UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FEDERICO WASSERMAN

# hNode - multiscreen alert hub and message exchange server

Trabalho de Graduação.

Trabalho realizado em Convênio de Dupla Diplomação com o INP-Grenoble

Porto Alegre, Junho de 2013

O seguinte documento apresenta o trabalho realizado como projeto de fim de estudos na França dentro de um acordo de dupla diplomação. O objetivo foi a criação de um servidor denominado hNode para uma plataforma de mensagens estruturada chamada Hubiquitus que teve suas origens no protocolo de mensagens XMPP usado em vários programas populares de chat, como por exemplo o Google Talk.

O servidor hNode junto com Hubiquitus oferecem uma plataforma que conecta através de mensagens diferentes componentes dentro de uma rede de forma síncrona. Toda troca de mensagens dentro da plataforma é estruturada, permitindo que os diferentes componentes possam ser automatizados. O objetivo desta plataforma não é simplesmente passar mensagens textuais, mas, permitir aos componentes executar ações específicas ao receber certas mensagens.

O desenvolvimento foi realizado durante um estágio de tempo integral com duração de seis meses na empresa Novedia, localizada em Paris, França. A empresa é especializada em soluções personalizadas de software além de oferecer serviços de consultoria e mineração de dados. Hubiquitus, sendo uma plataforma que serve de base para vários projetos da empresa e de terceiros, é disponibilizado sob uma licença GPL, estando acessível na plataforma web GitHub.

Atualmente, a biblioteca está sendo utilizada em vários projetos de código tanto aberto quanto fechado. Entre outros, existem projetos da Novedia para automação de prédios cujos elevadores monitoram o seu próprio estado e informam de forma automática o técnico mais próximo em caso de problemas usando a plataforma (que suporta geolocalização e estado dos clientes). Existe também outro projeto que consiste numa página web onde cada cliente recebe informação em tempo real das emissões de televisão usando dados do twitter para analisar quais são as emissões mais comentadas.

O projeto incluiu além do desenvolvimento do servidor, a criação da especificação Hubiquitus, já que o desenvolvimento dele e do cliente ajudaram a definir a própria especificação que executariam. Este processo foi feito de forma iterativa por uma equipe que trabalhou em diferentes aspectos da plataforma, incluindo usuários que utilizavam a biblioteca para desenvolvimento dos seus próprios projetos.

O servidor hNode constrói sobre os pontos positivos de XMPP. Ele continua sendo utilizado para realizar certas tarefas de forma transparente aos clientes, já que ele tem todo um sistema extensível de autenticação, é estável e pode ser executado em ambientes distribuídos.

À diferença de XMPP que lhe deu origem, Hubiquitus tem uma estrutura rígida de mensagens definida usando JSON contra o XML usado por XMPP. Isto traz dois grandes benefícios para os objetivos da plataforma: o primeiro está relacionado ao fato de que o servidor foi escrito usando a linguagem JavaScript na plataforma NodeJS, fazendo com que toda mensagem que o servidor precise tratar fosse vista como um objeto nativo. A segunda vantagem está no fato de que usando uma estrutura melhor definida e mais rígida que a especificada por XMPP, a automação do processamento de mensagens, seja para filtragem ou para serem usadas na execução de comandos é muito mais eficiente e simples de usar.

O servidor XMPP que é utilizado como suporte pelo hNode oferece a autenticação e a distribuição de mensagens, ou seja, quando uma mensagem é publicada, é XMPP que

decide quem deve receber a mensagem. Todavia, XMPP é visto como uma plataforma de suporte que pode ser substituída a qualquer momento já que é completamente transparente para os usuários Hubiquitus. Isto quer dizer, que um usuário Hubiquitus envia suas credenciais para o hNode e não para o servidor XMPP diretamente e é ele que mantém duas conexões: uma com o cliente final que só se comunica com o hNode e outra simulando um cliente conectado à rede XMPP para ter atualizações de estado e poder enviar as mensagens.

Um dos grandes grupos de clientes da plataforma é formado por usuários móveis que têm conexões instáveis (passagem de 3G a EDGE, etc.). Isto provoca queda nas conexões com o serviço Hubiquitus que usa uma conexão bidirecional entre o cliente e o servidor. Considerando que cada cliente é *stateful*, já que hNode mantém uma conexão com o servidor XMPP para cada um deles além da conexão com o cliente, a reconexão causa sobrecarga no sistema e um atraso perceptível pelos clientes, pois autenticações completas demoram ao envolver vários pontos que podem não ser locais entre eles.

Para contornar este problema, as conexões XMPP dos clientes permanecem abertas no servidor durante um curto período uma vez que a conexão cliente é perdida de forma irregular e o mesmo pode realizar uma conexão chamada de *reattach* para recuperar uma sessão existente. Isso quer dizer que o cliente recuperará o seu antigo estado reutilizando a sua antiga conexão XMPP.

A introdução de um comando de *reattach* quando a conexão é estabelecida gera um inconveniente: é necessário introduzir uma forma relativamente segura para recuperar uma sessão sem a utilização da senha, que é desconhecida pelo hNode. O uso somente do usuário e um identificador de sessão não é suficiente já que um atacante pode ouvir uma única mensagem e obter todas as informações necessárias para roubar a sessão de outra pessoa.

Para isso, um contador é enviado com cada mensagem e deve ser enviado com o pedido de recuperação de sessão. Isso tenta garantir a autenticidade da pessoa, já que só ela deveria saber o estado atual do contador. Isto faz com que atacantes que procurem por mensagens do tipo *reattach* não possam reproduzi-las já que o contador será inválido e ele não saberá o novo estado. Porém, isto não garante completa segurança já que o atacante pode deduzir o contador a partir de mensagens antigas. Tendo isto em consideração, a funcionalidade é opcional no servidor e pode ser desativada.

A principal forma de comunicação entre os diferentes componentes ao utilizar a plataforma Hubiquitus é com o modelo *publish-subscribe*. Isto quer dizer que os clientes enviam pedidos de subscrição a canais onde informação é publicada e automaticamente recebem mensagens enviadas para esse canal. O sistema permite filtrar mensagens de interesse dentro das publicações aos quais o cliente está subscrito, usada, por exemplo, por clientes móveis que só querem receber mensagens de pessoas dentro da mesma área.

A publicação de uma mensagem é feita em duas etapas. Primeiro ela é enviada para o hNode que verifica se o cliente tem autorização para publicar no canal e a validade da formatação e posteriormente ele envia essa mensagem para o servidor XMPP para que a mensagem seja distribuída para as pessoas subscritas. Quando uma mensagem chega via XMPP para um cliente, o hNode pode processá-la antes de enviá-la ao propriamente ao

cliente final. Logo ele compara para ver se a mensagem é relevante baseado em filtros que o cliente aplicou, descartando as outras. Isto proporciona a possibilidade de usar um sistema de filtros mesmo em clientes móveis, já que o processamento está sendo feito num servidor e a mensagem nunca é enviada, evitando até mesmo o uso de banda.

A especificação prevê um sistema de envio de mensagens tipadas usando uma estrutura especial que denota um comando. Este, diferente de uma simples mensagem, não contém texto a ser lido por um usuário final, mas sim uma ação a ser executada. Eles foram principalmente criados com o intuito de enviar ações ao servidor, como por exemplo a subscrição a um canal. Contudo, como os comandos são encapsulados em mensagens e não têm nenhuma restrição no seu uso, é possível que um cliente qualquer receba um comando e execute uma ação personalizada. O hNode, por ser um servidor, oferece vários comandos para serem executados. Eles podem ser complexos ou simples: no caso de uma subscrição, um cliente envia o pedido de inscrição a um canal ao servidor e este analisará as credenciais para decidir se o pedido será aceito.

Várias outras funcionalidades foram adicionadas ao servidor para permitir os casos de uso básico de troca de mensagens dentro de uma rede, seguindo a especificação Hubiquitus. Eles são o comando de publicação e envio de mensagens, criação e a subscrição de canais , além dos comandos administrativos para se desassociar de um canal e editar os mesmos.

Todas estas funcionalidades são implementadas em forma de *plugins* para o servidor. Esses são arquivos seguindo uma estrutura que são lidos na inicialização do hNode, durante a qual registram identificadores que são usados pelos clientes quando querem invocar o comando. Isto proporciona uma plataforma extensível para futuras melhorias e para que terceiros possam adicionar as suas funcionalidades específicas relacionadas a sua lógica de negócios no servidor.

Dar aos *plugins* total liberdade de executarem código no servidor induz a um problema de segurança pois eles podem comprometer funcionalidades básicas do servidor se tiverem completo acesso e não é facilmente contornável sem limitar a utilidade dos comandos.

Neste contexto o servidor toma duas precauções: A primeira é executar o código oferecendo acesso restrito aos outros componentes, funcionando como uma caixa de areia. Como estar completamente isolado não é benéfico, o servidor providencia certas informações básicas além de métodos controlados para acessar o banco de dados. A segunda precaução é que durante o carregamento do servidor, quando os *plugins* são registrados, uma análise estática que procura construções consideradas perigosas é feita e inibe o uso dos mesmos.

O modelo *publish-subscribe* faz o fato de persistir as mensagens dentro de um canal natural, o que permite aos comandos acessá-las posteriormente. Para isso, os hNode estão conectados a um banco de dados e dependendo das opções especificadas na mensagem e no canal onde ela está sendo publicada, elas podem ser preservadas.

O banco de dados escolhido para o servidor foi mongoDB. Ele é um banco de dados NO-SQL que trabalha num modelo de documentos JSON armazenados em coleções, permitindo salvar informações sem precisar nenhuma conversão. O fato de trabalhar no modelo de documentos implica que eles não têm estrutura definida, e para uma

especificação que evolui seguidamente e que pode conter informações variáveis, facilita o seu armazenamento e permite uma adaptação entre diferentes versões.

A camada de abstração criada para o banco de dados é disponibilizada para os *plugins*. Isto possibilitou a criação de alguns comandos que permitem recuperar *logs* e estatísticas em relação às mensagens publicadas, como por exemplo, a recuperação das últimas mensagens de um dado canal ou a recuperação de todas as mensagens nos canais subscritos contendo uma frase.

Usos deste tipo de *plugins* podem ser mais personalizados seguindo a lógica de negócios da aplicação. Por exemplo, no caso da página que faz um comparativo das emissões mais populares, o carregamento inicial precisa de um estado para mostrar ao cliente até que informações em tempo real cheguem. Isto pode ser obtido pedindo as últimas mensagens publicadas no canal, assim podendo chegar num estado coerente com os clientes já conectados e podendo atualizar de forma correta quando novas mensagens chegarem.

Hubiquitus define uma estrutura chamada hChannel. Ela define o que é um canal. Ele é um objeto com um identificador usado pelos clientes para requisitar uma subscrição ou publicar nele, incluindo as permissões das contas autorizadas a publicar e se subscrever além de metadados que enriquecem todas as mensagens publicadas como por exemplo localização, etiquetas e outros.

Isso significa que para cada pedido de publicação feito por um cliente, o servidor precisa acessar essa estrutura para fazer verificações e enriquecer as mensagens antes de serem propriamente publicadas. A versão inicial desenvolvida do servidor fazia um pedido ao banco de dados por publicação pedindo o canal em questão, mas isto causa acessos IPC e, potencialmente , ao sistema de arquivos, consequentemente aumentando a latência percebida pelos usuários ao fazer publicações.

Seguindo um desenvolvimento iterativo, após uma primeira implementação que fazia pedidos ao banco de dados para recuperar cada canal, foi criada uma segunda versão que faz um pedido inicial ao banco de dados para carregar em memória os canais existentes, o que gerou um grande ganho no desempenho.

Na mesma linha, outra habilidade implementada no servidor hNode foi a possibilidade de ser executado de forma distribuída e coordenada, oferecendo balanceamento de carga e potenciais diminuições de latência já que os clientes podem estar se conectando a um servidor mais próximo geograficamente.

No entanto, a introdução desta capacidade traz à luz um problema recorrente em vários aplicativos que são executados em múltiplas instâncias: divergências nos estados das estruturas mantidas em memória.

Como todo hNode tem uma conta interna que se conecta ao servidor XMPP, ele mesmo pode estar subscrito a um canal e publicar mensagens. Em termos de sincronismo, as instâncias podem usar um canal de administração no qual elas estão subscritas para trocar mensagens. Logo, as instâncias podem reagir atualizando suas caches se uma mensagem é enviada quando um canal é alterado. Desta forma, implementando um modelo de sincronismo de caches usando o próprio modelo Hubiquitus. Para executar estas tarefas, as camadas de abstração do banco de dados criadas que permitem criar ganchos quando ações são terminadas e para mensagens

recebidas foram utilizadas, fazendo com que o servidor atualize sua cache na recepção de mensagens de controle e emita uma mensagem quando um canal é salvo.

Esta aplicação de Hubiquitus para o próprio servidor demonstrou o potencial da plataforma em diversos contextos, já que o próprio servidor o utiliza para se gerenciar.

Em termos de desenvolvimento do projeto, ele foi feito usando uma abordagem Scrum entre vários membros da equipe que trabalhavam em diversas áreas, tendo como ponto em comum o uso ou desenvolvimento de Hubiquitus.

O empreendimento foi levado a cabo em etapas que têm funcionalidades completas até o momento da sua publicação e recebiam posteriores correções de erros se necessário. Isto possibilitava diferentes grupos, dentro e fora da empresa, a utilizar rapidamente a biblioteca para seus projetos enquanto novas versões eram desenvolvidas.

Como a iniciativa foi na sua completude em código aberto e a sua publicação foi feita na plataforma GitHub, retorno foi recebido da comunidade rapidamente e foi possível interagir com outros grandes projetos para ter cooperarão mútua.

Atualmente Hubiquitus continua sendo desenvolvido usando como base as estruturas e o servidor criado neste projeto.

Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

# Projet de fin d'études Report

Done at Novedia Solutions

# hNode - multiscreen alert hub and message exchange server

Wasserman Federico
3e année – Option ISI

06 février 2012 – 07 juillet 2012

**Novedia Group**
94/96 rue de Paris
92100 Boulogne Billancourt

**Internship Manager**
Bureau Etienne
**Pedagogic Tutor**
Roncancio Claudia

**Abstract**

Development of a server for Hubiquitus, a near real-time message exchange API with support for remote command execution including improvements to its specification, validated through a client that connects to it through a web interface.

**Keywords:** Real-time, alert hub, publish-subscribe, message exchange, synchronous HTTP, multi-screen

# Contents

# 1 Novedia Group and Hubiquitus

## 1.1 About the Company

Novedia Group[9] is a Boulogne-Billancourt-based Group that has 4 branches, two specializing in marketing, communication and interactivity and other two specializing in technological knowledge.

Novedia Solutions, where this project was developed, is the branch that handles new technologies, for clients or for itself, creating complex software or software needing a deployment that are not simple websites or mobile applications. Also from the technological branches there is Novedia Decision that deals not in the software development but Business Intelligence creating data warehouses to analyze clients data.

From the communication and marketing branches there are Novedia Agency that deals with everything that is mobile software related or "front-end" for applications and Novedia Consulting that offers consulting plans for clients that need help to take decisions after an analysis of the client's data and needs.

Even though there are several branches that have different tasks they can do joint projects, for example, the front-end for a software being developed by Novedia Solutions can be made by Novedia Agency and big software projects that are not suitable to be made at Novedia Agency can be made at Novedia Solutions and then finished.

## 1.2 Introduction and history of Hubiquitus

Several Message Exchange protocols exist today, one of the most popular being XMPP (See appendix A), a protocol meant for chatting between clients.

Over the course of time, there was a need of a similar protocol but more structured, allowing computers to automatically parse and understand their content.

From this line of thought, the idea of Hubiquitus appears. It combines a publish-subscribe messaging protocol, adding a well defined structure for analyzing and handling messages with a *command* system that allows the remote execution of actions using messages.

Before the beginning of this project, Hubiquitus was used inside Novedia as a synonym for a pure XMPP system deployment. This means that they performed publish-subscribe using a XMPP module, but without the Hubiquitus defined structures or the commands support.

The publish-subscribe model used is a restricted broadcast model. It is a pattern where a publisher sends a message that is then processed by a server that broadcasts it to all subscribed users.

Contrary to a generic broadcast pattern, the publish-subscribe one only broadcast messages to users subscribed to receive it. *Subscribe* refers to the intention to receive a kind of message, generally by an identifier, while the publisher uses that tag to send the message.

In fall 2011 a data-model and a more detailed deployment specification, including a database and different client-communication protocols started to be developed to allow data mining, give actors a command system and reduce exchange latency.

The document born from this development is known as Hubiquitus Reference Guide, a Creative Commons[3] licensed document that contains the possible deployments scenarios, how the clients communicate, the structures understood by the framework and specifies a command sending system for actors to use.

The basic message exchange described by Hubiquitus is done through the use of a Publish-subscribe model using JSON[13] structures, that allows users to store them in database if wanted. In the case of Hubiquitus, clients are subscribed to *hChannels* and messages have a channel stamp to identify to which channel belongs to.

# 2 Project Organization and Objectives

## 2.1 Project Organization

The project had two sides: the development of a server that implemented the Hubiquitus Reference Guide and the improvement of such document.

The modification of the Guide was needed as it was a draft and changes were made by the team that worked in different aspects of Hubiquitus.

The development was done through an agile methodology: Scrum (see appendix B). Daily meetings meetings were done with the Hubiquitus group to discuss the advancement of each person and a small discussion of reference changes. If more time was needed for a particular point, a separate meeting was scheduled only with concerned parties.

For the task planning, a free software called RallyDev[16] was used. It automatically created charts for knowing project advancement, organized tasks and allowed a global view of what all project members were working on.

Objectives ("User Stories" in Scrum nomenclature) were set inside RallyDev for each Sprint (a fixed duration period) and were validated at the end of each one of them.

Through the development of the project the server was usable, even though it did not have all the features. This lead to a version concept that defined a set of features proposed to early adopters.

The versioning was decided from a feature perspective, while user stories set in sprints were technical oriented.

Because of this early release method, new features were added at each version and old ones reimplemented if needed be. If a better solution was found that better aligned new features with existing ones, it was adopted.

## 2.2 Planning

The development was done in sprints arranged in milestones (described as versions in Hubiquitus). Each sprint had a duration of about 2 weeks, depending on the overall complexity of user stories assigned.

The following is a list of the tasks done for each sprint separated in versions, managed through the RallyDev software. This list does not include meetings and time spent modifying the reference.

### Version 1

Sprint 0

| User Story | Complexity |
|---|---|
| XMPP Study | 2 |
| Socket.IO Study | 2 |
| XMPP Publish Subscribe (XEP-0060) Study | 2 |

Sprint 1

| User Story | Complexity |
|---|---|
| Establish socket from a client to the server | 1 |
| Create connection protocol | 2 |
| Implement Full connection (including XMPP part) | 2 |
| Parse and send XMPP messages to client | 1 |

### Version 2

Sprint 2

| User Story | Complexity |
|---|---|
| Receive subscribe packet and send XMPP equivalent | 2 |
| Receive publish packet and send XMPP equivalent | 1 |
| Receive unsubscribe packet and send XMPP equivalent | 1 |
| Receive list subscriptions packet and send XMPP equivalent. | 1 |
| Receive get messages packet and send XMPP equivalent. | 2 |

## Version 3

Sprint 3

| User Story | Complexity |
|---|---|
| Reattach: Allow to reuse XMPP connection | 2 |
| Allow to disconnect and erase connections | 2 |
| Connect: Readapt connection process | 1 |

Sprint 4

| User Story | Complexity |
|---|---|
| Connect: Add error codes | 1 |
| Allow clients to see technical errors from server | 1 |
| Connect hNode to XMPP using a Component connection | 8 |
| Allow the execution of a command in hNode | 2 |

Sprint 5

| User Story | Complexity |
|---|---|
| Change XMPP tags when sending commands | 2 |
| Create hPublish command to publish using hNode | 3 |
| Create hSubscribe command to subscribe using hNode | 5 |
| Create hUnsubscribe to unsubscribe using hNode | 1 |
| Create hGetSubscriptions | 2 |
| Allow the obtention of channel list | 3 |
| Allow commands to override default timeout | 1 |
| Add transient support for commands | 1 |

Sprint 6

| User Story | Complexity |
|---|---|
| Create command GetLastMessages | 1 |
| Create command for hChannel creation | 5 |
| Create local cache for hChannel list | 1 |
| Async save and publication of hMessages | 2 |
| Command controller instance per user not centralized | 2 |
| Allow the obtention of channel list | 3 |

## Version 4

Sprint 7

| User Story | Complexity |
|---|---|
| Create hRelevantMessages command | 2 |
| Create hGetThreads command | 1 |
| Create hGetThread command | 0.5 |
| Change hHeader structure | 1 |
| Change driver for MongoDB | 3 |
| Create admin channel for distributed cache | 2 |
| Use client connection instead of component one | 2 |
| Add filters for message reception and related commands | 4 |
| Use filters in getLastMessages, hGetThread, hGetThreads and hGetRelevantMessages | 2 |

## 2.3 Document Organization

The document is divided in three parts: first, in section 3, the necessary information about Hubiquitus is presented to understand the server. Following, in section 4, are details about the server developed.

This section starts with a server overview, followed by the technologies used. After, the architecture and deployment are shown, explained in detail on subsequent subsections, comprehending solutions developed. This section concludes with the details of the server validation.

The third part of the document refers to the project conclusion and a personal assessment with an appendix to offer more technical documents.

# 3 Hubiquitus Data Models

The data models in Hubiquitus are the structures understood by clients and servers of the network. These structures are mainly envelopes for the data itself that allow efficient processing and indexing.

They are defined as JSON[13] encoded objects, making them portable to different systems and languages.

The format was chosen as a compromise between format structure overhead and human readability. It is also closely related to Javascript, as is a subset of it, allowing deep integration with web technologies, that are the main applications of the Hubiquitus API.

## 3.1 hChannel Data Model

The publish-subscribe model uses an identifier in the data sent to discover which clients are subscribed and should receive the message. A *hChannel* is the structure in Hubiquitus that contains that identifier.

This structure also acts as a permission controller, having a list of *members* that are clients allowed to publish and retrieve messages from the channel stored in the database.

Besides identifying and controlling permissions, the hChannel has metadata that extends published messages properties, adapting them accordingly to channel parameters.

The full structure of a hChannel model can be seen in the image 4 in the appendix.

## 3.2 hMessage Data Model

Sending *data* through the network implies the creation of a *hMessage* structure, an envelope for the *data* that wants to be sent containing metadata (that can be later extended or changed by channel specification).

This envelope allows the identification of the content by its date, publisher and the channel it belongs to, allowing the correct publication, also having other optional attributes like location.

Even though the payload is user-dependent and can be any UTF-8 encoded character, certain payloads with defined structures are described in the Hubiquitus Reference Guide and can be specified accom-

panied by a *type* property correctly set to allow command execution over these special payloads possible.

The full structure of a hMessage model can be seen in figure 3 in the appendix.

## 3.3 hCommand Data Model

Commands are actions that can be executed on the server or another actor. A command can be a directive for the server to retrieve a certain kind of messages, unsubscribe from a channel or a custom action not related to Hubiquitus like shutting down a computer.

Commands are defined as a pair: a *hCommand* and a *hResult*. The execution instructions and parameters are given in the hCommand, while the hResult contains the response.

A hCommand is identified by its *cmd* attribute that is user-defined and can be any string, thus allowing the creation of arbitrary commands. As with hMessage type, certain cmd keywords are reserved for commands that are part of the Hubiquitus core, and a server implementing the reference specification must support them. Some of those commands include the subscription and unsubscription from a channel, the channel creation and generic message retrieval.

The complete structure of a hCommand can be seen in image 5 in the appendix.

## 3.4 hResult Data Model

The counterpart to a hCommand is a hResult, containing the response to the execution of a command. This response can be a simple success or failure code, or more complex data, like an array of retrieved messages from the database.

This response is expected to always arrive, even if the command fails or times out. The timing out must be insured by the actor executing the command, while the nonexistence of the receiver must be insured by the server to which the sender is connected.

The complete structure of a hResult can be seen in figure 6 in the appendix.

# 4 Hubiquitus Server: hNode

The reference implementation of a Hubiquitus Server is the hNode. It is closely related to the Hubiquitus Reference Guide as they were written in parallel and while the definitions of the guide define the

server's behavior, when a change was needed the latter modified the former after an initial implementation.

The server's name derives from the platform used to run it, NodeJS[15], a Javascript server-side application interpreter built from the Chrome Javascript Engine.

In essence, the server has three main tasks: allow clients to connect to it maintaining a session, execute arbitrary commands in the form of hCommands and process incoming and outgoing hMessages using the model publish-subscribe, filtering or modifying them.

## 4.1 Technologies and Tools

While the server itself uses NodeJS as a framework, being Javascript it's language, several other supporting technologies and tools were used.

The choice of NodeJS as a platform to run the server comes from the advantages in relation to Hubiquitus and its paradigm: a completely asynchronous model.

As all Hubiquitus is defined using JSON that is a subset of Javascript object model, Javascript as a language integrates seamlessly with the Reference objects as they can be used without modification from the specification.

The asynchronous model, while not related to the Hubiquitus Specification, is a paradigm created to handle different requests without overloading servers, as ongoing tasks can be put on a queue for execution when the processor frees up. They can be seen when programming as threads working in parallel, as no assumptions over the execution order can be made.

From a supporting point of view, the project uses GIT[7] as a versioning system, more specifically a website that hosts a GIT server called GitHub[8], a mix of a versioning server with a social network.

Since the project is open source, the use of such a website allows code access to everyone, interaction with other projects and Hubiquitus promotion.

As all libraries used are also open source, and almost all of them are also hosted in GitHub, the usage also allowed the correction of bugs in said libraries by sending patches to interested parties.

Other technologies include XMPP (see appendix A) from which Hubiquitus originated that acts as a supporting server for handling messages and authentication, as explained in section 4.6, and MongoDB, the database used to store all server documents, having a separate chapter to describe it in section 4.7.

In the tools context, for validation purposes, the Mocha Framework

[11] tool was used described in section 4.11 while RallyDev software [16] functioned as a manager for project activities, as was shown in section 2.1.

## 4.2 Server Architecture

The server is a mix of connectors and a sandbox environment for command execution.

The latter because the main task of the server is to execute generic commands following a syntax, and while the execution was passed to the command, a server component controls that execution so that in case of an error, the control flow returns to the server instead of crashing.

The former because message handling and authentication are controlled by XMPP, as can be seen in sections 4.4 and 4.6. As a result of this choice, modules are needed to translate the Hubiquitus login packet and messages arriving from the XMPP network in the form of XML.

As such, the server architecture can be seen in figure 1. This architecture is explained in detail in the following sections.

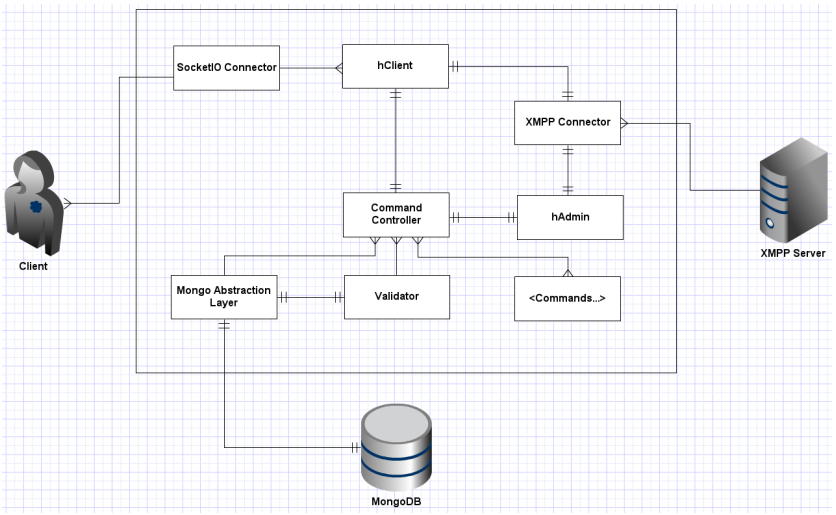


Figure 1: System Architecture where hClient handles parsing and filtering, hAdmin is the server XMPP connector and Commands is an alias for all available commands. Also, shown are the mongo connector, the command controller, the client connector (SocketIO connector) and a structure validator module.

## 4.3 Deployment

The simplest deployment of a hNode is an instance of the hNode accompanied by a XMPP Server and a MongoDB database.

This deployment has a potential performance loss as NodeJS is not multithreaded. Meaning that hNode must handle all connections with the same thread even though they are completely asynchronous and self contained.

This poses a problem when there are an increasing number of clients and forking a process for each client becomes heavy as for each one an instance of NodeJS must be launched and it consumes approximately 10MB.

The solution found is to launch several instances of hNode, each listening to a different port and for the clients connected to the same port, a monothread model is used.

This leads to a load balancing that is client dependent, meaning that the server charge will depend on how random is the choosing of connecting ports by the clients.

Thus, a second type of deployment can be made using several instances of hNode connected to the same XMPP server. This functioning is assured from the hNode side as it synchronizes its instances by using systems like the distributed cache (explained in section 4.10).

This is the expected hNode form of deployment.

## 4.4 Client connection and authentication

The HTTP protocol was thought as a model where clients requested data from servers and they responded in a process called polling.

Through the years, developers started to feel the need for servers to send data asynchronously to clients. The client would make a request to the server and if no data was available, the request would stay opened until new data arrived, and if the request ended, a new one would be made automatically, being called long-polling.

In recent years, a new protocol called WebSocket was conceived, one that can be seen as an extension of long-polling. The basic idea is that after the initial request, the connection will stay opened indefinitely, minimizing the overhead imposed by subsequent requests in long-polling.

With the advent of this protocol and several others competing ones, the choice of the *best* protocol depended on browser support, sometimes making WebSockets a bad choice if older browsers should be supported.

Most of the clients connecting to hNode do so from a web browser. Because of that, the server needs to support a wide range of browsers

using compatible protocols, while minimizing the overhead from older protocols since a lot of data will transit. To solve this dilemma, a library called socket.io[17] was chosen.

This library proposes an abstraction from the protocol used supporting several and choosing, in order of priority, the best available for the current user, thus allowing the server and client to be coded once while letting it handle handshake and transport idiosyncrasies.

While a handshake is done through socket.io to establish a connection to hNode, the authentication is handled by a XMPP server backend that serves as a signaling and authentication provider.

The usage of XMPP as a backend for authentication purposes allows the use of several connectors for handling connection, like authentication through LDAP servers. Also because of the communication model of XMPP, a contact list is kept, informing of status changes from people in that list, making it a simple and efficient signaling system.

This structure makes the client connection a multi-step process:

1. Client establishes a connection through socket.io to hNode.
2. Once handshaked server opens a client connection to XMPP.
3. Once authenticated, inform client of successful connection.

## 4.5 Session Reattachment

Even though the XMPP backend that handles authentication is generally connected to the hNode through a fast connection, or is even executing locally, there is an overhead of establishing a new connection, making a handshake, etc.

This problem is worsened because one of the client target platforms for the use of Hubiquitus are mobile web browser ones, that tend to have a unstable link and doing a re-connection each time would make the user experience lagging.

Taking this into account, a feature called *reattach* was implemented. In simple terms, it leaves the XMPP connection opened for a while after the client connection closes abnormally to allow a new connection to be created and reuse the same XMPP one.

Not only the XMPP connection is kept opened when the client socket is closed, but also all the Hubiquitus context, meaning that actions affecting the active session will continue to be valid after a re-connection.

To implement this feature a *reattachment protocol* was created. This was needed to identify the session to which the user needed to be reattached and to forbid other users of stealing sessions.

The steps for reattaching are as follows:

1. Client establishes a connection through socket.io to hNode.
2. Client sends a *reattach* packet.
3. Client acquires the old XMPP connection and the Hubiquitus Context and is informed of successful reattachment. In case of failure or destroyed context by timeout, a new connection packet must be sent to authentify the user, without establishing another connection through socket.io.

The Reattach Packet is a JSON object containing the identifying information of the session. It is sent once an initial connection is completed and it is composed of three attributes:

**Publisher**
: A full JID[1] obtained after a successful connection to the XMPP server.

  JID is the user identifier form used by XMPP, hence by Hubiquitus and has a general form <*user*>@<*domain*>. The *full* part refers to a classic JID added of a *resource* that serves as an identifier between different connections of the same user, having a final structure <*user*>@<*domain*>/ <*resource*>.

  This can be considered a unique id in the network, as XMPP handles the resources it grants.

**SID**
: A session ID given by socket.io at connection time. This works as another unique identifier between the hNode instance as it is used by socket.io to differentiate between all the open sockets that handles.

  Even though it seems redundant to use this value once a unique identifier is also given, this works as a password as it is only known to the user and the server, while the full JID is public as it is the address of a specific client.

**RID**
: The RID, short for Request ID, initiates as a random number given by hNode. This random number is then increased when a hMessage, hCommand or hResult transits through the socket.

  *Transit* is understood as the reception of a message or the sending of a command.

While the Publisher works as an identifier and the SID as a *password*, these two values can not guard against a replay attack. This is because those values do not change over time, as after an initial connection they remain constant.

This is what RID addresses. Because it depends on messages exchanged and they cause a modification of a local value while the value itself is only transferred once and not all packets sent cause a modification it inhibits the use of a replay attack.

Even though ideally those three values would match against the ones stored in the client session, in practice, tests have shown that after a disconnection sometimes messages were sent, changing the value on one side while remaining constant in the other.

This discrepancy made the reattach process useless, generally failing to reattach and forcing the user to send another packet, effectively increasing connection time.

The solution found was to allow a small *window* of accepted RIDs. This means that instead of matching the sent RID against the value stored in the server, it is tested against a small range of values, by default RID plus or minus 5.

Although this brings again the replay attack problem, the replay window is kept small enough to be difficult to explore.

Because all these tests are performed within the hNode, the reattachment dramatically reduces the connection time of a client, specially if authentication is done through the use of an external service like LDAP, that adds another connection step besides the XMPP one.

Even though this feature is part of the hNode server and can be implemented by clients, it is not defined in the Hubiquitus Reference Guide as part of the Hubiquitus Core Reference.

## 4.6 Message Handling and Distribution

hMessages are *published* in hChannels, where all subscribed members receive it following the publish-subscribe pattern. This action is done through a hCommand with the identifier *hPublish*.

XMPP specifies a publish-subscribe model[5] that can be used with the XMPP protocol. In the specification *messages* are *published* to *nodes*, that are the equivalent to hChannels in Hubiquitus nomenclature.

Because XMPP servers have already implemented this feature, the publish-subscribe implementation used in hNode uses the XMPP server as a backend. This allows users to publish hMessages that are ligther than XMPP stanzas as they use JSON and are tailored for Hubiquitus, while the server, that has a fast link to the XMPP server, sends the heavier XMPP stanza, thus reducing bandwidth and response time client to server.

### 4.6.1 Message Reception and Publishing

As a result of using XMPP as a backend, when a user *subscribes* using a subscription hCommand, in addition to store the Hubiquitus subscription in the database it will subscribe the user at the XMPP server using the XMPP protocol.

The same can be said of hPublish. It will validate the hMessage but instead of handling the distribution, it will publish it at the XMPP server via a XML stanza and let it handle it.

Consequently, each connected client (*hClient* in the architecture image shown in figure 1) has an instance of a XML parser for stanzas arriving from the XMPP server that will extract the encoded hMessage and properly send it through the socket to the client.

It is important to highlight that the use of a XMPP server for handling message distribution can be changed at any given time as it is transparent to the user.

### 4.6.2 hCommand Handling

In the special case of sending a hCommand that is not directed to the hNode itself for execution, but to a command capable client, the hCommand will be sent using XMPP, encoding it inside a XML stanza.

This is implemented in such a way for two main reasons:

1. Not all connected clients are necessarily connected to the same hNode instance. hNode is prepared to be executed in an environment with multiple instances, mainly for load balancing purposes. Meaning that is the XMPP server that knows all clients.
2. The current implementation does not have a full list of connected clients to the instance for sending commands without passing through XMPP. While this list could speed up the hCommand sending and later reception of the hResult, it would not solve the other items.

In conclusion, XMPP is the link that ties together the different servers and clients. While the server is focused in creating a high-level interface for more complex operations, low level ones, like the knowledge of all connected clients enabling the distribution of messages is left to the XMPP server.

## 4.7 Database Component for hNode: MongoDB

The selected database to store hMessages, hChannels and other Hubiquitus objects is MongoDB.

This is a NO-SQL database, working on a *document* model, meaning that what is stored in the database are JSON documents and they are separated in *collections* that group them without imposing structure equality contrary to the SQL model that stores lines in tables having fixed *columns* for all elements.

The reason for using this model are as follows:

1. hMessages payload are user-defined, potentially having different types, and other structures that tend to evolve over time. A NO-SQL approach with a document model allowed different documents to be stored in same categories, even though they do not share the same structure.
2. Documents stored in the database are JSON encoded. Since Hubiquitus has all its structures defined using JSON objects, this solution permitted the storage of native Hubiquitus objects directly to the database, making the transition of Hubiquitus-network to storage a seamless one.
3. On-the-fly collections creation. Meaning that a new collection can be created at run-time without affecting the database. A useful feature for making a horizontal partitioning when storing messages.
4. Horizontal partitioning capabilities (Sharding in MongoDB), are meant to be done in a collection based approach. This implies that each collection can be ultimately *sharded* to a different location, allowing an easy and fast data distribution if needed for a deployment.

### 4.7.1 Database Structure

The database is divided in collections that contain documents. All documents stored are Hubiquitus Data model ones with small modification, save special internal documents used to accelerate certain actions.

The collections with the documents structure they contain are depicted in figure 2. Although a document structure is present, this does not mean that other JSON objects cannot be stored in those collections, but that the database, as of now, stores them like this.

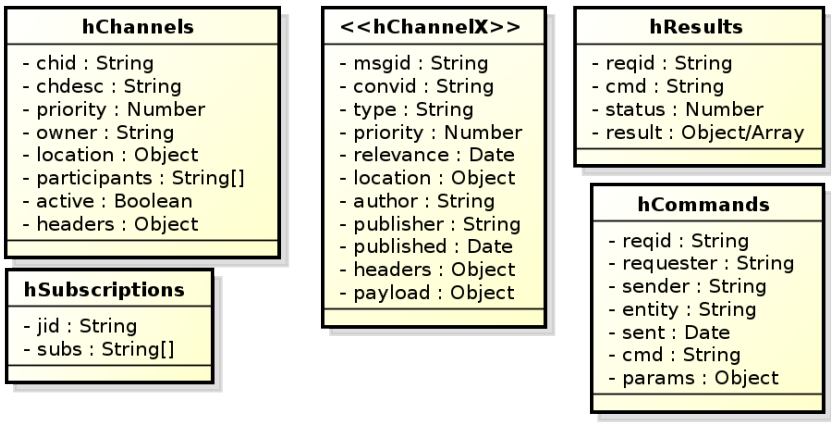


Figure 2: Mongo Collections, where <<hChannel$X$>> depicts the channel name

Each hChannel has their own collection for storing their hMessages, that is what the special collection <<hChannel$X$>> shown in the picture characterizes. This allows sharding hMessages from same channel to the same location, optimizing partitions.

Since not all hMessages are stored in database and some channels will never have persisted messages, the creation is done at run-time when the first hMessage that needs to be persisted is received.

### 4.7.2 hNode to MongoDB Connection Layer

The driver used to connect the database to hNode can be seen as very primitive: each time a document is persisted, options must be passed, a callback needs to be specified, etc.

As a result of those limitations, an abstraction layer was made. Simplifying access to the database through the usage of methods that store in the correct collection and the correct format Hubiquitus documents.

These documents can be stored receiving a confirmation of the recording process through a callback or nothing if the storage wants to be done completely asynchronously and go through an adaptation process to optimize disk usage.

1. hMessages are passed through a method that will store them to a collection having the name of the channel (a new collection is created if needed).
2. The hMessage *msgid* is generated by the hNode at publishing time and considered unique, so it is used as primary key (PK)

in the database. By using this value, there is no need to use the automatically generated PK by MongoDB.

3. hChannels *chid* are also considered unique and thus used as primary keys, discarding the one generated by MongoDB.
4. hCommands and hResults have a ReqID, an identifier that is sent by the client to correlate result to command. This value can not be considered unique as nothing guarantees its singularity.

   As a result, the received ReqID is forgotten and a new one is generated only for storing purposes, so as to correlate a hResult to a hCommand.
5. All Structures are stripped of empty JSON attributes and default values. This means that is the responsibility of the hCommand retrieving data to reconstruct missing attributes with default ones and left others empty if needed.

These modifications, even though just change one or two fields are considered valuable, as a lot of messages can be exchanged, and each time irrelevant data would be stored.

Because the structure of the database is well defined, the collection where the documents will be stored is abstracted, as the user will call the helper function that will correctly store it in the right collection.

Since a high level storing interface was created, some features were added to the standard persisting mechanism provided by the driver: Validation and Hooks.

**Validation**

Validation refers to the action of verifying a document before storage, if the validation does not pass, the document will not be persisted.

This validation is done through the application of functions in a asynchronous fashion. hCommands or the hNode itself add functions to a list of validators, and if all functions validate the document, it can be stored.

To validate a document, a validator must trigger a callback at the end passing a return value, informing if the document analyzed passed or not.

This construction is useful for three main reasons: It executes asynchronously not blocking other connections, it is generic so as to be applied to different Hubiquitus documents without modifications and allows specification evolution as a new validator can be added instead of altering existing code.

**Hooks**

The hook concept in hNode is used in what is called *onSave*

functions. These are a group of functions that will be executed, if and only if, the storage of a document succeeds. This means, if it passes validation and no error is reported from the database at persisting time.

This feature, as the validators, works asynchronously. A function is added that needs to trigger a callback at the end and there is no guarantee in which order the *onSave* functions will be executed.

*onSave* functions allow the addition of third-party triggers as a system administrator may want for instance to receive an email when a publication is correctly persisted. Currently, it is used for the cache synchronization mechanism as shown in the section 4.10.

In case the caller wants a confirmation that everything went correctly, the callback passed will only be executed once all the *onSave* functions are finished.

These features are generic and applied in a collection based approach, allowing collection-based customization, making it a matter of calling a method

$add\{Saver|Validator\}(\langle Collection\rangle, \langle Function\rangle)$

This brings a problem in the case of hMessages that are not all stored in the *same* collection but sometimes *global* triggers applied to all hMessages are wanted.

As a result, there are two ways to add hooks and validators to hMessages: through the use of the collection where the hMessages will be stored, triggering the hooks when a message in that specific hChannel is stored or by the use of a *virtual* collection called *hMessages*.

This collection does not exist in MongoDB, but the hNode understands that it refers to the whole set of hMessages and will be applied to all hMessages. Triggers added to a collection and to the *virtual* collection will be all executed asynchronously.

## 4.8 Command System

The command system is the component charged of executing received commands from clients destined to the server.

Because of the commands nature, the system works as a controller, passing the control flow to an external plugin, possibly created by a system administrator.

A command is identified by the hCommand *cmd* attribute and the controller expects a result from the plugin that will be formatted as a hResult for later sending to the client that executed the command.

The general sequence followed by the execution of a command is as follows:

1. Analyze the structure of the received packet to check validity.
2. Check if the command must be persisted in the database. In that case do so asynchronously.
3. Search for a file named <*cmd*>.js inside the defined command folder.
4. Check if command overrides the default timeout for commands. And launch a timer for the command execution.
5. Launch the command inside a *try-catch* block to avoid crashing.
6. Receive result from the command, catch error or timeout
7. Format it to a hResult, persist it asynchronously if needed and send result to client.

Since hCommands are what administrators that deploy the server will want to extend to add their customized actions, this allows a flexible and elegant way to execute external sources.

The only requirement is that the plugin follow a small and well defined structure that is to have a initialize method with a specific signature and that at the end of the execution a callback function passed as a parameter to the initial method is triggered.

Each client connection has its own instance of this controller, allowing a completely asynchronous execution between the connected clients. This is also useful because it allows the controller to pass the commands certain attributes about the client executing the command to customize its execution.

Even though the Command Controller acts as a supervisor of the commands, it does not restrict their usage, meaning that they have full access to the database and to the XMPP connection of the client.

### 4.8.1 Available resources for command execution

When the initial method from a hCommand is called by the controller, three parameters are passed: the received hCommand, a callback that has to be called when the method finishes executing and a context.

While the hCommand is the exact object as received by the controller, allowing the plugin to review the sender and arguments sent with the command for correct execution, the context allows access to the stored attributes of the running client.

In particular, the context provides the following high-level methods:

**Message Filtering**

This method allows the usage of filters, as explained in section 4.9, inside commands. This is useful when commands need to retrieve messages from the database, but not all of them are relevant for the connected user.

The usage of this method is not mandatory but it cannot use filters selectively. It is expected that commands that find the need to filter messages do so. The function works as a black box for the command that will receive a list containing only the messages that are valid according to the tests.

**IQ Sender**

Because some commands need to execute administrative tasks and modify the user status at XMPP level, the client's XMPP account is available to the commands for using.

IQ is a type of message that can be sent through XMPP, generally used to perform administrative tasks. It shares some properties with hCommands as both expect a response after sending the message.

Since the library used to connect to the XMPP server is not a high-level one, only providing basic support for sending and receiving raw XML stanzas, a convenience method was created.

This method receives XML content that the command wants to send, envelops it inside a proper IQ stanza as defined by XMPP and sends it using the client's XMPP connection. Once a response is received, the method will call a callback sent by the command to continue execution.

Even though the command provides a callback that will be executed after the reception of a response, the command is not obliged to wait for the dispatch and reception, as all is executed asynchronously.

In addition to the context and other parameters received through the initial method, there are two notable singletons that can be accessed through the import of files:

**Server XMPP Account**

The server XMPP account is almost equivalent in structure and functioning to that of a client account.

It is a special account that is created at XMPP level and used solely by hNode instances. It is not defined in Hubiquitus reference guide, but provides several advantages compared to always using the client's account.

The main purpose of using this account is the sending of XMPP IQs that are not necessarily tied to the user, like the creation of a channel, that at XMPP level are not owned by the clients.

This account will also be used for internal purposes and not directly via a hCommand, as is the case with the distributed cache, as shown in section 4.10.

**Database Connection**

The database connection is the singleton that allows access to MongoDB through the application.

All tasks performed using this module are executed asynchronously and allow the saving, retrieval and validation of objects.

The methods and available functionality are described in section 4.7.

### 4.8.2 Publish-Subscribe hCommands

While hCommands are extensible and only need to follow certain guidelines to work, some default commands are already provided to allow users to perform Hubiquitus related operations. One of the installed groups is the Publish-subscribe one.

This is a set of commands that share a common trace: they all need to use XMPP to perform a part of the action as hNode relies on XMPP for distribution and handling of messages, as explained in section 4.6.

The commands that comprise this genre of commands are the following:

**hChannel Creation and Updating Command**

The creation and updating of a hChannel is done through the same command. This arrangement works because to update a channel, the whole description with the new values is needed, not only a delta. Thus the difference between updating and creating resides in the fact of the previous existence of the hChannel *chid*.

It is worth noting that the updating process differs a little from the creation in terms of members handling. This happens when a user is removed from the members' list as their user must be unsubscribed from the hChannel (the contrary does not happen as a new member starts off just having the right to publish and after they decide if they want to subscribe).

Finally, channels are created using the hNode XMPP account that will own all channels at XMPP level. This structure allows the management of all channel aspects at any given time, as this account is always connected while the server is running, enabling

other users to manage the channel even though the owner is offline.

**hChannel Subscribe and Unsubscribe Commands**

Subscribe and Unsubscribe commands are two different commands that share common tasks: both alter the same MongoDB collection and both send a subscription-related XMPP stanza to alter the client status.

What these commands do is evaluate the current status of the user, alter it in the database and send a XMPP stanza that will effectively change the status at handling level.

Even though the XMPP stanzas could be sent using the clients' XMPP account, the one performing these tasks is the hNode one. Thus allowing other commands to call on these hCommands to act on behalf of other clients.

If the clients account were used, the executor of the command should have been the client as only the user himself and the owner of the channel can alter subscriptions status.

This way, as the hNode is the XMPP owner of all channels, it is free to alter everyone's subscription. Consequently the user in question would not need to be online.

**Publish Command**

The publish command receives as an argument the hMessage to be published. This message will be processed, validated and, if successful, will be published using XMPP so that the XMPP server handles the distribution.

Because no errors are handled from XMPP as the stanzas are created in the server and are expected to be correct, once the hMessage is validated, a hResult informing of a successful publication is sent to the user, while the *real* publishing and persistence in database are done asynchronously.

Even though XMPP has a sender, this attribute is ignored as the one in the hMessage is used. This allows, as with the subscription commands, to use the hNode account, enabling other commands to send a hMessage in behalf of another user.

If the process, that access the disk and sends a network message were done synchronously, the server would remain blocked for attending other requests. But as a result of this organization, the process is highly asynchronous and does not block the hNode.

Other commands that handle publish-subscribe actions were also created. These commands query the current status of subscriptions and channels and send the result to the user. Since this actions do

not need to make any changes at XMPP level, they are limited to querying data stored in MongoDB.

It is important to highlight that while the subscription and publication are done through commands, that are initiated by the user, the reception is not done through a hCommand, as it is initiated server side and as such is handled by the *hClient* in hNode.

This reception is done through the client module that handles online messages as explained in section 4.6.

### 4.8.3 Database hCommands

Another group of implemented commands are the ones used to retrieve hMessages from the database. These commands perform queries using the native interface provided by the MongoDB driver available for NodeJS.

Due to the fact that the hNode tries to adapt to generic situations, leaving space for administrators to add support for their own needs, in general terms, they are generic queries that format the output to make them *Hubiquitus compliant* as seen in the Hubiquitus Data Model section.

**hGetLastMessages**
: Retrieves from the database the $N$ last messages, as passed in the command parameters. Contrary to first impressions, getting $N$ messages is not an atomic query specifying a quantity, as the filters that are applied to the messages alter the result and subsequent queries might be made.

**hRelevantMessages**
: Similar to hGetLastMessages, instead of returning an array of $N$ last messages, returns all published hMessages in a hChannel that are allowed through the filter.

**hGetThread**
: Using the *convid* attribute from a hMessage, retrieves all published messages having the same conversation ID in order of publication.

**hGetThreads**
: Retrieve from a channel an array of IDs corresponding to all different conversation IDs in a hChannel.

  Two algorithms were implemented to retrieve this array: one using map-reduce[12], and the other not. Both having a complexity $\mathcal{O}(N)$ where $N$ is the quantity of messages, but map-reduce having a bigger constant.

Test made showed that map-reduce was slower when executed in a single machine environment. Possible explanations are the use of another algorithm and the lost of parallelism as it is a single machine and the algorithm was conceived to be parallel.

Consequently, an algorithm thought for a single machine environment execution was created that is only faster than map-reduce if the latter is not executed distributively.

## 4.9 Message Filtering System

Filters allow the selective reception of hMessages through a session. This means that while the user is connected, once he sets a filter to receive only certain hMessages, not matching ones will be ignored.

To do so, an object defined as a *hFilterTemplate* was created. This object sent by the user contains filtering attributes that hMessages must match, having a few special restrictions. For making this match, the template structure is similar to that of a hMessage and only filled attributes will be compared.

The system allows enchaining several filters. Contrary to the normal usage of a NodeJS application where all filters would be applied asynchronously and later a callback would be called, the filters must be applied in order.

The reason for this implementation is that a filter may stop more messages than others, thus the other filters would not need to be executed. This means that a performance decision is left to the user that can accelerate the execution of all tasks just by changing the order of set filters.

A special attribute of the template is the *radius* one. This attribute calculates a radius from a certain location set also in the template and only allows messages inside of it.

To calculate this radius the Haversine formula[14] is used. This form starts showing small meter errors in calculation at distances bigger than 10.000 kilometers[2]. Because the radius was an attribute meant to be used inside cities, or between cities, these errors are acceptable, as they are in the order of meters.

Since Javascript allows object introspection, a hMessage is matched against the template using a recursive function that covers all object's attributes. The method will verify if the attribute in the filter exists and in that case if it is a basic type that can be compared for equality (not by reference). If not, it will recursively analyze the content of the attribute until it finds a basic type to check.

Filters are applied automatically to all incoming messages and they do not interfere with publication. This means that while a user filters

certain messages coming from a hChannel, it can still publish those kinds of messages to it.

The support of this feature inside hCommands is optional and is provided through the use of the same function that filters incoming messages and is available through the reference to the user that hCommands have while they are executing.

To allow the management of the filters, three comands were created:

**hSetFilter**
: A hCommand that receives as a parameter a hFilterTemplate and adds it *in order* to the list of filters to apply. In case the filter exists (verified by the *name* attribute), it will substitute the old one.

**hUnsetFilter**
: Removes a filter applied identified by the name passed as a parameter.

**hListFilters**
: Returns an array of hFilterTemplates that are currently being applied in order of application.

## 4.10 Distributed Cache for hChannels

One of the most accessed objects when executing hCommands are the hChannels ones. This is due to the fact that hChannels contain permissions for accessing hMessages, altering subscriptions or publishing.

Two actions are almost always executed when a hCommand runs: member verification and activeness of the channel. This leads to frequent database access to retrieve information, meaning a possible speed loss as this may cause disk accesses.

Considering that the hNode needs to be synced between different instances, as explained in the section about deployment (section 4.3), a common approach would be to use a third party software component like Memcached [4], a general-purpose memory caching system. While this procedure solves the problem, it adds another layer of complexion to the deployment and a lot of unused features.

Since hNode had some restrictions to what Memcached features offered and synchronization could be done through the use of Hubiquitus system through a hChannel, a self-developed cache system was created. This cache was created keeping in mind the following hNode constraints:

- hChannels are not numerous.

- hChannels are rarely updated or created after an initial setup.
- cache states need to be shared between different instances of hNode.
- small temporary discrepancies between instances are admissible.

The resulting cache is seen as a normal Javascript object available to the plugins executing commands. This allows a cache system of key-value pairs (as is the nature of Javascript objects) that recovers the value synchronously with one memory access.

Because of the channels' number constraint, they are all kept in memory, dismissing the possibility of a cache miss, meaning that if the channel was not found in the object, it can be assumed nonexistent.

Initial cache load is done at server start-up by a database query and subsequent modifications are received through a special hNode administration channel: *hChannelAdmin*. This channel is exclusive to hNode and the only member and subscriber is the hNode XMPP account.

hMessages for cache updating have a type set to *hChannel* and a full JSON hChannel as a payload. Once a hChannel is updated or created, a hMessage is published and, as subscribers, received by all hNode instances. When the message is received, the cache will be updated to mirror new modifications.

This is possible as a result of a read-only cache and special functions created for saving in the database. The MongoDB abstraction layer created as shown in section 4.7.2. allows the use of methods to save a hChannel, and when that *successfully* happens, automatically trigger a function (onSave hooks).

The triggered function is configured at start-up and is responsible for publishing the hMessage needed for updating. The message parser in the hNode account is in charge of reading the hMessage triggering another function that will update the cache when a hMessage of type *hChannel* is received.

Because the hMessage *type* and hNode *onSave* features are used, this process remains generic so that it can be used to speed up other parts of the system that actively access the database if needed in the future.

## 4.11 Validation

The validation of the server has two faces: the automatic one through the use of a Test-Driven-Development(TDD) model and a manual through the use of a client that connects to the hNode using a web interface.

The first validation was done every time a task was finished, while both of them were executed at the end of each sprint for accepting the user stories marked as finished.

The TDD model was applied to all server component, being a mix of integration and unitary tests. This is a result of the deep integration of the server with XMPP and MongoDB while mocks for these two components were not available and were not created.

This model is based on the idea of writing tests first and then code the functions. As a result, more than two hundred tests were written to comply with the specification.

Since everything is written in Javascript and has asynchronous calls, the software used for testing needed support for it. Consequently the application chosen was Mocha[11].

Mocha is a Javascript testing suite with support for asynchronous tests, coded in a Behavioral testing fashion. This means that tests are separated in classes and each test has a mandatory *promise* that needs to fulfill, for instance "should return a client connection".

For the manual validation, a Javascript web-client that uses socket.io to connect to the hNode was created. It is composed of a basic html page with buttons, depicted in the appendix figure 7, and scripts for connecting and allowing the use of server functions.

The scripts done for connection and interaction can be used as clients for other projects as they are generic and work as a library. They were put together with the html page to allow a simple demonstration of the hNode capabilities and features and for validation purposes.

Since the client library was also done in Javascript and is even NodeJS compatible, a TDD methodology was also used, with more than a hundred tests. They allowed integration tests to be done client side in relation to the hNode by using also Mocha.

# 5 Conclusion

Hubiquitus and hNode were not thought to be written from scratch without reusing already existing technologies. Instead, it uses strong points of different solutions to create a new protocol and improve the weaknesses while adding new functionality.

Currently, Hubiquitus is used for different projects, ranging from websites doing a showcase of television viewers in real-time [10], to projects monitoring buildings.

This shows Hubiquitus flexibility. By itself it does not provide a solution, but works as a tool to create systems and implement ideas.

The project future has different paths ahead, some already having a draft in the Reference Guide as Bots (special automatic clients performing publishing messages and capable of receiving commands) and others only discussed at meetings as direct connections between clients after an initial discovery to reduce latency.

As of today, there are eighteen thousand lines of code written, with code refactored. This was done to adapt to new features, as shown in figure 8 of the appendix, exposing the intense activity on the server.

Taking today and the future into account, it is safe to say that Hubiquitus is a work in progress, with no deadline to be concluded. Each time a new feature is added, a new version is released and code is changed, being this feature a user suggestion or an internal need of a Novedia project.

# 6 Assessment

The experience of working in a big company with a team of people is something new to my curriculum. Having worked in small teams inside StartUps, the change is considerable.

The usage of scrum showed very good results, although several critics where discussed between the team at the beginning. This concluded with a few adaptations to *normal* Scrum by not using certain features as Complexity Cards and each person deciding their own complexity for their user stories.

From a developing point of view, the usage of a new technology as Javascript was interesting. Specially when a change of context was needed and the client library needed to be developed and I had to use the same language with a different purpose (web usage) as the platforms were different.

Each step taken needed studies as I was generally the only person to deal with each library, server or platform used. Thus, the studying came from tutorials, manuals and communities, giving me a good connection with renown developers.

Overall the project was a very satisfying experience: a mix of developing using new technologies and discussions over specifications without fear to change what was already developed because a better way was found when the specification changed.

# Appendices

## A XMPP

XMPP is a streaming XML Protocol[6] for exchanging data. In this protocol a connection is established when each side opens a specific XML tag while everything that happens in this connection will be well-formatted *children* XML stanzas of the initial tag.

The user identification is done by using a structure called a *JID* (Jabber ID), a unique user identifier in the form of *[user@domain/resource]*. The peculiarity of this ID is the fact that uses the notion of *resource*. A resource is an identifier given by the server to distinguish separate connected sessions of the same user. The protocol defines that a message from another client sent to a JID *without* the resource will be broadcasted to all the connected users matching the JID while the use of a specific resource sends the message only to that session.

XMPP establishes a connection protocol and different types of recognized stanzas. The types include *message*, *iq* and *presence*, each one of them having a different role.

A *message* stanza is the basic type used to send a message between two clients. They do not impose any restrictions to the content inside the tag and when XMPP is used as a normal messaging protocol, a chat client will show the content of this tag as the *chat message* sent by another client.

An *iq* stanza is one that requires a response. There is no restriction as to whom this stanza is sent but it is expected that the destination will respond to the sender with another *iq* indicating the success or failure of the operation, maybe with a result content.

Finally there is the *presence* stanza. This stanza is a special message sent to the server so that it can broadcast the new *status* to the clients.

Several extensions were made to the basic XMPP protocol, the most important for Hubiquitus is the Publish-Subscribe one. This extension uses *iq* stanzas to publish and receive items and is very popular for broadcasting purposes.

Each extension adds a new syntax inside the tags that must be followed and each one has a specification available, as is the case of the Publish-Subscribe one[5].

## B Scrum

Scrum is an agile methodology used in several companies.

In Scrum there is a backlog, a set of tasks to do called user stories organized by priority and complexity using *complexity points* (defined by the developers). From this backlog, stories are selected to be done in a defined time period called *sprint*.

This division allows a set of stories to be finished and approved at the end of small development cycles, while the complexity points and priority help decide which tasks will be done at each development stage.

Another concept used in Scrum is that of an Epic. An epic is a set of user stories, bigger than a sprint but that are related somehow. A development team can use an epic to establish bigger objectives and decide how to prioritize.

A normal development cycle is composed of an initial meeting to choose stories for the sprint and divide them in tasks to complete them. Following, a short daily meeting is done through the whole cycle to know the advancement of each person in the group and discuss problems.

At the end of each cycle, a *Sprint Review* is done to evaluate the difficulties and successes of the sprint and validate the user stories done.

# C Figures

| *Property* | JSON Format | *Description* | Mandatory |
|---|---|---|---|
| msgid | String | Provides a permanent, universally unique identifier for the message in the form of an absolute IRI. | Yes |
| chid | String | The unique ID of the channel through which the message is published. | Yes |
| convid | String | The ID of the conversation to which the message belongs. | Yes |
| type | String | The type of the message payload. | No |
| priority | Number | The message priority. | Yes |
| relevance | Date | Date until which the message is considered as relevant. | No |
| transient | Boolean | Indicates if the message must be persisted. | No |
| location | Object | The geographical location to which the message refers. | No |
| author | String | The JID of the author (the object or device at the origin of the message). | No |
| publisher | String | The JID of the client that published the message. | Yes |
| published | Date | The date at which the message has been published. | Yes |
| headers | Object | A key-value pair map with metadata. | No |
| payload | Object | Variable content of the message. | No |

Figure 3: hMessage Structure

| *Property* | *JSON Format* | *Description* | *Mandatory* |
|---|---|---|---|
| chid | String | Provides a permanent, universally unique identifier for the channel in the form of an absolute IRI. | Yes |
| chdesc | String | Topic of the channel in comprehensible form. | No |
| priority | Number | The default priority of the published messages | No |
| location | Object | The default messages geographical location | No |
| owner | String | The channel creator JID. | Yes |
| participants | String Array | A list of authorized entities JIDs to publish messages. | Yes |
| active | Boolean | If active actions on the channel are permitted. | Yes |
| headers | Object | A Headers object in the form of a key-value pair map. | No |

Figure 4: hChannel Structure

| Property | JSON Format | Description | Mandatory |
|---|---|---|---|
| reqid | String | Request ID, used for correlation purposes. | Yes |
| requester | String | JID specified if command sent on behalf of someone else. | No |
| sender | String | Command request sender's JID. | Yes |
| entity | String | JID of the destination of the command | Yes |
| sent | Date | The date at which the command has been requested. | Yes |
| cmd | String | The name of the command to execute. | Yes |
| params | Object | The parameters to pass to the command. | No |
| transient | Boolean | If false, the message will be persisted in the database if available. | No |

Figure 5: hCommand Structure

| Property | JSON Format | Description | Mandatory |
|---|---|---|---|
| cmd | String | Corresponds to the hCommand cmd sent. | Yes |
| reqid | String | Request ID, used for correlation purposes. | Yes |
| status | Number | 0 if correct, other value in case of error. | Yes |
| result | Object or Array | Optional result data sent with annexed to the object | No |

Figure 6: hResult Structure

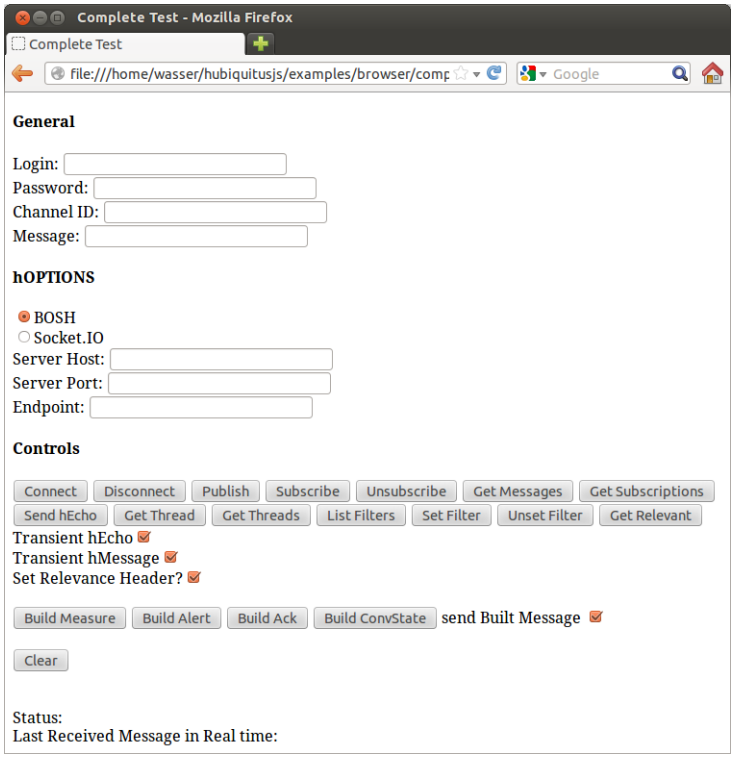


Figure 7: Screenshot of a demo client that connects to the server and executes all commands

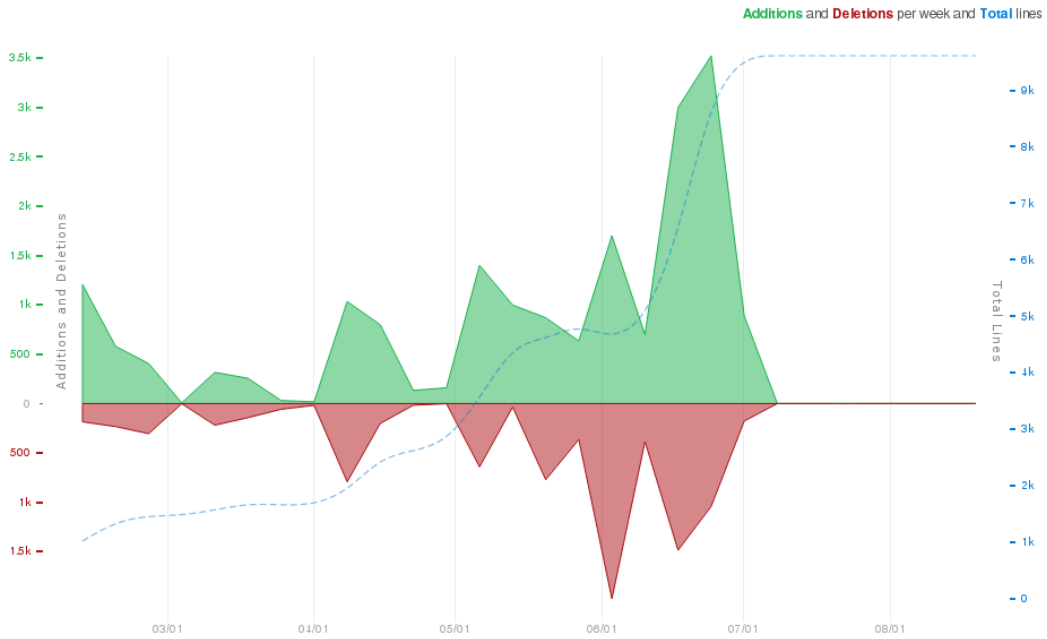


Figure 8: Graph showing code modification through the project. Green line are additions while red are deletions. The blue line is the total code lines in the project. Extracted from GitHub

# References

[1] Saint Andre. Extensible messaging and presence protocol (xmpp): Address format. http://www.rfc-editor.org/rfc/rfc6122.txt, 2012.

[2] Robert G. Chamberlain. Best way to calculate great circle distances. http://www.movable-type.co.uk/scripts/gis-faq-5.1.html, 2012.

[3] Creative Commons. Creative commons license homepage. http://creativecommons.org/, 2012.

[4] Brad Fitzpatrick. Memcached homepage. http://memcached.org/, 2012.

[5] XMPP Foundation. Publish subscribe xmpp extension. http://xmpp.org/extensions/xep-0060.html, 2012.

[6] XMPP Foundation. Xmpp introduction. http://xmpp.org/about-xmpp/faq/, 2012.

[7] GIT. Git version control homepage. http://git-scm.com/, 2012.

[8] GitHub. Github homepage. https://www.github.com/, 2012.

[9] Novedia Group. Novedia group homepage. http://www.novediagroup.com/, 2012.

[10] Novedia Group. Socialtv homepage. http://socialtv-livebattle.fr/, 2012.

[11] Holowaychuk. Mocha test framework homepage. http://visionmedia.github.com/mocha/, 2012.

[12] Eliot Horowitz. Mapreduce documentation for mongodb. http://www.mongodb.org/display/DOCS/MapReduce, 2012.

[13] JSON. Json homepage. http://www.json.org/, 2012.

[14] Movable Type Ltd. Haversine implementation in javascript. http://www.movable-type.co.uk/scripts/latlong.html, 2012.

[15] Node.JS. Nodejs homepage. http://www.nodejs.org/, 2012.

[16] RallyDev. Rallydev homepage. http://www.rallydev.com/, 2012.

[17] Guillermo Rauch. Socket.io homepage. http://socket.io/, 2012.