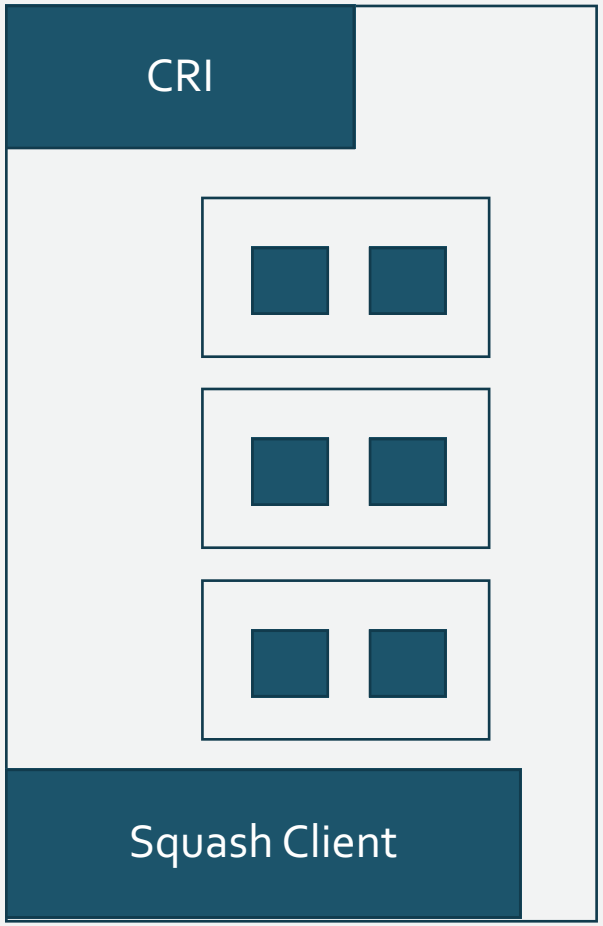

```
func FindPidsInNS(inod uint64, ns string) ([]int, error) {
   var res []int
   files, err := ioutil.ReadDir("/proc")
   if err != nil {
      return nil, err
   }

   for _, f := range files {
      if !f.IsDir() {
         continue
      }
      pid, err := strconv.Atoi(f.Name())
      if err != nil {
         continue
      }

      p := filepath.Join("/proc", f.Name(), "ns", ns)
      if inod2, err := processwatcher.PathToInode(p); err != nil {
         continue
      } else if inod == inod2 {
         res = append(res, pid)
      }
   }

   return res, nil
}
```

obtain host pid



Squash Client runs at the host namespace – we need to translate the pid of the process (application that run in the container) to the host pid namespace to allow debugger to attach.

- It is not going to be always container of docker type
- What if the container cannot run ls ?

Squash Architecture: Squash Client

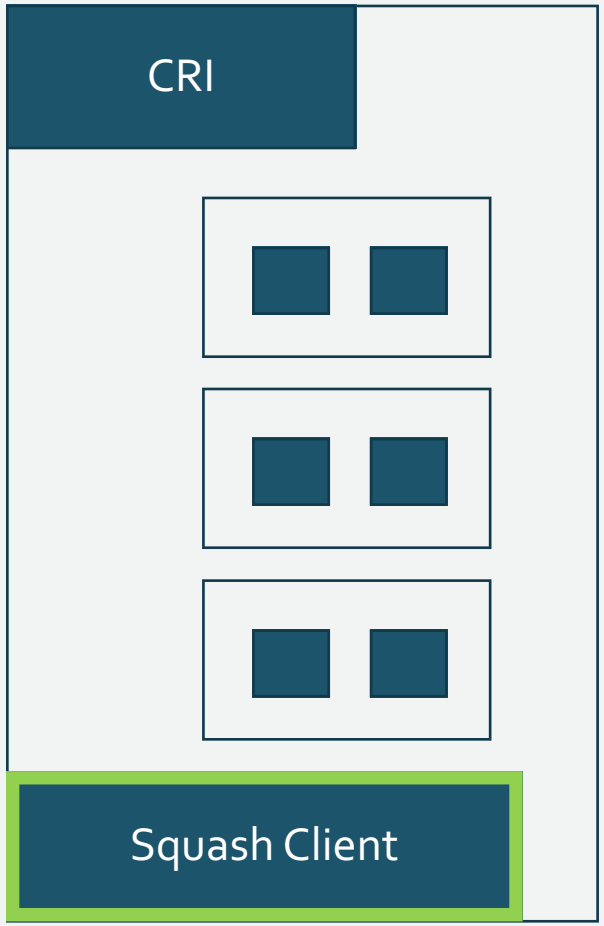
Squash server ➜ ***Squash client***

Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ return debug session.

squash client: debuggers




```
FROM ubuntu:16.04

RUN apt-get update
RUN apt-get install --yes gdb build-essential
RUN apt-get install --yes git
RUN apt-get install --yes curl
# RUN apt-get install --yes golang-1.8-go


RUN curl https://storage.googleapis.com/golang/go1.8.linux-amd64.tar.gz | tar -C /usr/lib -xz

ENV GOROOT /usr/lib/go
ENV GOPATH /gopath
ENV GOBIN /gopath/bin
ENV PATH $PATH:$GOROOT/bin:$GOPATH/bin

RUN mkdir -p $GOPATH/src/github.com/derekparker/ && cd $GOPATH/src/github.com/derekparker/ && git clone https://github.com/derekparker/delve/
RUN cd $GOPATH/src/github.com/derekparker/delve/ && git checkout v1.0.0-rc.1
RUN cd $GOPATH/src/github.com/derekparker/delve/cmd/dlv && go install

COPY squash-client /

EXPOSE 1234

ENTRYPOINT ["/squash-client"]
```

Squash Architecture: Squash Client

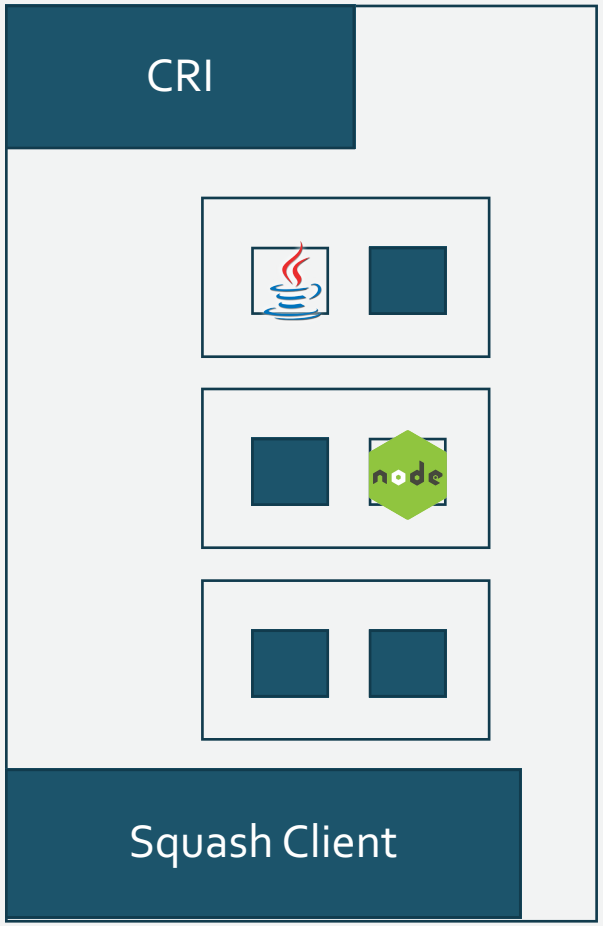
Squash server ➜ ***Squash client***

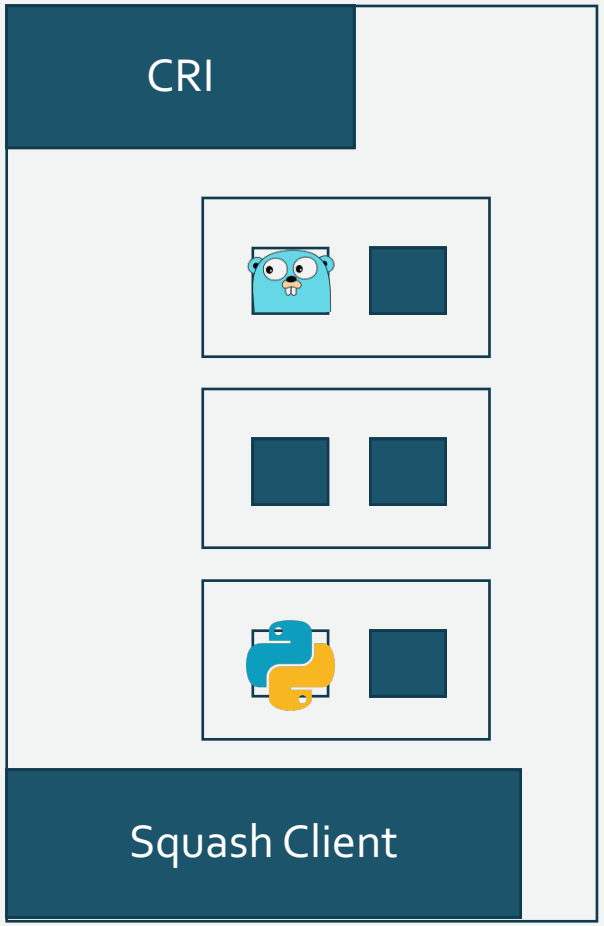
Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ return debug session.

multi languages support





Squash high level Architecture

Platforms | IDEs | Debuggers

Squash high level Architecture

Add Mesos/Marathon platform support #13

Open **cdennison** opened this issue 2 days ago · 5 comments

cdennison commented 2 days ago

Hi everyone,

I was thinking about adding Mesos/Marathon support, but there is a technical limitation on the tooling side - they don't support "docker exec" yet from their CLI - for doing things like "getting the pid."

It looks like Docker Swarm has the same issue (**@crackerplace**) - there are third party tools like this but nothing bundled with the official tool.

Here are a couple ideas I had for how to achieve that functionality, but none of them are ideal so I'd love your thoughts. I can also jump on slack to discuss further.

Goal is to achieve this:

```
req := &kubeapi.ExecSyncRequest{
        ContainerId: containerid,
        Cmd:         []string{"ls", "-l", "/proc/self/ns/"},
        Timeout:     1,
}

ctx, cancel := context.WithTimeout(origctx, time.Second)
result, err := cli.ExecSync(ctx, req)
```

Add Swarm platform support #11

Open **crackerplace** opened this issue 10 days ago · 5 comments

crackerplace commented 10 days ago

Just started getting some comfort level with the code.
Would like to to add swarm support for squash incrementally.

Add support for python's pdb and/or the iPython debugge #8

Open **SEJeff** opened this issue on Sep 7 · 1 comment

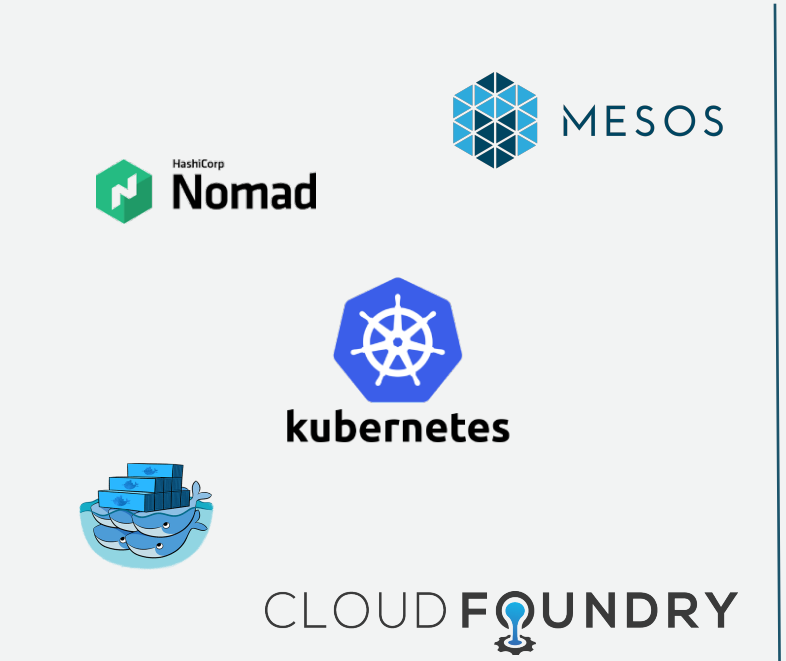
SEJeff commented on Sep 7

For those of us who run python web applications in production

Squash high level Architecture

Platforms | IDEs | Debuggers

Squash high level Architecture



Platforms

IDEs

Debuggers

Platform Interface

```
/// Minimal represntation of a container, containing only the data squash cares about -
/// The container's name, image and the node it runs on.
type Container struct {
   Name, Image, Node string
}

/// Runs in the squash server:

/// Get the container object from its name.
// Note: in environment like kubernetes, the containername will be namespace:pod-name:container-name
type ContainerLocator interface {
   Locate(context context.Context, attachment interface{}) (interface{}, *Container, error)
}

/// Runs in the squash client:

/// Get the pid of a process that runs in the container. the pid should be in our pid namespace,
/// not in the container's namespace.
type Container2Pid interface {
   GetPid(context context.Context, attachment interface{}) (int, error)
}

type DataStore interface {
   Store()
   Load()
}
```

Debuggers interface

```
package debuggers

type DebugServer interface {
   Detach() error
   Port() int
}

/// Debugger interface. implement this to add a new debugger support to squash.
type Debugger interface {

   /// Attach a debugger to pid and return the port that the debug server listens on.
   Attach(pid int) (DebugServer, error)
}
```

DELVE
A Debugger for the Go Programming Language

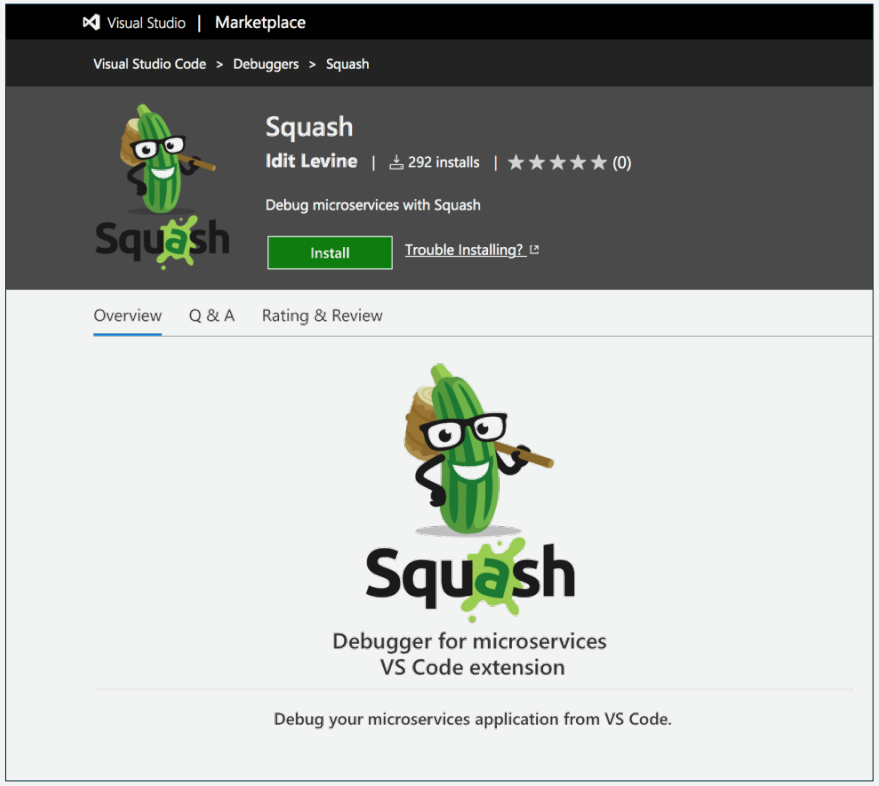
GDB
The GNU Project Debugger

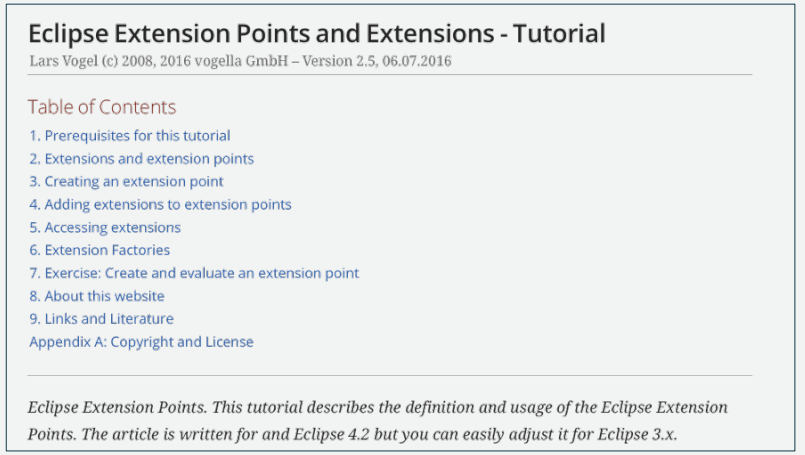
IP[y]:
IPython

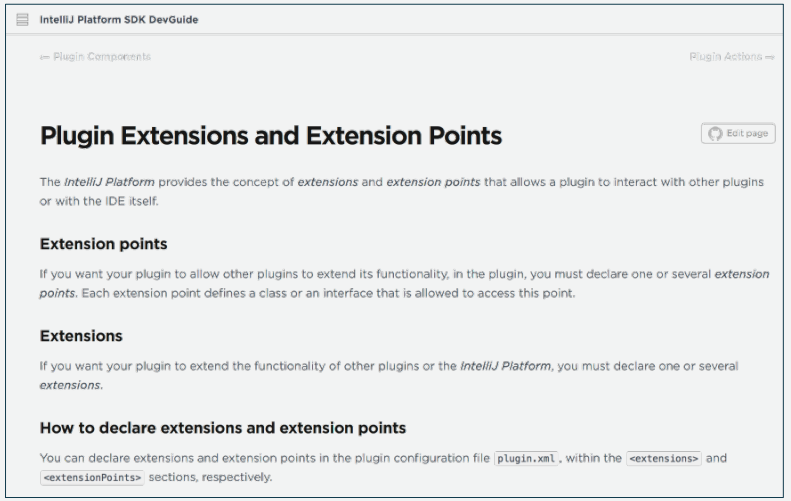
python
Debugging
with pdb

… and add it to the client docker file

Squash: IDE


Visual Studio | Marketplace
Visual Studio Code > Debuggers > Squash
Squash
Idit Levine | 292 installs | ★★★★★ (0)
Debug microservices with Squash
Install
Trouble Installing?
Overview
Q & A
Rating & Review
Squash
Debugger for microservices
VS Code extension
Debug your microservices application from VS Code.


Eclipse Extension Points and Extensions - Tutorial
Lars Vogel (c) 2008, 2016 vogella GmbH – Version 2.5, 06.07.2016
Table of Contents
1. Prerequisites for this tutorial
2. Extensions and extension points
3. Creating an extension point
4. Adding extensions to extension points
5. Accessing extensions
6. Extension Factories
7. Exercise: Create and evaluate an extension point
8. About this website
9. Links and Literature
Appendix A: Copyright and License
Eclipse Extension Points. This tutorial describes the definition and usage of the Eclipse Extension Points. The article is written for and Eclipse 4.2 but you can easily adjust it for Eclipse 3.x.


IntelliJ Platform SDK DevGuide
Plugin Components
Plugin Actions
Plugin Extensions and Extension Points
Edit page
The IntelliJ Platform provides the concept of extensions and extension points that allows a plugin to interact with other plugins or with the IDE itself.
Extension points
If you want your plugin to allow other plugins to extend its functionality, in the plugin, you must declare one or several extension points. Each extension point defines a class or an interface that is allowed to access this point.
Extensions
If you want your plugin to extend the functionality of other plugins or the IntelliJ Platform, you must declare one or several extensions.
How to declare extensions and extension points
You can declare extensions and extension points in the plugin configuration file plugin.xml, within the <extensions> and <extensionPoints> sections, respectively.

Squash: open source project

We are looking for community help to add support for more debuggers, platforms and IDEs.

Check out at github:
https://github.com/solo-io/squash

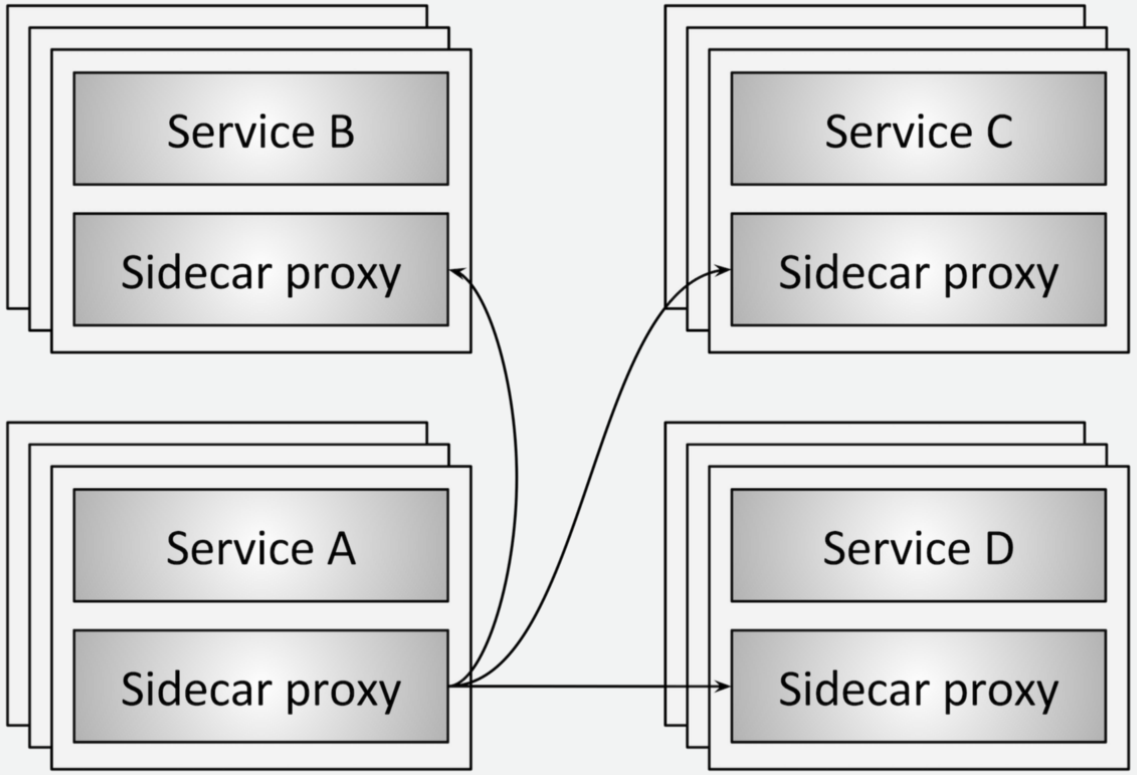
Solution III

Service mesh

Service Mesh

Service mesh data plane:
Touches every packet/request in the system. Responsible for ***service discovery***, ***health checking***, ***routing***, ***load balancing***, ***authentication/authorization***, and ***observability***.

Service mesh control plane:
Provides ***policy*** and ***configuration*** for all the running data planes in the mesh. Does not touch any packets/requests in the system. ***The control plane turns all of the data planes into a distributed system***.



Envoy – data plane

The network should be transparent to applications. When network and application problems do occur it should be easy to determine the source of the problem.

Out of process architecture: developers to focus on business logic

Modern C++11 code base: Fast and productive.

L3/L4 filter architecture: Can be used for things other than HTTP (TCP proxy at its core)

HTTP L7 filter architecture: Make it easy to plug in different functionality.

HTTP/2 first! (Including gRPC and a nifty gRPC HTTP/1.1 bridge).

Service discovery and **active health checking.**

Advanced ***load balancing***: Retry, timeouts, circuit breaking, rate limiting, shadowing, etc.

Best in class ***observability***: stats, logging, and tracing.

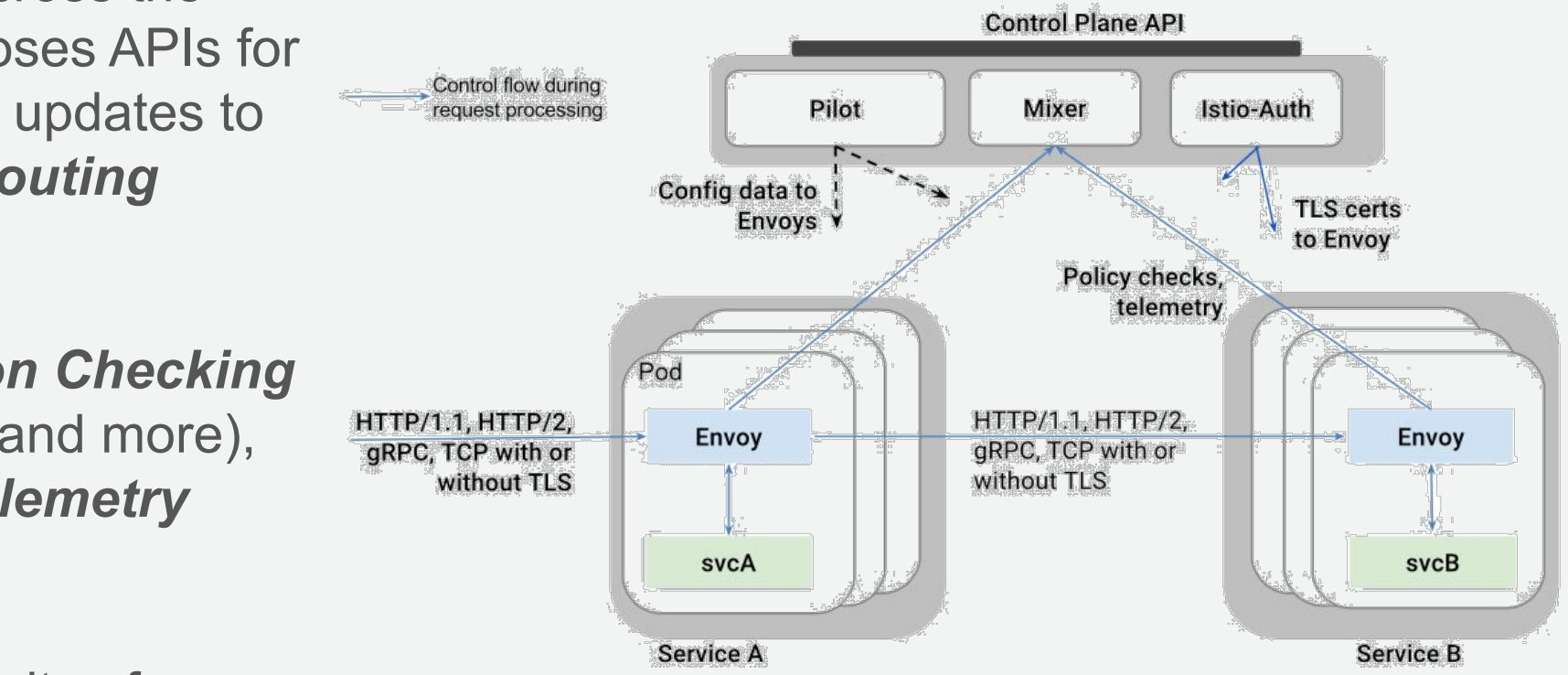
Edge proxy: ***routing*** and ***TLS***.

Istio – control plane

Pilot: responsible for the lifecycle of Envoy instances deployed across the Istio service mesh. Pilot exposes APIs for ***service discovery***, dynamic updates to ***load balancing pools*** and ***routing tables***.

Mixer: provides ***Precondition Checking*** (authentication, ACL checks and more), ***Quota Management*** and ***Telemetry Reporting.***

Istio-Auth enhance the security of microservices and their communication without requiring service code changes.



Istio Architecture

Towards an integrated solution

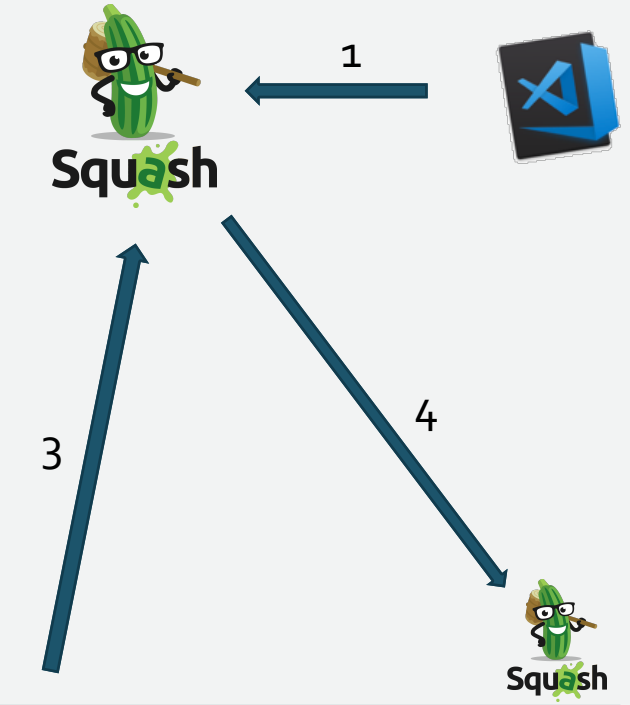
Service mesh, OpenTracing, and Squash

The whole solution

Step 1:
vs code extension ➜ ***Squash server*** creates a debug config (service & image) and waits for the debug session to connect.

Step 2:
envoy gets a curl request with squash header

Step 3:
➜ ***envoy*** asks ***Squash server*** to debug itself (namespace & pod) and waits for the debug session.



The whole solution

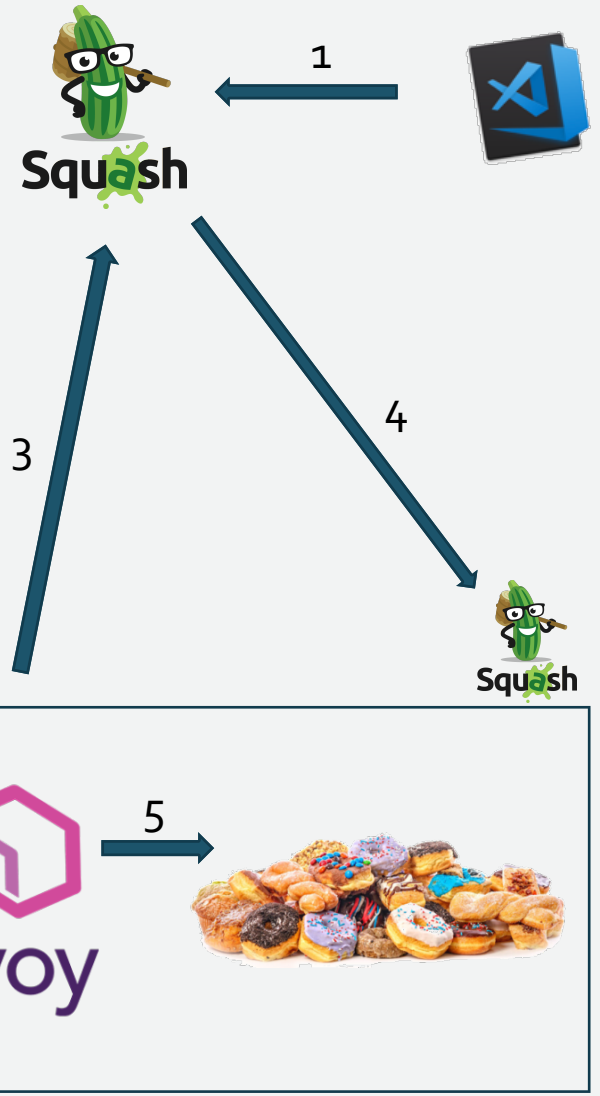
Step 4:
Squash server ➜ ***Squash client***

Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ returns debug session.

Step 5:
vs code extension ➜ connects to the debug server & transfers control to the native debug extension.
envoy resumes traffic to the app



Envoy plugin

```
Envoy::Http::FilterHeadersStatus
SquashFilter::decodeHeaders(Envoy::Http::HeaderMap& headers, bool ) {

if (squash_cluster_name_.empty()) {
ENVOY_LOG(warn, "Squash: cluster not configured. ignoring.");
return Envoy::Http::FilterHeadersStatus::Continue;
}

// check for squash header
const Envoy::Http::HeaderEntry* squasheader =
headers.get(Envoy::Http::LowerCaseString("x-squash-debug"));

if (squasheader == nullptr) {
ENVOY_LOG(warn, "Squash: no squash header. ignoring.");
return Envoy::Http::FilterHeadersStatus::Continue;
}

// get pod and container name
const char* podc = std::getenv("POD_NAME");
if (podc == nullptr) {
ENVOY_LOG(warn, "Squash: no podc. ignoring.");
return Envoy::Http::FilterHeadersStatus::Continue;
}
std::string pod(podc);
if (pod.empty()) {
ENVOY_LOG(warn, "Squash: no pod string. ignoring.");
return Envoy::Http::FilterHeadersStatus::Continue;
}
```

only be added if squash server install &
not in squash pods – configuration in pilot

```
const char* podnamespacec = std::getenv("POD_NAMESPACE");
if (podnamespacec == nullptr) {
ENVOY_LOG(warn, "Squash: no podnamespacec. ignoring.");
return Envoy::Http::FilterHeadersStatus::Continue;
}
std::string podnamespace(podnamespacec);
if (podnamespace.empty()) {
ENVOY_LOG(warn, "Squash: no container string. ignoring.");
return Envoy::Http::FilterHeadersStatus::Continue;
}

ENVOY_LOG(info, "Squash:we need to squash something");

// get squash service cluster object
// async client to create debug config at squash server
// when it is done, issue a request and check if it is attached.
// retry until it is. or until we timeout
// continue decoding.
Envoy::Http::MessagePtr request(new Envoy::Http::RequestMessageImpl());
request->headers().insertContentType().value(std::string("application/json"));
request->headers().insertPath().value(std::string("/api/v2/debugattachment"));
request->headers().insertHost().value(std::string("squash-server"));
request->headers().insertMethod().value(std::string("POST"));
std::string body = "{\"spec\":{\"attachment\":{\"pod\":\"" + pod + "\",\"namespace\":\"" +
podnamespace + "\"}, \"match_request\":true}}";
request->body().reset(new Envoy::Buffer::OwnedImpl(body));

state_ = CREATE_CONFIG;
in_flight_request_ =
cm_.httpAsyncClientForCluster(squash_cluster_name_).send(std::move(request), *this,
timeout_);

return Envoy::Http::FilterHeadersStatus::StopIteration;
}
```

Istio – envoy leverage

Debug in production without pausing the cluster!

- Pilot support for envoy plugins – today hardcoded (Envoy plugin extension without recompile

- We will work with it with envoy and istio team and contribute the code upstream

Service Mesh Demo

Future ideas

- Can automate by leveraging similar mechanism of envoy retries:
  - on getting response of 500 (internal errors) run the request with squash header.

- Integration with github

- Web browser IDE

- Integration with OpenTracing

- Detect latency and zoom in the debug

Check Squash out: *github.com/solo-io/squash*

Squash