Interaction Systems Group (REACT)

Master semester project

graasp-websockets

Real-time content change notifications in a web-based service
using a full-duplex communication channel

Alexandre CHAU

supervised by
Dr. Denis Gillet
Dr. Nicolas Macris
André Nogueira

June 11, 2021

Contents

1 Abstract

Graasp is a social media learning platform that provides an ecosystem of software applications for digital education. It is developed by the Coordination & Interaction Systems Group (REACT) at EPFL.

In late 2019, a major overhaul of the *Graasp* web software stack was initiated using state-of-the-art technologies. This modernization allows for the adoption of recent open standards such as WebSocket, which in turn enables the implementation of new classes of features including real-time bidirectional communications between clients and server.

This project provides real-time content change notifications in *Graasp* web-based services using the WebSocket protocol as a full-duplex communication channel.

Keywords: WebSocket, Digital Education, Graasp Platform

2 Introduction

Graasp provides a social media platform for teachers and students in primary, secondary school and higher education that supports collaborative learning, personal learning and inquiry learning. It aims at providing digital tools for education that resemble social media services which feature web-based applications, user-generated content, service-specific profiles and social networks: "the need to provide a versatile and agile social media platform strengthening digital education has emerged. The main features of such a platform mimicking the social media solutions people are using daily include the ability (i) to create online spaces targeting dedicated learning activities by aggregating resources from various cloud or local sources, (ii) to share these spaces with peers (either teachers or students), (iii) to exploit these spaces by interacting with the integrated resources, available tutors, and collaborating peers, as well as (iv) to archive these spaces together with the learning outcome (produced artifacts) for later use or as proof of personal achievements in lifelong learning." (Gillet et al. 2016)

Starting in 2019, the *Graasp* web software was rewritten to refactor and consolidate its codebase (thereafter referred to as *Graasp v3*) with modern technologies, both on the back-end server side (with tools such as the NodeJS runtime and the Fastify web framework) and in front-end user applications (with e.g. the ReactJS user interface library and ECMAScript 2015). Browser vendors and open standards organizations (e.g. W3C, IETF) also evolved web browsers and Internet protocols into intercompatible systems based on open standards with richer feature sets. In particular the WebSocket protocol was introduced which "enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code." (RFC 6455)

This allows web applications running in client browsers to keep interacting with (and receiving data from) the server even after the initial request was answered, and without the need for the client to initiate another transaction. In the context of *Graasp*, new features can be developed using this technique. In particular, we are interested in providing real-time notifications that reflect state changes that occured on the server (for instance a document created by a user) directly to all affected clients (such as another user that is currently watching the folder in which the document was created). Real-time notifications can also inform users when unexpected errors occur (e.g. a database server crashing) immediately. Otherwise, the client user would have to wait until he/she performs another action (refreshing the page, clicking on a button) to trigger a new request and then obtain the updated response / error.

WebSocket is thus particularly well-suited for that purpose, as it allows the server to push such notifications directly to connected clients. Other suboptimal techniques (e.g. polling) could also be used, but they consume network resources even when no new state is available.

2.1 Previous work

Preliminary prototypes for real-time content changes were implemented in *Graasp* using Socket.IO, an event-based communication library built on top of WebSocket and HTTP long polling. However, several limitations eventually led to Socket.IO being discarded:

- **Lack of compatibility**: Socket.IO implements a non-standardized layer of abstraction above WebSocket. As such, clients cannot connect to a Socket.IO server without the corresponding client Socket.IO library, and Socket.IO is not directly compatible with native WebSockets. A client implementation of Socket.IO must exist for every existing platform; even though Java, C++, Swift, Dart, Python and .Net versions are available, this limitation would prevent the Graasp ecosystem to grow outside of these languages.
- **Lack of integration with Fastify**: Authentication through Fastify on Socket.IO resources (such as “rooms”) proved to be difficult. In particular, cookie exchanges led to unexpected behaviours.
- **Library size**: The server library of Socket.IO weighs 1.03 MB (npm), while the minified client library has a size of 62.77 KB (cdnjs). By using native WebSockets from the client side instead, no additional library is required or downloaded.

Hence the notification delivery system in *Graasp v3* requires a domain-specific real-time protocol based on native WebSockets.

2.2 Contributions

This project provides a WebSocket-based solution for real-time delivery of content to specific clients, which can be plugged into the core *Graasp* server code through a single function call. The code is available as an open-source repository on Github (graasp/graasp-websockets) with an associated toolchain and continuous integration running on Github Actions. In addition, a client-side API is provided for a sample front-end application (graasp-compose). The code is modularized such that independent components can be reused for different purposes (even unrelated to *Graasp*).

3 Implementation

3.1 Requirements

The purpose of this project is to abstract global broadcast (i.e. server to all clients) and channel broadcast (i.e. server to specific groups of clients given some identifer) operations over an entire distributed network of computers (with any number of clients and servers) using only point-to-point websocket connections between each client and server.

To do so, we define a custom, domain-specific protocol for messages that are exchanged over WebSocket. We provide additional data structures and logic to handle channels, stateful connections, garbage collection, and load distribution over many servers.

3.2 Back-end: graasp-websockets

3.2.1 Tooling

The *Graasp v3* rewrite is based around a core server that handles the essential abstractions required in the *Graasp* applications ecosystem (e.g. users, or items which represent learning resources). The *Graasp* core is written in Typescript, runs on the NodeJS runtime, and uses the Fastify web framework to serve client requests over an HTTP JSON REST API.

`graasp-websockets` depends on the following libraries and technologies:

- NodeJS: a standalone Javascript runtime environment for multipurpose computing
- Typescript: a high-level language that extends Javascript with types
- The *Graasp v3* core Fastify server instance (graasp/graasp)
- ws: a low-level WebSocket server library and the fastify-websocket adapter
- AJV: a JSON validation library
- Redis: an in-memory data structures server used to provide state replication across multiple *Graasp* core instances (when distributing load) and the ioredis client library
- Jest: a Javascript testing library and ts-jest a Jest transformer for Typescript

3.2.2 System architecture

Fastify instances can be extended through plugins. A plugin is a function that takes the Fastify server instance as input and registers any additional behavior or state on the instance as a side-effect. This allows a clean separation of concerns, as any additional related group of features in *Graasp* can be added as another plugin.

`graasp-websockets` is implemented as a Fastify plugin that interacts with services attached to the core server instance that can provide real-time updates, such as the items service that handle persistence changes in resources storage.

The plugin is composed of several independent modules. They are reflected in the files hierarchy of the `src/` folder:

- **service-api.ts**: the entry point that exports the plugin function. It defines the HTTP routes on which clients can upgrade their connections to WebSocket, performs authentication, validation and deserialization on the incoming requests (HTTP and WebSocket),

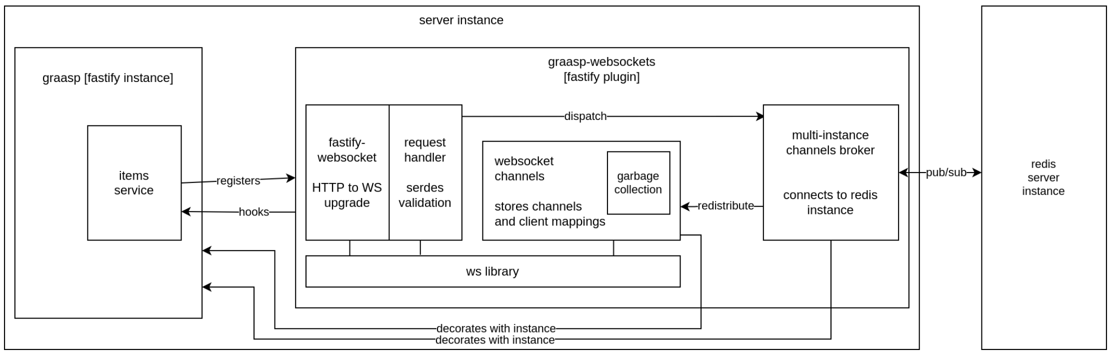


Figure 1: Block diagram of the architecture of graasp-websockets

instantiates the other modules and registers *Graasp*-specific interactions with WebSocket. For instance, it dispatches the events from the items service to related clients through WebSocket.

- **ws-channels.ts**: it defines the data structures that represent a WebSocket client, as well as channels which represent *rooms* or *topics* that clients can *subscribe to*. Messages sent (*published*) on a given channel are received by all clients subscribed to it, and none others. Clients can subscribe to specific channels by sending WebSocket request messages to the service API. This module implements the WebSocket channels management abstraction, which allows the creation and deletion of channels, the dispatch of messages over specific channels, and also provides functions for the clients to (un-)subscribe (from) to given channels through the service API. It also implements garbage collection as a heartbeat mechanism: since barebone WebSockets do not detect broken connections, the channels abstraction would otherwise always leak memory and keep accepting new connections until the server runs out of memory. Instead, it periodically sends ping messages to clients (as described in the WebSocket specification) and destroys the instances of the ones that do not reply. Similarly, it can remove empty channels as a pessimistic optimization, which ensure that only clients and channels that currently exist are being used and allowed to occupy server resources. The relationship between clients and channels is kept in 2 synchronized maps for performance: one maps each channel to its client subscribers, and the other each client to its channel subscriptions.

- **multi-instance.ts**: this module allows multiple instances of the *Graasp v3* core to co-exist while maintaining a global publish-subscribe relationship between any server and client pair. This is useful when multiple copies of the server run in parallel (for instance for load balancing purposes). It provides a message format and serialization / deserialization for Redis, and uses the Redis Pub/Sub interface to relay message across instances. It thus exposes a message broker for the Fastify instance to dispatch a message among all servers. Since each instance is also subscribed to the same Redis channel itself, modules that use this message broker do not need to send WebSocket messages through the channels abstraction directly: they can send it through the broker, which will receive it again itself, and redistribute it in the corresponding channel through the channels abstraction to its own clients. As a corollary, channels thus do not need to exist ahead-of-time, and they do not need to exist on all instances simultaneously. Only instances that have active subscriptions to the given channel will hold its representation in memory, which can be created on-the-fly when the first client requests the given channel, and be destroyed as soon as none of its own clients are subscribed to it anymore, irrespective of other instances and their connected clients.

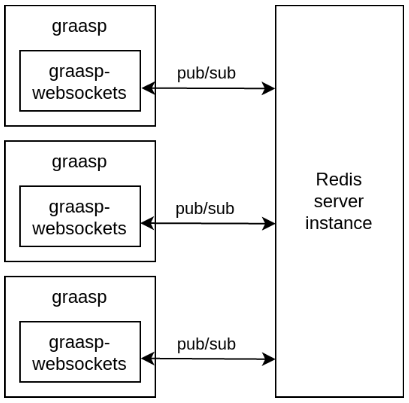


Figure 2: Multiple instances of graasp with the websockets plugin communicating through Redis

- **interfaces/message.ts** and **schemas/message-schema.ts**: these interfaces describe the shape of all possible messages exchanged between the server and its clients. In sum, they specify the message format of the application-specific protocol that is used over WebSocket. For instance, clients may send "subscribe" messages as specified in the `ClientSubscribe` message definition. Schemas describe these exact same shapes when serialized into JSON using JSON Type Definitions (JTD).
- **impls/message-serializer.ts**: the above equivalent definitions allow the AJV library to compile efficient parsers, validators and serializers for the corresponding object types, which are more performant than generic solutions such as `JSON.parse`.

To demonstrate that modules are indeed independent, the repository provides an example of an application that only uses the WebSocket channels abstraction that showcases a real-time chat room (a feature currently unrelated to *Graasp*).

3.3 Front-end: graasp-compose

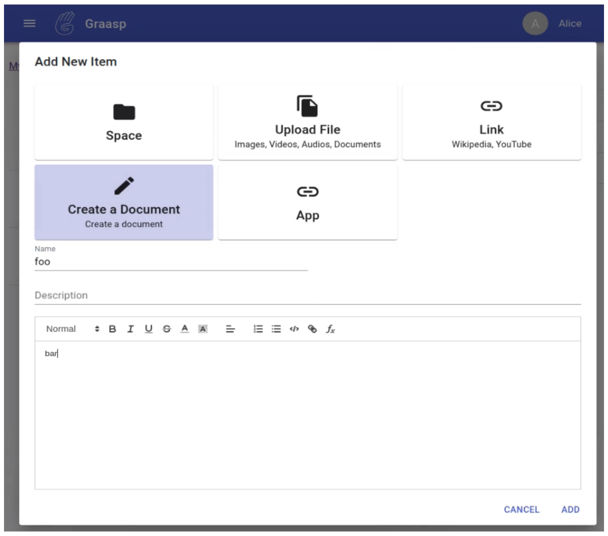
The *Graasp v3* rewrite also includes a rework of the front-end ecosystem. To demonstrate use cases of the `graasp-websockets` extension, we implement real-time interactions in `graasp-compose`, the new web application to manage learning content in *Graasp*. `graasp-compose` allows user to create content by uploading files, writing documents, linking external resources with URLs, integrating other *Graasp labs* and applications, . . .

`graasp-compose` is written in ReactJS, a JavaScript library for building user interfaces following a reactive, declarative MVVM paradigm. It also uses react-query, a state synchronization and caching library for React applications.

We implement several server state changes that are pushed in real-time (without any page reload or user interaction) to client interfaces:

- when a user *shares* the ownership of an item with another user, the recipient sees the new item immediately in his/her "Shared with me" list

- when an item is created in a *space*, other users that are currently browsing that *space* see the new item appear inside immediately
- when an item is deleted in a *space*, other users that are currently browsing that *space* see the old item disappear inside immediately



Name
foo

bar

Figure 3: A screenshot of graasp-compose (in development)

3.3.1 Architecture

The WebSocket extensions in the front-end are implemented in 3 parts:

- The *websocket notification API* (`src/api/notif.js`) implements a thin wrapper around the native WebSocket connection available in web browsers. It handles the protocol previously defined by the back-end service API and keeps stateful bindings between channel subscriptions and code handlers that will modify the UI in response to channel events.
- The *custom hooks* define React hooks as functions that perform side-effects: they are bound to the lifecycle of the UI components wherever they are called. They use the websocket notification API defined above to allow components to subscribe to updates of a specific channel, and use the `react-query` query client to mutate the cached data previously received from the server. For instance `useSharedItemsUpdates` will subscribe to the channel of the items shared with the current user, and will force a cache flush of the locally stored shared items from the current session if new items are shared with him/her.
- The above custom hooks are inserted into the *React UI components tree* so that displayed elements that should change with updates can re-render when new state is available. For instance, the `useSharedItemsUpdates` hook is called in `src/components/SharedItems.js` which describes the UI view of the list of shared items of the current user.

In the shared items example, the `SharedItems` view subscribes to the corresponding channel for the current user when the view is mounted into the webpage. When a new notification is sent from the server, the websocket notification API calls back the corresponding custom hook

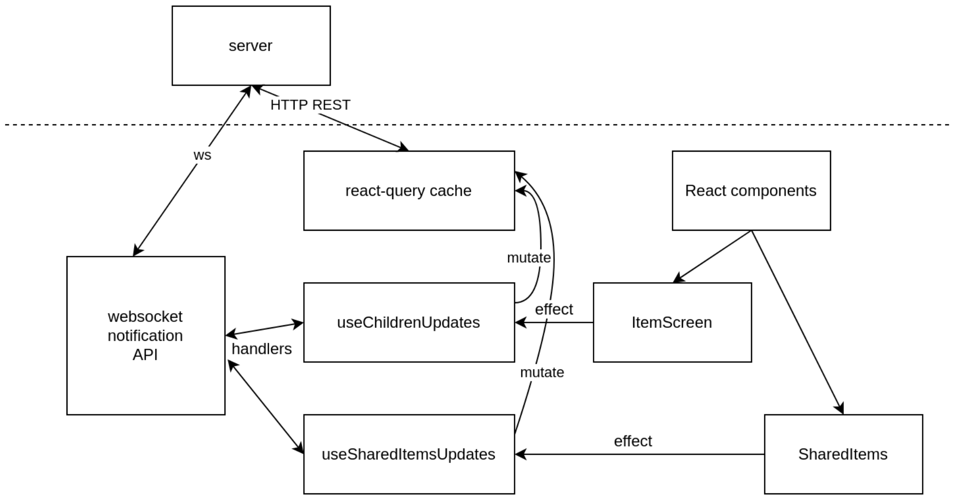
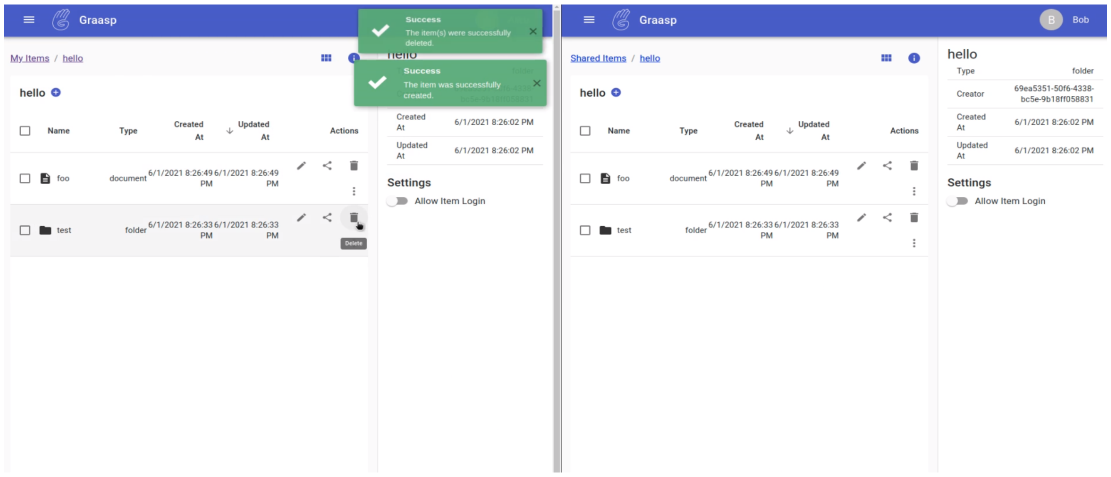


Figure 4: Block diagram of the WebSocket architecture in graasp-compose

handler. This updates the cache of the shared items with new items from the server, which in turn triggers a re-render of the view through the React component lifecycle and which finally displays the updated list of shared items to the user. Similar implementations are written for the `ItemScreen` component which displays the children elements of a space.



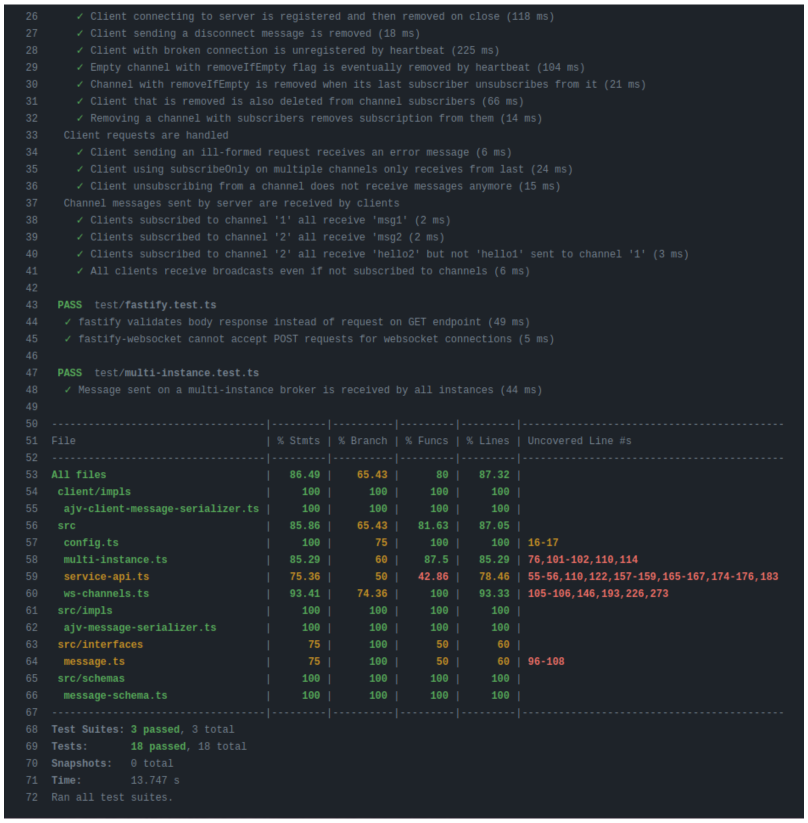
Name	Type	Created At	Updated At	Actions
foo	document	6/1/2021 8:26:49 PM	6/1/2021 8:26:49 PM	
test	folder	6/1/2021 8:26:33 PM	6/1/2021 8:26:33 PM	

Type	folder
Creator	69ea5351-50f6-4338-bc5e-9b18ff058831
Created At	6/1/2021 8:26:02 PM
Updated At	6/1/2021 8:26:02 PM

Figure 5: Screenshot of a live demo of graasp-websockets: on the left, items are being deleted by the user "Alice." She receives visual feedback on the top-right corner that the items are successfully deleted on the server as well. On the right, Bob also had the same elements removed, as evidenced by the same number of children items on the right.

4 Testing

The codebase is tested with a fairly extensive test suite of unit tests as well as end-to-end tests using the Jest framework. Code coverage of the tests is also tracked using the `istanbul` tool integrated in Jest. The code is built and tested using continuous integration (CI) on a fresh environment using Github Actions at every push to the repository. Code lint issues are checked as well, and mocks are implemented for injected services (such as database, task manager).



```
    ✓ Client connecting to server is registered and then removed on close (118 ms)
    ✓ Client sending a disconnect message is removed (18 ms)
    ✓ Client with broken connection is unregistered by heartbeat (225 ms)
    ✓ Empty channel with removeIfEmpty flag is eventually removed by heartbeat (104 ms)
    ✓ Channel with removeIfEmpty is removed when its last subscriber unsubscribes from it (21 ms)
    ✓ Client that is removed is also deleted from channel subscribers (66 ms)
    ✓ Removing a channel with subscribers removes subscription from them (14 ms)
  Client requests are handled
    ✓ Client sending an ill-formed request receives an error message (6 ms)
    ✓ Client using subscribeOnly on multiple channels only receives from last (24 ms)
    ✓ Client unsubscribing from a channel does not receive messages anymore (15 ms)
  Channel messages sent by server are received by clients
    ✓ Clients subscribed to channel '1' all receive 'msg1' (2 ms)
    ✓ Clients subscribed to channel '2' all receive 'msg2 (2 ms)
    ✓ Clients subscribed to channel '2' all receive 'hello2' but not 'hello1' sent to channel '1' (3 ms)
    ✓ All clients receive broadcasts even if not subscribed to channels (6 ms)

 PASS  test/fastify.test.ts
  ✓ fastify validates body response instead of request on GET endpoint (49 ms)
  ✓ fastify-websocket cannot accept POST requests for websocket connections (5 ms)

 PASS  test/multi-instance.test.ts
  ✓ Message sent on a multi-instance broker is received by all instances (44 ms)

-----------------------------------|---------|----------|---------|---------|-------------------------------------------
File                               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------------|---------|----------|---------|---------|-------------------------------------------
All files                          |   86.49 |    65.43 |      80 |   87.32 |
 client/impls                      |     100 |      100 |     100 |     100 |
  ajv-client-message-serializer.ts |     100 |      100 |     100 |     100 |
 src                               |   85.86 |    65.43 |   81.63 |   87.05 |
  config.ts                        |     100 |       75 |     100 |     100 | 16-17
  multi-instance.ts                |   85.29 |       60 |    87.5 |   85.29 | 76,101-102,110,114
  service-api.ts                   |   75.36 |       50 |   42.86 |   78.46 | 55-56,110,122,157-159,165-167,174-176,183
  ws-channels.ts                   |   93.41 |    74.36 |     100 |   93.33 | 105-106,146,193,226,273
 src/impls                         |     100 |      100 |     100 |     100 |
  ajv-message-serializer.ts        |     100 |      100 |     100 |     100 |
 src/interfaces                    |      75 |      100 |      50 |      60 |
  message.ts                       |      75 |      100 |      50 |      60 | 96-108
 src/schemas                       |     100 |      100 |     100 |     100 |
  message-schema.ts                |     100 |      100 |     100 |     100 |
-----------------------------------|---------|----------|---------|---------|-------------------------------------------
Test Suites: 3 passed, 3 total
Tests:       18 passed, 18 total
Snapshots:   0 total
Time:        13.747 s
Ran all test suites.
```

Figure 6: Example successful run of continuous integration testing on Github Actions

5 Limitations

- As TypeScript is used to provide better context about the code to tools and programmers and AJV is used to perform validation and (de-)serialization of messages to program objects, every protocol-related type must be described twice: as a TypeScript structural definition, as well as a JSON Type Definition schema. There exist tools to convert from one to the other, however not all edge cases can be computed.
- TypeScript is not a silver bullet: dynamic decoration of objects at runtime (for instance using indexed accesses) prevents the compiler from distinguishing when such properties actually exist.

- As in many stateful web services, the memory that a user can consume (by subscribing to channels) is generally not bounded. This means that a malicious client could craft a DoS attack in an attempt to exhaust server instances from available memory. This is mitigated by authenticating the user and performing garbage collection of unused resources periodically, however the period interval defines the possible attack window.
- npm (the dependencies manager for the NodeJS environment) is unable to resolve private git dependencies correctly when using the `npm ci` command on Github Actions. This command should provide better guarantees about fresh testing environments, however the current CI script uses `npm install` instead which resolves these dependencies correctly.
- ts-jest does not compile Typescript sources and type definition files from dependencies in the `node_modules` folder (i.e. it considers that all imports from this folder are ready-to-use Javascript files). Specific configuration for each such import must be specified in `jest.config.js` with the `moduleNameMapper` and `transformIgnorePatterns` entries, as well as a custom compiler configuration (`tsconfig.test.json`).
- In the current implementation, each possible notification must be encoded as a message type, a corresponding back-end request handler entry, a binding to some action hook from the fastify instance properties, a front-end React hook that modifies some corresponding cache entry(-ies) and a call to that hook from some React component. Currently, a given channel can only bind to single React hook as well. The design could be improved with an additional layer of abstraction to generalize the API to more use cases.
- Directly editing the React Query cache entries is potentially error-prone, especially if asynchronous updates are received out-of-order. Instead, the React hooks could simply invalidate the cache entries and trigger a complete refetch of the corresponding data through the HTTP REST API at every notification, however this would trade stronger consistency against additional network usage (as all data would be refetched at every update, instead of incremental patches).

5.1 Future work

- More interactions and notification types will be added from user scenarios of front-end graasp applications such as `graasp-compose`
- The front-end library will be consolidated to support multiple hook handlers per channel
- Additional checks can be added against potentially malicious requests (for instance, the server instance could check against the database if the client is allowed to subscribe to any specific channel before creating it if required).

6 Conclusion

In this project, we implement a WebSocket-based content distribution system for real-time updates targeting specific groups of clients. It abstracts global and channel broadcast operations across a network of computers (composed of any positive number of clients and servers) using only point-to-point WebSocket connections between each client and server. It defines a custom domain-specific protocol for messages exchanged over WebSocket.

`graasp-websockets` is designed as a Fastify plugin that can be plugged into the core server instance, with reusable modules. It provides support for dynamically allocated channels, garbage collection, and multiple server instances by relaying notifications through Redis. It also features front-end integration examples for React-based applications.

7 References

1. Denis Gillet, Andrii Vozniuk, Maria Jesus Rodriguez Triana, and Adrian Christian Holzer. 2016. *"Agile, Versatile, and Comprehensive Social Media Platform for Creating, Sharing, Exploiting, and Archiving Personal Learning Spaces, Artifacts, and Traces"*.
http://infoscience.epfl.ch/record/221529

2. Alexey Melnikov and Ian Fette. Melnikov, Alexey, and Ian Fette. 2011. *"The WebSocket Protocol"*. Request for Comments. RFC 6455.
https://doi.org/10.17487/RFC6455

3. *Socket.IO*. Accessed on 2021-06-11.
https://socket.io/

4. *Fastify*. Accessed on 2021-06-11.
https://www.fastify.io/

5. *Redis Pub/Sub*. Accessed on 2021-06-11.
https://redis.io/topics/pubsub

6. *AJV JSON Type Definition*. Accessed on 2021-06-11.
https://ajv.js.org/json-type-definition.html

7. *React Introducing Hooks*. Accessed on 2021-06-11.
https://reactjs.org/docs/hooks-intro.html

8. *React Query QueryClient*. Accessed on 2021-06-11.
https://react-query.tanstack.com/reference/QueryClient

9. *graasp-websockets*. Accessed on 2021-06-11.
https://github.com/graasp/graasp-websockets