EnTrance design notes

Contents

[Introduction 2](#_Toc5878699)

[Who’s the boss? 3](#_Toc5878700)

[Concepts and terminology 4](#_Toc5878701)

[Basics 4](#_Toc5878702)

[Startup 5](#_Toc5878703)

[RPCs 6](#_Toc5878704)

[Server modularity 7](#_Toc5878705)

[Client modularity 9](#_Toc5878706)

[Feature lifecycle 10](#_Toc5878707)

[Connection handling 14](#_Toc5878708)

[Introduction 14](#_Toc5878709)

[Concepts and terminology 14](#_Toc5878710)

[Target features 16](#_Toc5878711)

[Multi-user apps 18](#_Toc5878712)

[Introduction 18](#_Toc5878713)

[Auth Features 18](#_Toc5878714)

[Security models 19](#_Toc5878715)

[Appendix – Feature class hierarchy 20](#_Toc5878716)

# Introduction

This is a quick set of notes on the structure of the framework. The goal is to provide some basic orientation in the absence of a full FS/DS.

EnTrance is a framework for easily producing simple web apps, using a Python3 backend and Elm frontend. The target audience is teams who mostly spend their time doing something else, and just quickly want to spin off GUI experiences every now and then. So correctness and maintainability are priorities – it should be easy for someone new to pick up a tool, do a little bit of work on it, and then leave it for months, without incurring technical debt. It is also a priority to make it easy to get something working quickly that looks decent-ish with just a single unified toolchain (without requiring eg lots of CSS knowledge or lots of javascript plugins).

An EnTrance application runs on an “evergreen” web browser that keeps itself reasonably up-to-date; legacy browsers are a non-target. The framework has everything required to make these applications responsive and mobile-friendly, but it is up to the application to decide how much effort to put in to optimising this experience.

# Who’s the boss?

In “Web 1.0” apps, everything happened on the server, including all the HTML rendering. “Web 2.0” sprinkled on enough javascript to give a richer interactive experience, but the source of truth for application state was usually still the backend.

However, the trend with modern web apps is for the source of truth to be on the client. There may not be a backend at all – so-called “serverless” – or more precisely, the backend services may be splintered across many microservices (eg one for authentication, another for database access, yet another for messaging APIs) that know nothing about each other. The communication between client and server is in terms of abstract messaging (eg JSON objects) and the UI rendering is done by the client.

EnTrance embraces this trend. So do not think of an EnTrance app as “a server-side Python app, with some client-side logic to enhance the UI”. The correct mental model is “a client-side Elm app, that calls out to the passive server when it needs a non-local service”. This is also, conveniently, a simpler programming model.

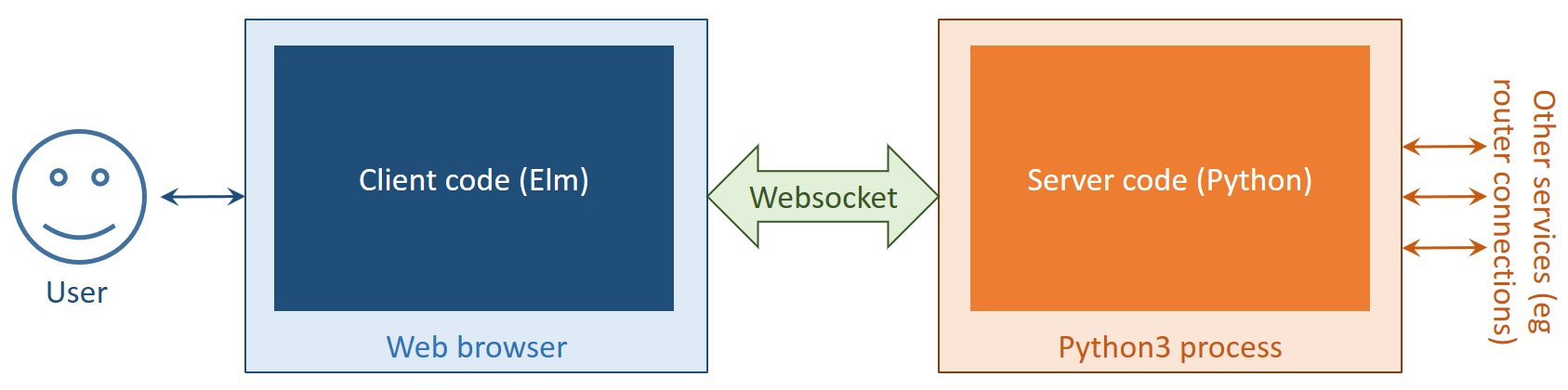
It also means code management is simplified. Each app has its own repo that includes the full front end code (which tends to be tailored to each use case, and thus not easily shareable). But the server side does not require any app-specific python whatsoever – it can just invoke the generic entrance package (that could be pushed to PyPi) from a shell script; the javascript and css served from the static directory determine what the app then does.

If an app does require more complex server-side functionality specific to that app, then of course it is easy to add that. But the shared package remains an inert buffet of server-side functionality from which each app selects, and this buffet can grow over time to cover a growing set of functionality and application types.

# Concepts and terminology

## Basics

The most coarse modular breakdown is between client and server:



The websocket provides an asynchronous reliable bidirectional communication channel. EnTrance uses JSON as the encoding format within this. By convention, a **request** is a JSON-encoded message sent from client to server, and a **notification** is a JSON-encoded message sent from server to client.

This convention is purely to give a consistent naming structure to the code. Requests and notifications are entirely symmetrical, and can be asynchronously issued whenever the client or server (respectively) feels like sending one. This document distinguishes request and notification example messages using colour, but there is no ambiguity if reading in monochrome.

Every request has a req\_type string field indicating what sort of request it is, and every notification has a symmetrical nfn\_type string field indicating what sort of notification it is. For example, the client can check the basic liveness of the connection and server by issuing a request consisting simply of {"req\_type":"ping"}. The server should reply with a notification of {"nfn\_type":"pong"}.

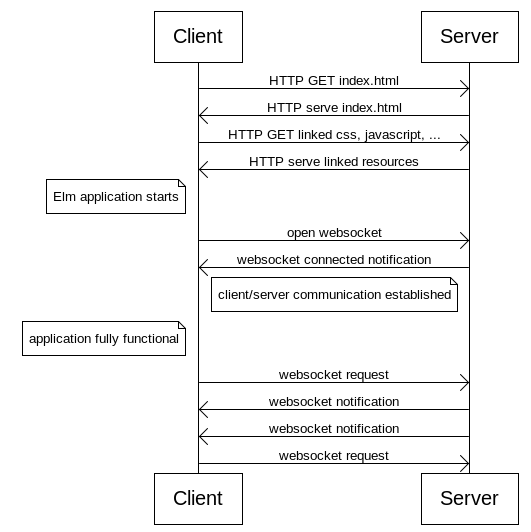
There will be enough JSON fragments in this document to give even a minor typographical optimization some value. So from now on, fragments in ⟨angle brackets⟩ will be JSON objects with string-typed keys, from which the quotes around keys are omitted. So the examples above would be displayed as ⟨req\_type: "ping"⟩ and ⟨nfn\_type: "pong"⟩

Both requests and notifications may have additional fields not expected by the receiver; these are safely ignored.

To further disambiguate terminology, a **connection** always refers to a connection from the server to some other entity (eg a router). The connection between the client and server is always referred to simply as the **websocket**.

## Startup

The client code has to get into the browser somehow, along with any other web assets such as CSS and image files. So the server is also a regular HTTP server, serving static files; one of these is a javascript file which is the compilation target of the Elm source. So the startup sequence actually looks more like the following:



## RPCs

A certain class of request/notification pairs occurs so frequently as to have a special convention: RPCs initiated by the client. An RPC is a request, for which the client is expecting exactly one reply notification, and that reply indicates either success of failure.

Other unrelated requests and notifications can flow freely during the interval between RPC request and notification of either success or failure. So the RPC itself represents a synchronous operation, but without blocking the underlying asynchronous websocket transport.

RPCs have the following additional conventions, which are simplified versions of the JSON-RPC spec:

* The client supplies a numeric id field that is reflected back in the resulting notification. (The client is expected to insert a sufficiently unique value here, for it to correlate the notification with the original request.)
* The resulting notification has the same nfn\_type value as the original req\_type, and also contains exactly one of:
  + A result field (of arbitrary type) on success
  + An error string field on error

For example, the “CLI Exec” server feature enables interactions like this (simplified for brevity):

[C🡪S] ⟨req\_type: "cli\_exec", id: 42, command: ”show clock”⟩

[S🡨C] ⟨nfn\_type: "cli\_exec", id: 42, result: "10:20:30 UTC Jan 10 2010"⟩

[C🡪S] ⟨req\_type: "cli\_exec", id: 43, command: "show weasels"⟩

[S🡨C] ⟨nfn\_type: "cli\_exec", id: 43, error: "Invalid input"⟩

As with all subsequent examples, this uses the abbreviated JSON syntax described in the Basics section above.

Note that an RPC requires the client to care about the success/failure state (and probably have some UI to represent that to the user). There are plenty of other client-server interactions where simple send-and-forget asynchronous messages are more appropriate.

The Python code reflects back the id field for any request where it is present. (Code note: the Feature base class also has helper \_rpc\_success and \_rpc\_failure methods to easily implement the result/error part of an RPC notification response.)

The websocket javascript code (Elm dropped native support for websockets in version 0.19, so a small amount of supplementary javascript is required, that is invoked over Elm ports) sends unique id fields when appropriate, and validates that RPC replies have the correct id. (If not, eg for an out-of-order reply from an old stale slow request, then the notification is dropped with a javascript console log.)

## Server modularity

The server has a set of infrastructure that is independent of any actual websocket-interacting functionality. Every module that actually does something for the client is called a **feature**. This is a subclass of the Feature class in entrance/feature/base.py.

Each websocket to a client has its own set of feature instances – typically a subset of all the Feature classes available – that are relevant to that client. The client can request optional features to be started for that websocket; see later sections for more details.

Each feature specifies a simple schema for the req\_type values that it listens for (along with the other parameters it expects, such as command in the cli\_exec example above), and the infrastructure validates requests against the schema and dispatches valid requests accordingly.

In code terms, for example the entrance/feature/tgt\_cli\_exec.py feature class has

my\_requests = {'cli\_exec': ['command']}

indicating that an incoming request with a req\_type of "cli\_exec" belongs to an instance of that class that is running on any given websocket, and it requires an additional command value to be parsed out the JSON message. If this is missing then the infrastructure returns an error without executing any feature code. Otherwise, the do\_cli\_exec method is invoked to handle the request. (There is a magic argument value of "\_\_req\_\_" that supplies the entire request dict to the handler, if it needs to do fancier message parsing.)

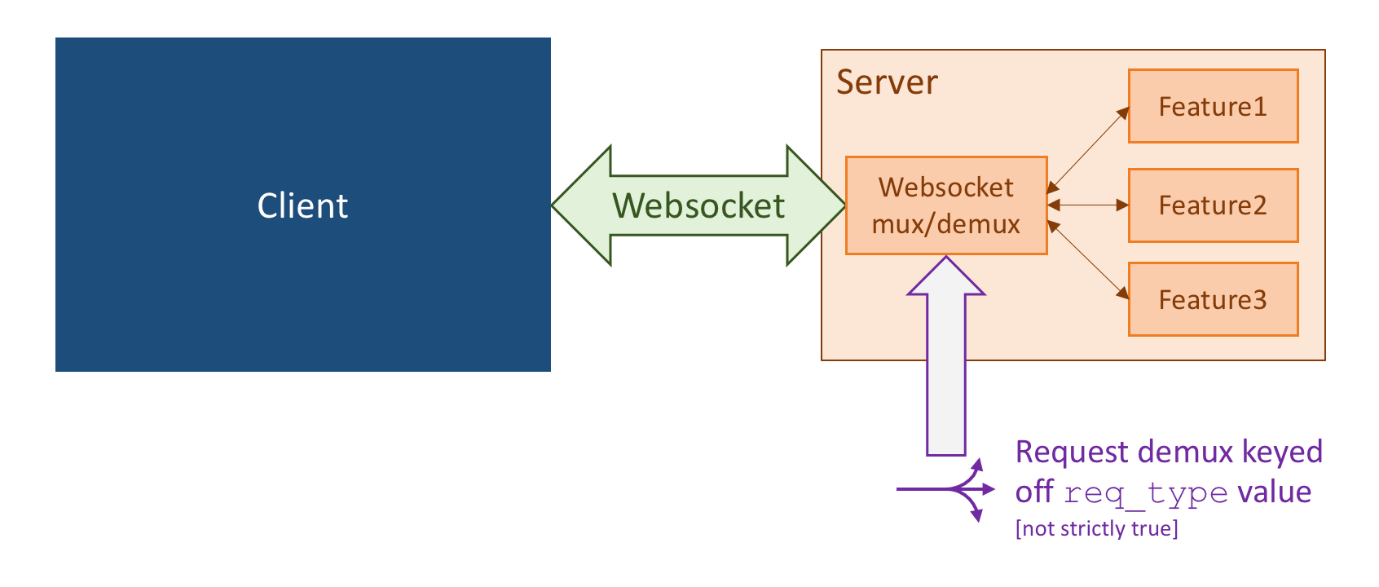
The schema also specifies notifications that the feature can send; all requests are automatically permissible notifications (since the RPC convention is so common) but additional ones have to be specified (or else the app will intentionally crash). For example, entrance/feature/cfg\_core.py has

notifications = ['pong', 'websocket\_up']

because these two notifications are not RPC responses.

The mapping of req\_type to running feature instances is maintained independently per-websocket, so each client can have a different set of features running, and thus a different set of req\_type values that will get a valid response.

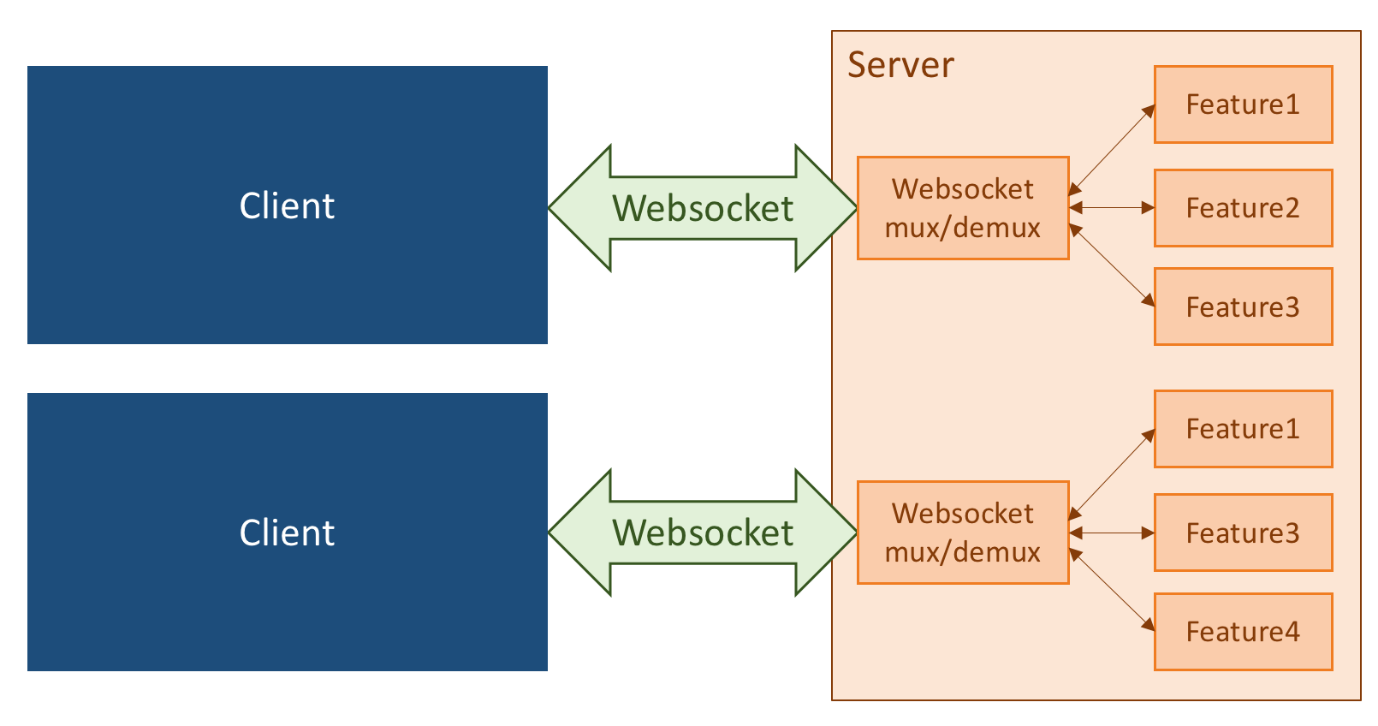
So a more precise diagram for the server, stripping away the edges, is as below.



As noted in the picture though, the demux description still isn’t exactly correct – we will make this fully accurate after talking about client modularity.

Any running feature is also free to send notifications (from right to left in this picture) whenever it likes, not simply in response to incoming requests. The server runs on the Python3 asyncio framework, so features have full access to other events whenever they happen.

If there are multiple clients running (either from multiple human beings using with the same app backend, or a single human being with multiple tabs or browser windows open) then each client has its own websocket, and its own set of feature instances, as per the following diagram. So one instance of a Feature class is responsible for serving exactly one client.



## Client modularity

There is also of course a modular structure within the client. However, a symmetric demuxing there on nfn\_type is not sufficient, since multiple different client modules can invoke the same backend service, for which all the reply notifications would have the same nfn\_type.

So each request also includes a **channel** string field, denoting the sub-module within the client that originated the request. The server reflects back the channel field in every reply notification, and the client uses this to demux that notification back to the originating sub-module. So the cli\_exec RPC example was not strictly accurate, and more precisely looks like, eg:

[C🡪S] ⟨req\_type: "cli\_exec", id: "42", channel: "exec" , command: ”show clock”⟩

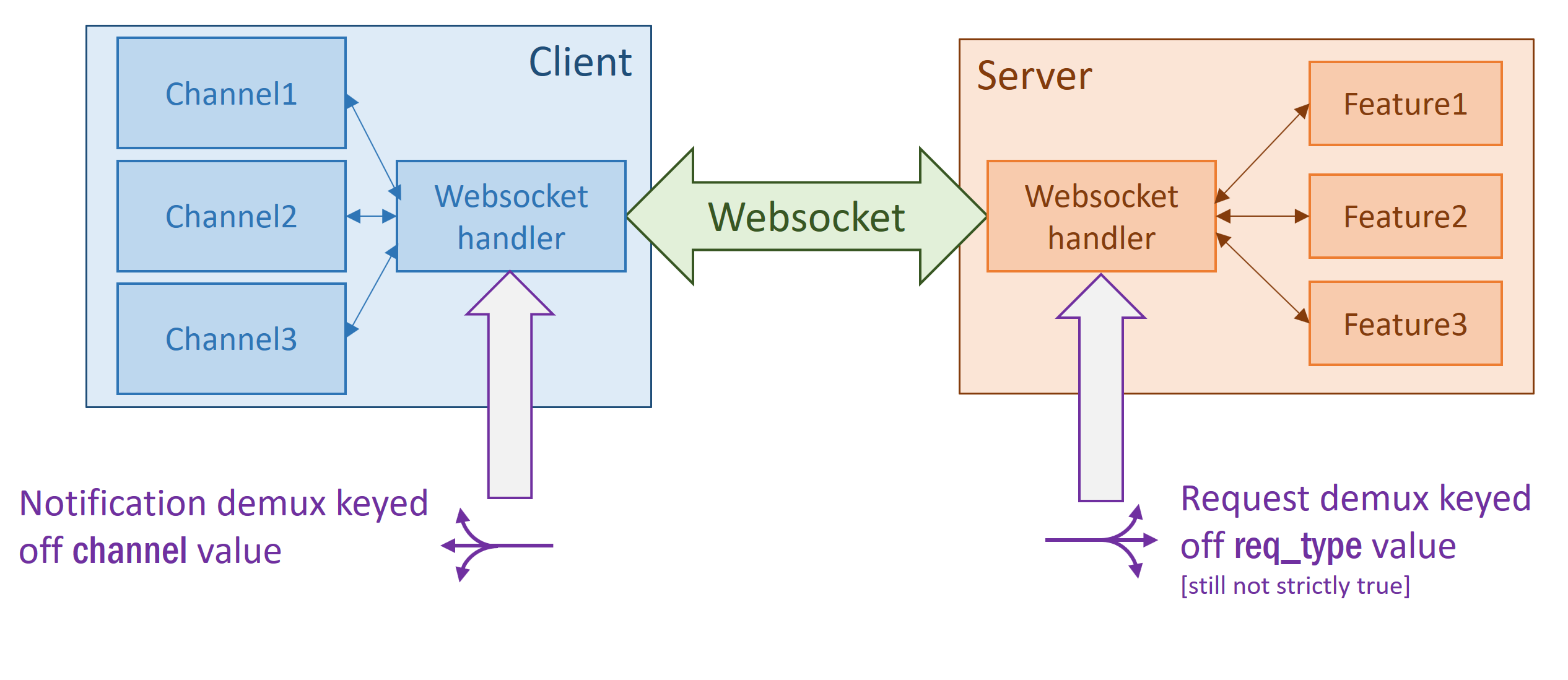
[S🡨C] ⟨nfn\_type: "cli\_exec", id: "42", channel: "exec" , result: "10:20:30 UTC Jan 10 2010"⟩

There is one “magic pseudo-channel” called “error”. This is used by the server to send global error messages to the client, for which it can’t know a specific channel. For example, if the client sends a websocket request that is invalid JSON. In such cases, the server simply sends an unsolicited notification with

[S🡨C] ⟨nfn\_type: "error", channel: "error" , value: "Some useful error message"⟩

These are expected to be rare, and the client can just display these in some globally-visible form.

So yet a more precise diagram (but still not quite correct yet – we’ll fix that in just a moment) is as follows.

In code terms, the EnTrance.Channel module explains how clients create and use channels, and EnTrance.Request explains how to construct and send requests.

## Feature lifecycle

Each feature for a single-user app is either a **configured feature** or a **dynamic feature**. A configured feature provides a global service, and has exactly one instance is created for each websocket. For this reason, configured features must avoid setup that is expensive or has side-effects (eg ssh into a router). Rather, default features should start up cheaply, and serve their requests statelessly. Example default features include:

* The “persistence” feature allows the client to store and retrieve data in a server-side file
* The “core” feature provides functions like server liveness checks, server restarts, and starting dynamic features

Such features are called "configured" because there is a configuration file (by default config.yml) that specifies them for each app server; that way, potentially unsafe configured features cannot be accidentally exposed to clients. This file also allows configuration parameters to be passed into the feature. For example, the "persistence" feature specifies its config schema like this:

config = {'filename': 'persist.json'}

specifying it accepts one configuration parameter called filename with default value "persist.json". This can be overridden in the application's config.yml. For example:

features:

persist:

filename: preferences.json

both enables the persistence feature for this application, and overrides this parameter.

If your application wants to execute code that runs unconditionally once the server process has its event loop established (rather than executing once per websocket, as for a configured feature) then the main function in entrance.\_\_main\_\_ takes an optional task parameter, that lets you do this.

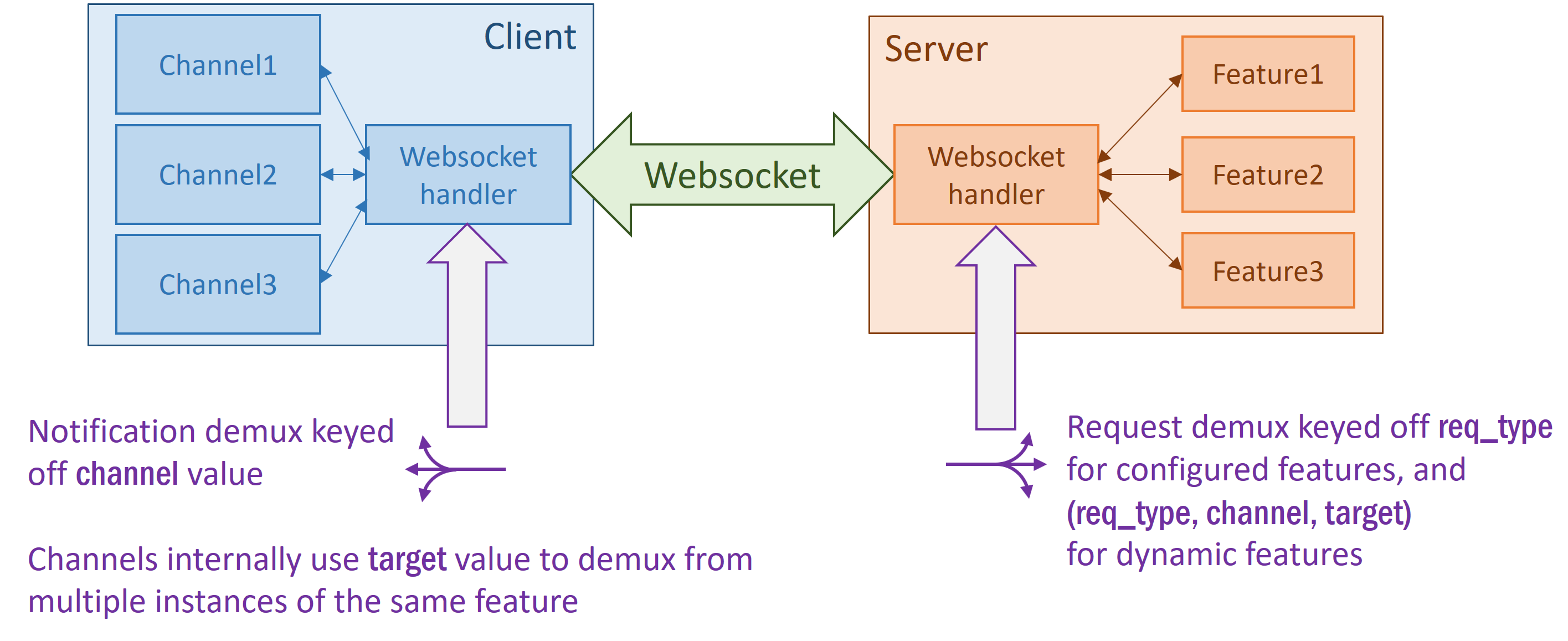
By contrast, dynamic features are not started automatically when a websocket connects. Instead, an instance is instantiated only if the client requests it, using the start\_feature request (that is itself handled by the core feature). This enables the server to present a buffet of many potentially useful features, from which the client can select what it wants, without worrying that unwanted features are incurring costs or side-effects. It also means dynamic feature instances can be started more than once. (There is also a corresponding stop\_feature request type.)

One common type of dynamic feature is a **target features**. These are features that maintain one or more outbound connections (eg to a router). The section on connection handling has more details on these. A subset of the class hierarchy is as follows (grey is abstract, coloured is concrete):

The start\_feature request also requires a target string parameter; this enables the same channel to instantiate more than one instance of the same dynamic feature (each with a different target value). All requests to the dynamic feature must also include the same target parameter (to specify which instance receives the request) and all corresponding notifications to the client also include the target parameter (so the client knows which instance sent the notification).

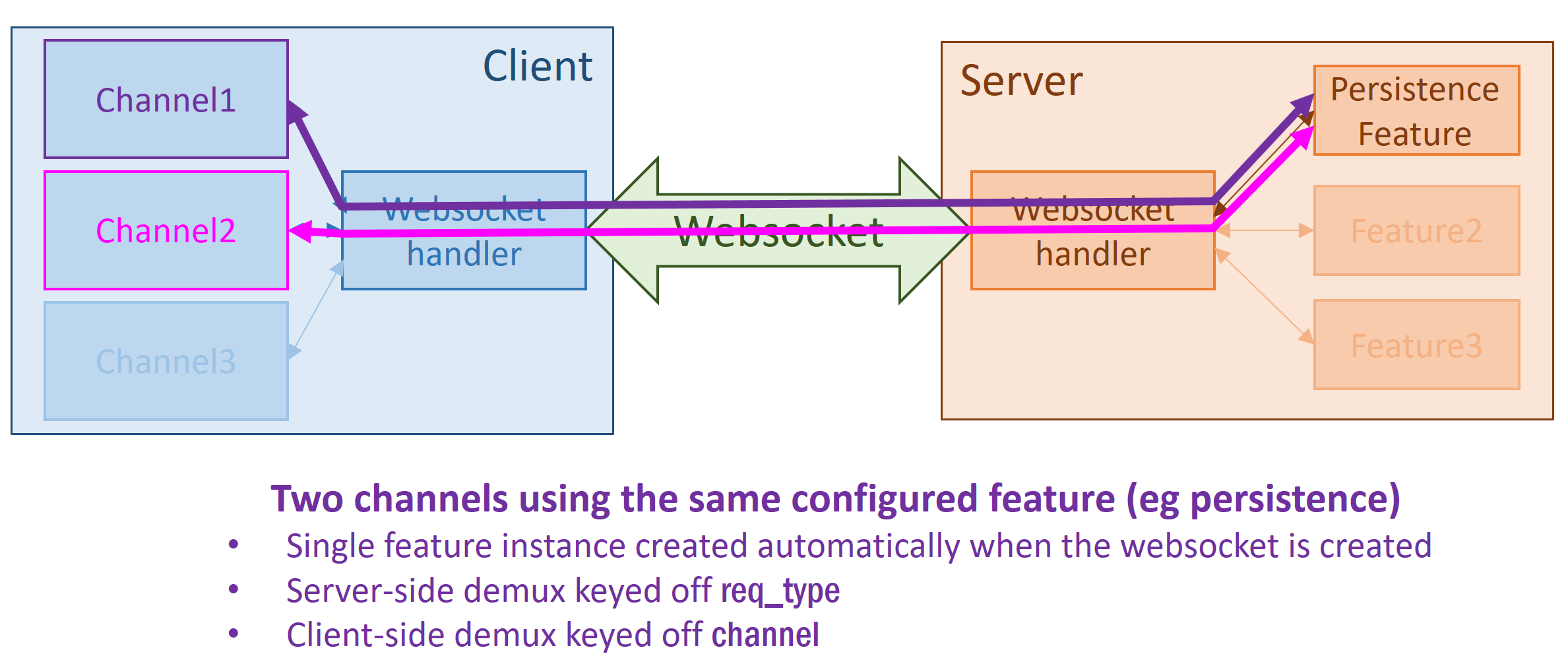
If the channel needs only a single instance, then it simply needs to provide some static default target value to start\_feature and each subsequent request to that feature. It can ignore the reflection of that same target value in subsequent notifications. The Elm package does this for all target feature requests.

We can now finally make the modularity diagram fully correct:

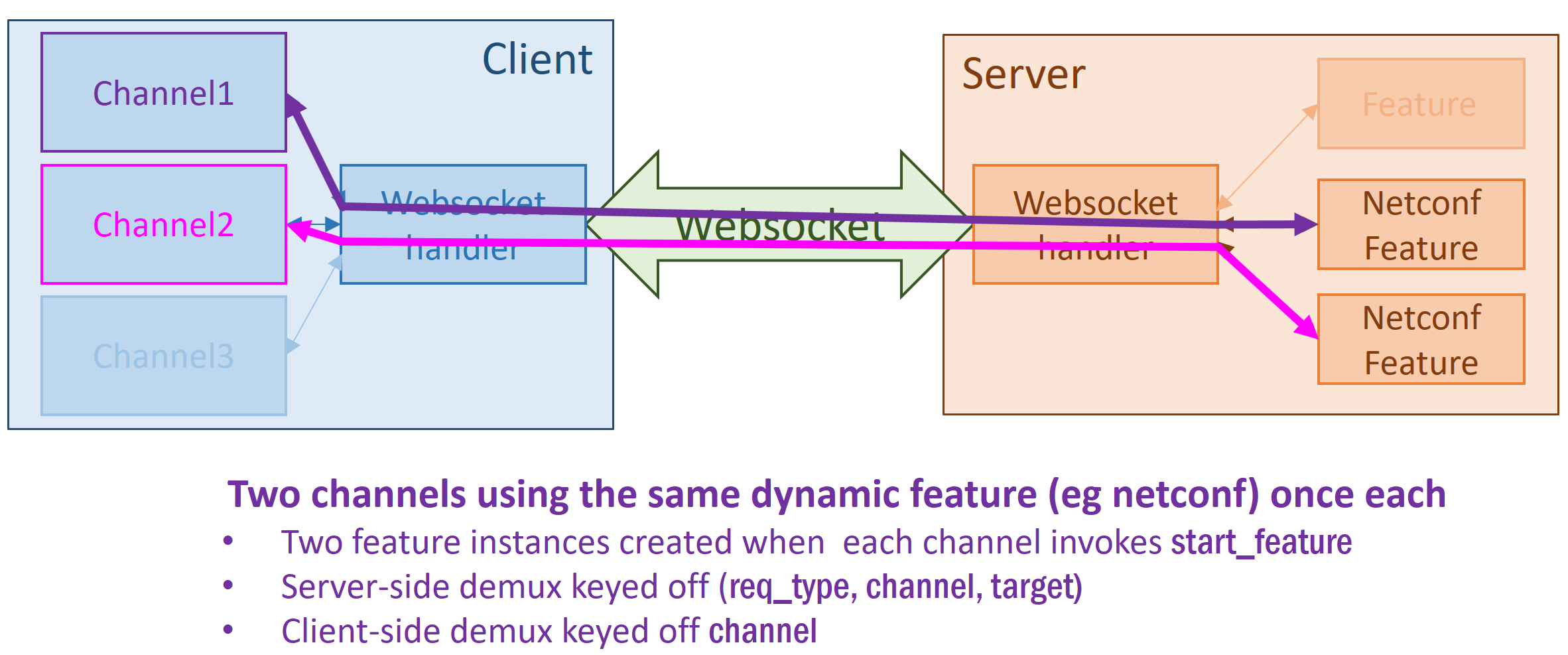


So a single channel can get multiple instances of the same dynamic feature (typically so they can do the same operation to multiple target routers), and orthogonally to that, multiple channels can each instantiate the same dynamic feature without knowing or caring about each other. This second case means the server-side demux has to take the channel into account when routing requests to the right feature instance.

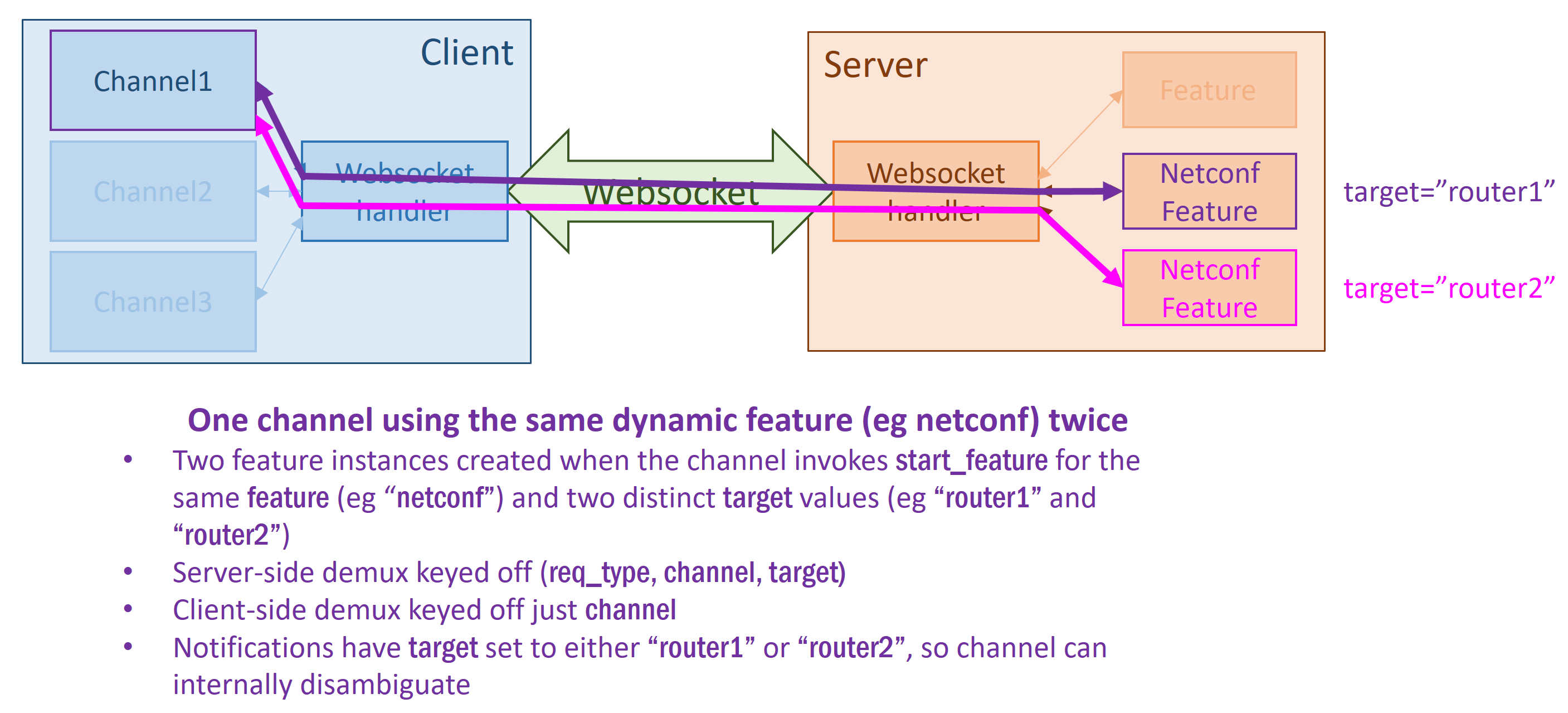
To spell this out more concretely, this is what it looks like when two features use the same default feature (eg persistence), for which a single instance is created automatically when the websocket is created:



and this is what it looks like when the same two channels use the same optional feature (eg netconf) that they each started independently once:



Finally, here is a single channel using two instances of the same optional feature by specifying two different target values:



If two channels really want to use the same dynamic feature instance, then they have to either merge, or communicate with each other. An channel is not visible in the UI (it doesn’t correspond to a tab or page, just a means of organising the client-side code) so merging channels doesn’t mean any user-visible changes.

The point of all this is really threefold:

* The app developer can simply focus on whatever individual module they are currently working on (either a client-side module with a channel, or a server-side feature) and not have to worry about interaction combinations, since something sensible should always happen.
* The “entrance” Python package can grow over time to include a wide variety of optional feature classes, without affecting any individual app (each in its own separate repo).
* An app that only talks to a single target entity (eg one single router) shouldn’t be burdened with complexity, but can easily evolve to talking to multiple target entities later if it needs to.

# Connection handling

## Introduction

Applications that involve outbound connections (eg to routers) present some particular challenges. Connections take time to establish, and can stall or fail for different reasons (eg issues with the target, or hitting an exception in the feature code on the server, thus triggering an automatic reconnect). The functionality of the app typically depends on all connections working correctly.

So more precisely, there are two sets of issues that connections raise:

* It can be complex for the app developer to handle all the different failure scenarios correctly
* It can be confusing for the user to understand what’s going on, and whether they just need to wait a few seconds or take some remedial action (such as triggering a reconnection attempt)

EnTrance comes with a suite of functionality to help address both these sets of issues. This is somewhat intricate, with the aim of controlling the inherent complexity in the infrastructure; then each app is liberated to concentrate on its specific task at hand, while still being robust and simple to use.

The informal goal is to make it easy to write an app that removes the most likely 95% of bugs or confusions. There will still remain some long tail of rarer issues that may not be handled perfectly (eg a user clicking on a button at the same moment that a connection fails) but that is deemed tolerable.

## Concepts and terminology

A **target** is an entity (such as a router) to which the server can initiate a protocol session over a particular transport. (It is not coincidental that this is the term also used to disambiguate multiple instances of the same optional feature, when they are created by the same channel.)

Typically a single router is represented by a single target, but it is possible for one router to correspond to multiple targets if it is accessed via multiple connection protocols that are totally unrelated (eg ssh and gRPC).

A **connection** is a single protocol session from a server feature to a target. Each connection has a name (that is unique across all connections to a single target) and is always in exactly one of the following states:

1. Disconnected
2. Connected
3. Failure while disconnecting [plus error message]
4. Finalizing (ie protocol established, but doing further initializing, so not yet usable)
5. Connecting
6. Disconnecting
7. Reconnecting\_after\_failure [plus error message]
8. Failed to connect [plus error message]

The order is important, since a set of connections can have an aggregate state, defined by the maximum number of all the connections in the set. This aggregate state presents an abbreviated view of the state of all those connections, that can be presented to the user in a simplified UI. If all is well then the aggregate state will either be disconnected (we don’t need any sessions to be established) or connected (all sessions are fully up). Any other combination results in an indication of what’s wrong – either a transient condition, or a more permanent failure that might need manual attention.

A **connection factory** is an object in the server that knows how to produce connections to a single target. Currently the only connection factory class is SSHConnectionFactory, that is supplied with things like an IP address and ssh credentials when it is created, and can be used to create either console or netconf sessions to the specified target, using ssh transport. In the future there may be other connection factories (eg for gRPC, or direct connections into certain virtual routers).

Finally, a **target feature** is an optional feature that requires one or more connections to the same target. These are described in more detail next.

## Target features

There are two types of target features: those that directly manage one or more connections to a target, and those that aggregate other target features.

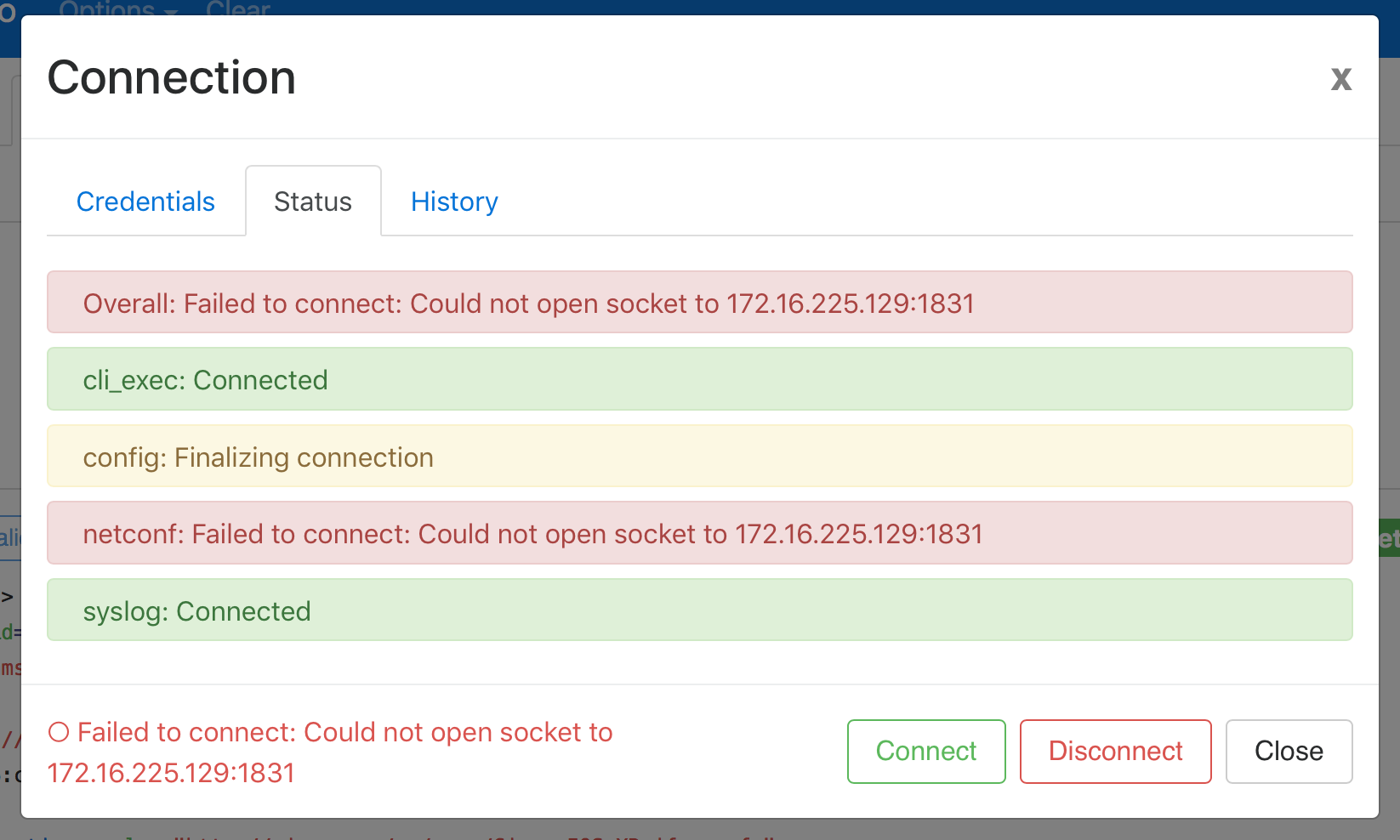
For example, suppose a network management application requires a Netconf and a Syslog connection to each router under management, and these are grouped in a hierarchy:

Each of these boxes is a target feature. The orange boxes are instances of the **target group feature**, whose children are other target features. The blue boxes are other instances of target features, whose children are actual connections to targets.

Each target feature maintains an aggregate state, based on the state of its children (whether those children are direct connections, other target features, or a combination). The aggregate state is computed from the state of all children via connection state ordering (see previous section). So a possible set of states is as follows:

Here green means "connected", orange means "finalizing connection", and red means "failed to connect". The Netconf/Syslog states are determined by the actual connections, and they ripple up according to the connection state ordering in the previous section. So this enables a user interface that lets the end user get an appropriate summary of the connection health at whatever level of detail they are interested in.

For example, below is a snapshot of a demo app showing the connections to a single target. This shows how the overall state (that is visible in more places than this detailed per-connection status tab) indicates the problem the user probably has to care about the most for that target (in this case a misconfigured netconf port):



Connection state notifications for any given target feature are requested by the client by supplying a ⟨con\_state\_subscribe: true⟩ argument to the start\_feature request. The target feature sends a connection\_state notification whenever a child object (either connection or child target feature) changes state. This notification includes:

* The name of the child object that caused the state change, and its new state
* The overall state of the target feature
* A boolean summary of the target feature state
* A timestamp

The boolean summary (basically, is this target functional or not) enables the user interface to disable functionality that won't work yet (eg greying out a button) in a simple way.

A target group feature's children are determined by its target value. If the client creates a Netconf feature, a Syslog feature, and a target group feature, all with a target value of "Core1", then they are automatically associated into a hierarchy. If the target group feature has a parent\_target name of "Core routers", then it will automatically become a child of another target group feature whose target value is "Core routers". And so on – the client can declaratively assemble a hierarchy however it likes, in any order.

# Multi-user apps

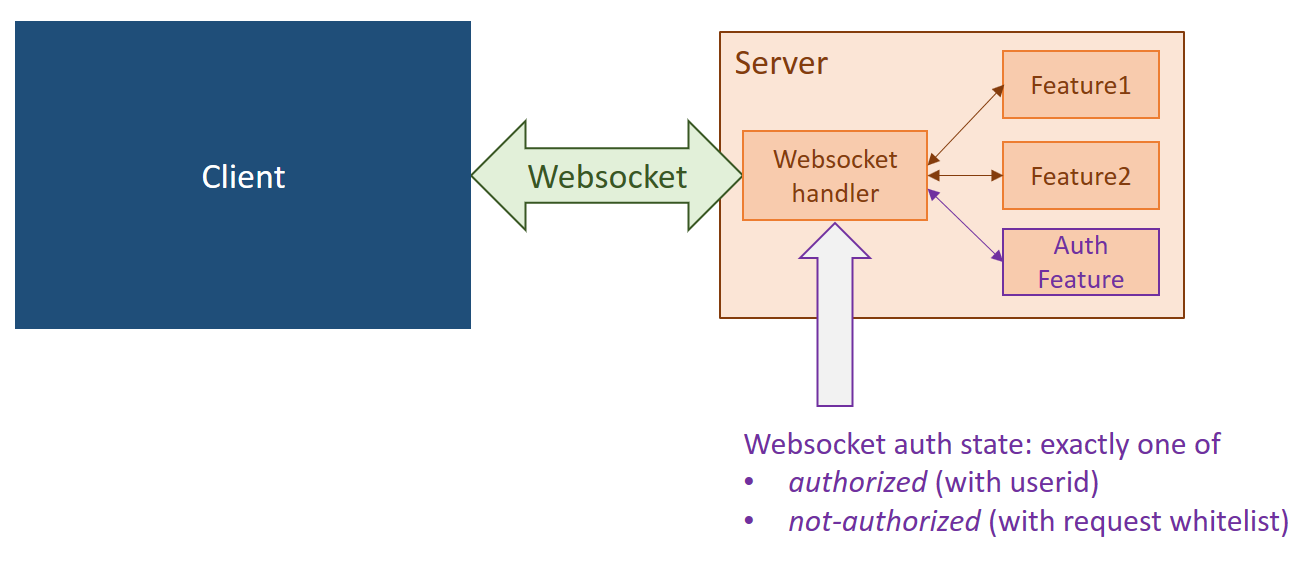
*Note – the functionality in this section has not been implemented at the time of writing.*

## Introduction

The application designs so far are suitable for apps that require no client authentication. For those that do (either to serve a single authenticated user, or multiple users) some extensions are required. All such cases are termed “multi-user” here, since even a single authenticated user represents a distinction between the intended user and others.

## Auth Features

Multi-user apps make use of a special type of configured feature: **auth features**. Auth features are the only ones that set the state of the server websocket handler to be either *authorized* or *not-authorized*:



In the absence of an auth feature, the state is always *authorized*, and the userid parameter is always “default”. In authorized state, the mux/demux adds a ⟨userid: <userid>⟩ field to every client request, before dispatching to the owning feature. If there is an auth feature configured, then this is specified once the client is successfully authorized. The userid field is then a reliable indicator of user identity for any features that want to make use of that (eg for storing per-user preferences).

In *non-authorized* mode, the mux/demux checks each request type against a whitelist provided by the auth feature (typically just ping and login). A request not matching the whitelist is dropped, and a notification is returned to the client (ie a notification with nfn\_type set to the request’s req\_type, channel/id set to the request values) with an error string explaining that the user is logged out. A login request is handled by the auth feature like any other request, and either solicits an error message, or a successful transition from *non-authorized* to *authorized* state. A subsequent logout request transitions back to *non-authorized* state.

For now a login request consists simply of userid and password parameters, that are checked by the configured auth subclass against LDAP, local unix authentication, or similar, as specified in the configuration file. This can adapt to other authentication mechanisms like OAuth in the future.

An authorization feature also sends a login\_required notification to the client both on websocket connect and transition from *authorized* to *non-authorized* state, in order to prompt the client to present an appropriate UI to the user.

## Security models

When the user is authorized, one might imagine handing over a cryptographic token that has to be used in subsequent requests (either just reflected in plaintext, or used as part of a cryptographic digest for each request). This reflects some common web paradigms (starting from the humble cookie) but these are intended for a model where the client makes HTTP requests to many servers, not just the one that originally authorized it; hence the client needs some portable attestation to its authorization.

In EnTrance, by contrast, everything happens over the single websocket. So the client has no need of a portable attestation of authorization. There are theoretical threat models that might be mitigated this way that involve TCP spoofing, but these are both unlikely, and already addressed if the client-server connection is secured by TLS (which is obviously best practice if passwords are going over the wire).

So the authorization state is simply a per-websocket boolean maintained by server, and has no further impact on the messaging format.

# Appendix – Feature class hierarchy

