hawthorn CHAT

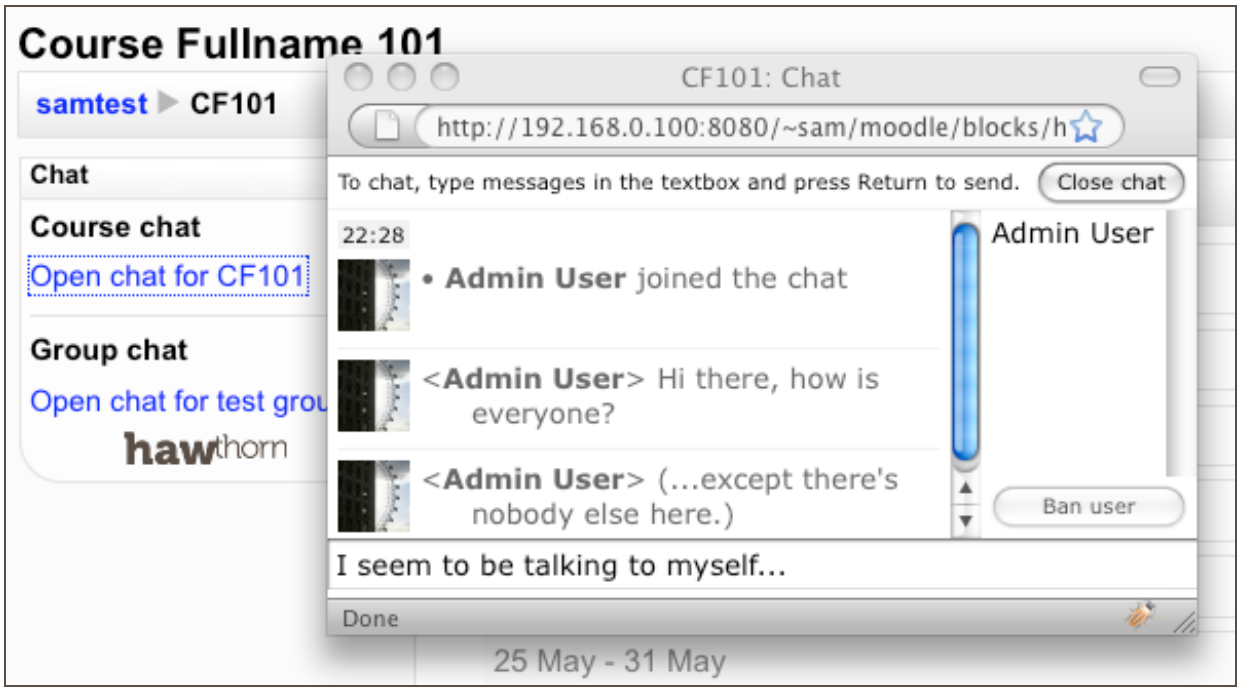
System manual

Contents

Introduction

What is Hawthorn?

Hawthorn is a JavaScript-based text chat system. Here's a screenshot of Hawthorn running within the Moodle educational environment.



Basic requirements

In order to use Hawthorn, you need:

- An existing website with its own user authentication system.
- A server that can run Java applications.

Audience

This document is intended for:

- System administrators who intend to install Hawthorn on their system.
- Developers who intend to implement a Hawthorn connector for a host system.

This is not an end-user document. Technical knowledge and system administration skills are required.

Terms

This glossary lists basic terms used in this manual and in the Hawthorn system. You don't need to understand all this before reading the rest of the manual.

- **User**
  Somebody who can connect to the Hawthorn system. A user is identified by a unique *user name*.
- **User name**
  A short text string consisting only of ASCII characters A-Z, a-z, 0-9, - and _. Not usually displayed on-screen to users.
- **Display name**
  A text string containing arbitrary Unicode characters (except control characters and the " symbol) that represents the user. This is often displayed to users.
- **Extra data**
  Extra data stored per user, defined by the host system. This could include information that will allow a user picture to be displayed, or other integration details. In purely standard Hawthorn usage, it is an empty string.
- **Permissions**
  User permissions defined by a text string such as `rwma`. Available permissions are read, write, moderate, and admin.
- **Channel**
  A channel or chat room in which users can talk. Hawthorn channels are completely independent of each other.
- **Channel name**
  A short text string consisting only of ASCII characters A-Z, a-z, 0-9, - and _. Not usually displayed on-screen to users. Hawthorn does not maintain user-visible channel names; this is the responsibility of the host system.
- **Key**
  An authorisation key (sequence of hexadecimal digits) that permits access to a Hawthorn channel for a particular user. Created using the SHA-1 hashing algorithm.
- **Key time**
  The time at which a key expires, stored as a number of milliseconds since Jan 1, 1970. Hawthorn keys are granted for a limited time.
- **Host system**
  An existing Web system that provides users with access to the Hawthorn chat system.
- **Connector**
  Program code implemented within the host system (not Hawthorn) that provides links and files that give users controlled access to Hawthorn.

Hawthorn is...

Hawthorn is a text-only Web chat system.

It is based on the following design decisions and principles. If these do not sound appropriate for your usage, then another system might be better for you.

Integrated

Hawthorn cannot be used on its own. It must be integrated into another Web system referred to as a **host system** which provides user information and authentication, and determines which channels are available.

There is no need for ‘single sign-on’ support for Hawthorn; it has no other way to operate.

Goals

- Simple for programmer to implement connectors on most Web systems and major platforms.
- Example code for (initially) JSP and PHP.
- Fully-working connector for (initially) the Moodle educational system.
    - Additional connectors to be developed and contributed by third parties.

Built to perform

Hawthorn is designed to perform as well as possible.

Goals

- High performance of Hawthorn infrastructure.
    - Typical request handling in the 1ms range.
- No performance drain on linked host system.

Simple

Hawthorn does not have many features. It is designed to provide basic text chat facilities, not to compete with major chat systems such as IRC, Jabber (XMPP), or closed systems such as MSN or Skype.

Goals

- Basic group chat in channels and nothing else.

Easy to use

On the client side, Hawthorn is implemented entirely in JavaScript. This allows operation on major browsers without the requirement for any plugin.

Goals

- Operate correctly on browsers that cover a large majority of Web users.
  - All modern browsers (latest versions of Firefox, Safari, Chrome, and Opera).
  - Internet Explorer 6, 7, and 8.

Transient

Hawthorn does not provide guaranteed message delivery or long-term history. In order to achieve high performance, all data is stored in memory.

Goals

- In-channel data is stored only for minutes in memory.
  - Log files can contain all chat messages for review in case of user disputes or other audit requirements.

Enterprise-class

Hawthorn can cope with a large number of users and can be designed to be robust in the event of system failure.

Goals

- Flexible server configuration.
  - Single-server basic system where redundancy is not required.
  - Redundancy through linked live servers.
- Infinite scalability.
  - Because Hawthorn channels are completely independent, you can divide channels between multiple unconnected Hawthorn servers.
- Simple server management.
  - No data is stored except for configuration and system logs.
  - There is almost no backup requirement.
  - A replacement server can be configured in minutes.

Architecture

Hawthorn has a simple architecture that is designed to be *loosely connected*. There is no database and nothing to configure. All information is held in memory.

Overview

Hawthorn systems combine three components:

- The **Hawthorn server**, which is a Java-based server that responds to HTTP requests. This provides all chat functionality.
- An existing Web-based **host system**, which can be implemented in any language, does not need to connect to the Hawthorn server, and does not need to be based on the same machine or network. The host system provides users with links and authentication keys that can be used to connect to the Hawthorn server.
- Client-side **JavaScript**, which is delivered by the host system and can be customised if required. This communicates with the Hawthorn server to implement the chat system.

Communication

JavaScript ↔ Hawthorn server

The JavaScript code communicates with the Hawthorn server using a method often referred to as **JSON**.

- The user's browser initiates communication by creating a JavaScript `<script>` tag inside the current web page.
- The address of this script is a URL that points to the Hawthorn server, including a specific command *verb* and parameters such as the key and user name.
- Hawthorn responds to the request using standard HTTP. It outputs JavaScript code which includes parameters that describe the result of the request.
- The user's browser runs this code, which informs the main JavaScript that a response has been received and can be processed.

Unlike other methods of JavaScript communication, this allows the Hawthorn server to be located at a different URL from the server that delivered the page.

Host system ↔ Hawthorn server

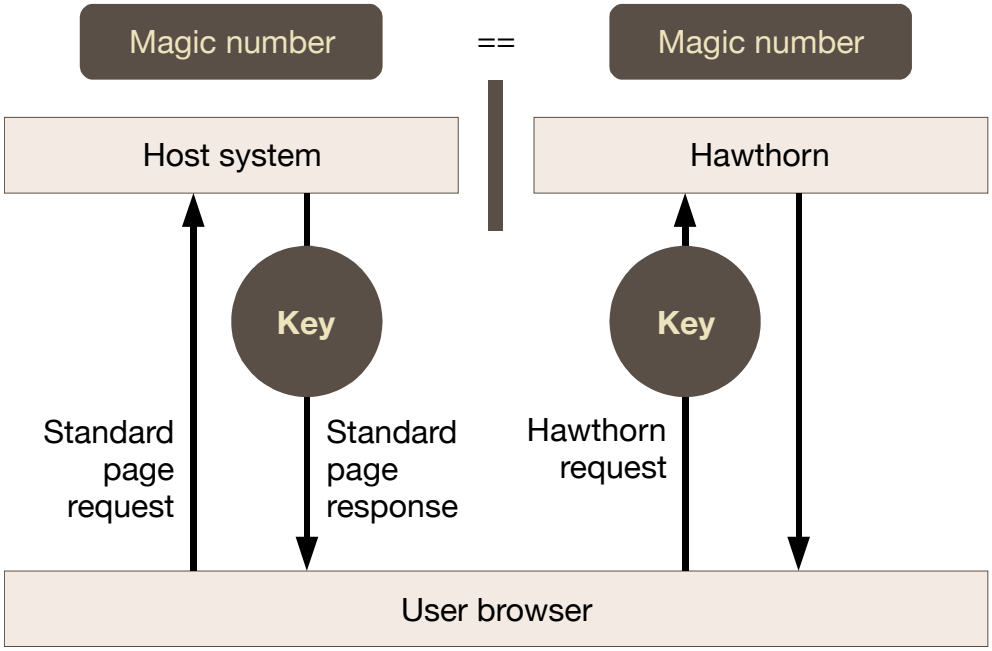
There is no communication between the host system and the Hawthorn server.

Hawthorn server ↔ Hawthorn server

When Hawthorn servers communicate, they use a custom format over standard TCP/IP.

Users and authentication

Hawthorn does not store any information about users. User authentication is managed via a **magic number** and SHA-1 hash keys. The magic number is shared between the Hawthorn server (in its configuration file) and the host system (likewise, somewhere in its configuration).



Authentication process

- The host system authenticates the user for normal access to its Web pages.
- When the authenticated user wants to access Hawthorn chat, the host system generates a **key** which is provided to the user's browser.
  - The key is a hash of the magic number, user name, display name, channel name, and an expiry time.
- Via JavaScript, the user's browser communicates **directly** with the Hawthorn server to initiate chat.

Because the host system controls key generation, it can allow users access to Hawthorn only if they are authenticated – and only with a user name, display name, extra details, permissions, and on a channel that the host system approves.

While chatting, all communication takes place directly with the Hawthorn server using the generated key for authentication. This ensures there is no performance impact on the host system.

The key lasts for a certain time (usually 1 hour) so that if a user's access is revoked by the host system, they cannot continue to use Hawthorn chat. A **re-acquire** process lets the user's browser request a new key from the host system if a chat session lasts longer than the key expiry time.

Host performance

This process ensures that there is no significant performance drain on the host system, compared to a chat system directly integrated as part of the host system.

With a 1 hour expiry time, the host system has to support no additional Web requests at all if a chat session lasts under 55 minutes, and 1 request per 55 minutes if the user stays in the chat session for longer.

A 90 minute chat session would require 1 request from the host system. Meanwhile, the Hawthorn server will service approximately 600 requests. If chat is a popular service, handling 600 requests per user directly (in a host system that is not primarily designed for the purpose and may already be under heavy load) might have serious performance implications.

User names

The host system can determine user names, but these may only contain the characters `A`-`Z`, `a`-`z`, `0`-9, `-` and `_`.

If the host system already has user names (or numbers) of that form, they can be used directly. If host user names do not directly follow the same restriction, they must be converted before creating Hawthorn keys. For example, arbitrary Unicode characters and _ could be converted into _ followed by the 4-digit hexadecimal Unicode representation.

Some organisations treat user names as 'secret' information. For this reason, only users with *moderate* permission (see below) are sent the actual user names of other users. Ordinary users receive hashed versions of user names beginning with the `?` symbol.

At present the current JavaScript implementation does not display user names, but they are required when a moderator chooses to ban another user.

Permissions

A user can be granted permissions by the host system. These permissions are included in the data used to create the authorisation key.

Permissions are defined by a string containing single letter codes. The codes must be listed in order. Available codes, and the verbs (see below) they grant access to, are:

- `r` (read) – able to read messages from the channel (`recent`, `poll`, `wait`).
- `w` (write) – able to write messages to the channel (`say`).
- `m` (moderate) – able to moderate the channel (`ban`).
- a (admin) – able to view channel logs (`log`).
  If granted on the special system channel `!system`, this also allows viewing of system logs and of current system statistics.

Extra data

If a host system needs to include additional user details in chat (e.g. a picture URL), these can be encoded into per-user *extra data*. The connector for the host system needs to add these details and customise the chat JavaScript to make use of the extra data.

Channels

Hawthorn supports an unrestricted range and number of channels. A channel is a defined location in which a group of users can chat.

In the current version of Hawthorn, everything said in that channel is visible to everyone in that channel.

Channel creation and deletion

Channels do not have to be created. Providing a user with a key for a particular named channel will allow them to enter that channel. If a second user is given access to the same channel, the two users can chat.

Just as channels do not need to be created, they do not need to be deleted. After all users have left a channel and all messages have expired from that channel (see below), the channel itself will be removed from memory automatically.

Channel names

The host system can determine channel names, but these may only contain the characters A-Z, a-z, 0-9, - and _. If these names are required to represent existing concepts within the host system which use a wider character range, then they must be converted as per ‘User names’ above.

Message lifetime

The message lifetime can be defined in configuration (see page 30). By default it is 15 minutes. After this time, messages are removed from memory.

The actual history displayed to users when they enter channels is limited to a specified number of messages (usually 10).

Hawthorn does not provide persistent message storage and is designed as a transient chat facility. If a server needs to be restarted, all stored messages are lost.

Channel management

Users with ‘moderate’ permission (`m`) are able to ban users from a channel. When banned, users cannot connect to that channel for a period of time.

Bans are usually short-term and are also transient like other Hawthorn data; they will be lost if a server is restarted. Hawthorn does not provide longer-term or stored bans. Permanent action against users should be taken within the host system, so that the host system will not grant them Hawthorn keys in future.

Because users are identified by the host system and are not anonymous, it is hoped that they will be discouraged from inappropriate behaviour.

Auditing

By default, Hawthorn logs everything spoken in any chat channel. There is a separate log file for each channel, organised by channel name. Because the log includes the user name determined by the host system, this can be traced back to the user's identity in the host system.

There is more information about logs on page 33.

Browser behaviour

The user's browser works together with the Hawthorn server to provide a chat session.

Verbs

User browsers communicate with the Hawthorn server by issuing commands known as *verbs*. Key verbs are:

- `recent` – get recent messages from a channel.
- `say` – say something on a channel.
- `poll` – get messages (if any) that have appeared on a channel since the last `poll`.

For a full list of verbs and parameters, see page 38.

Communication sequence

A user chat with Hawthorn generally follows this sequence:

1. User visits a Web page on the host system with information about a Hawthorn channel, perhaps including recent messages obtained via the `recent` verb.
2. User clicks the provided link to open a chat window. This uses the `recent` verb to obtain a more detailed list of recent messages as well as a list of people currently present in the channel.
3. Every few seconds, the user's browser uses the `poll` verb to check if there are new messages. The response also tells the browser when to check next (soon, if the channel is active; after a longer delay, perhaps 15 seconds, if it is inactive).
4. If the user types a message, this is sent using the `say` verb.

Deployment

This chapter describes how to deploy a Hawthorn system. It covers:

- **Server configurations** – how to lay out one or multiple Hawthorn server machines.
- **System requirements** – infrastructure configuration for each individual machine.
- **Installation** – how to install a Hawthorn server on an individual machine.

Throughout this chapter, some options are noted as [**recommended**] or [**not recommended**], or neither. You are welcome to deploy Hawthorn in a manner that is not recommended; the recommended options might provide an easy option if configuring servers from scratch. The reasoning for each recommendation is given as a footnote.

Server configurations

Single server [recommended[1]]

You can use a single Hawthorn server. This simple configuration is easy to set up. Hawthorn is a high-performance system, so one server may be sufficient to meet your performance requirements. The system will become unavailable if you have a hardware or network failure on the machine.

- Log files should go to a local disk. System log files are required. Chat logs are optional.
- If you expect low usage, you can configure the server on the same machine as another Web server using a different port.

Multiple linked servers [recommended[2]]

You can link two or more Hawthorn servers. When two servers are linked, users can connect to channels on either one and will see the same messages.

- The servers must be able to connect to one another on the same server port.
- Two linked servers offer more performance than a single server, but not as much as two independent servers; linking servers reduces performance. If you link servers, it is probably best to use as few as possible, and not more than three.
    - Chat systems that expect a large number of servers typically use an hub/leaf architecture where most servers only need to communicate with one other server, and messages are not passed on.
    - Hawthorn is not built this way. Every server connects to every other server. If you require high capacity, the solution is to use independent Hawthorn systems (see below).

1 Basic install suitable for small system that does not need high availability.
2 Efficient way to achieve availability with simple installation and best chance of seamless server changeovers.

- Hawthorn's JavaScript system can automatically select between the servers. It selects a server at random. If one server is not available, it can pick the other one.
    - This random selection has an equal chance of picking each server. Consequently, all servers should be of equal capacity.
    - A failure that occurs in the middle of a chat session will usually not be apparent to the user. When any request fails, it will automatically be transferred to the other server. Users only see an error if all servers fail.
    - If network connections do not report failure, but hang indefinitely, there is a time out (currently 20 seconds) after which a request will be treated as failed and passed to the other server.
    - Hawthorn does not offer guaranteed integrity. It is possible that some user messages might not get through during a failure, without any error display. However, the system does avoid duplicated messages.
- If your two servers have different power and network infrastructure, this provides for a fault-tolerant system without the need for any extra hardware or configuration.
- When a second server comes back up, it does not acquire message history from the first server; Hawthorn data is transient. The second server begins with no data and only receives messages from that point on.

Fail-over servers [not recommended[3]]

If you have front-end network hardware it may be possible to use Hawthorn in a fail-over configuration. In this configuration, two Hawthorn servers are set up independently of each other (not linked). The front-end hardware directs all requests to one of the servers. If that server fails, requests are directed to the other. At all times, one server is in use, and the other remains idle and ready.

- This relies on suitable front-end hardware. The front-end hardware needs to itself be fault-tolerant if you require a fault-tolerant system.
- Because there is no connection between Hawthorn servers performance is identical to a single server.
- Hawthorn's JavaScript would be given only a single server address in this case.
- At the point of fail-over, all message history will be lost. Unless the fail-over is instant, users may see error messages and need to reconnect.

3 Inefficient for two reasons: one server is wasted, and there's an additional piece of hardware in the network path. Harder to install because the front-end server must be configured.

Load balancing [not recommended[4]]

You can also use load-balancing hardware in front of a pair of linked Hawthorn servers (rather than relying on the JavaScript load balancing).

- You may be able to use more intelligent load balancing that takes into account server load to better balance demand on the servers.
- The JavaScript system is given a single address. It cannot itself select between the servers. If the load balancer directs it to a server that has failed, users will see an error. Users will not be able to reconnect until the load balancer realises that the server is down.
- As HTTP servers, Hawthorn servers are extremely limited. If the load balancer provides features that assume a full HTTP server, these will not work correctly and should be disabled.

Multiple independent systems [recommended[5]]

In very heavy loads, more than one Hawthorn server might be required simultaneously. You can use a linked system of two or three servers, but beyond that the system is unlikely to scale well.

The solution is to divide your channels into two separate groups. Each group will be served by different Hawthorn server(s). If you require these servers to be redundant, each group will need a pair of servers configured in one of the manners described.

- A simple way to divide channels would be to take a numeric hash of the channel name. If this is odd, count it in one channel group; even is the other one.
- You could also make the host system allocate channels in a different way.
- The provided connectors do not currently include support for multiple channel groups, although the JSP and PHP libraries remain usable. You might need to implement a custom connector.

4 Less efficient due to additional hardware; requires extra configuration.
5 Provides infinite scalability for very large systems.

System requirements

Operating system

Hawthorn can run on Linux [**recommended**[6]], Mac OS X[7], or even Windows[8], as well as other platforms. It is likely to run acceptably on any platform.

Hawthorn can run in 64-bit mode [**recommended**[9]] or 32-bit. Performance differences will be minimal.

Java

Hawthorn is a pure Java application requiring Java 5 or Java 6 [**recommended**[10]].

CPU

Hawthorn performance is limited by CPU power. A fast CPU will be an advantage.

Hawthorn is able to scale to at least two processor cores. It may scale successfully to four or more, but this has not been tested. There is a potential limitation because key network operations (accepting new connections, reading request data) are currently handled on a single thread. (See the information about MAIN_THREAD_BUSY in the server statistics, page 36.)

If your server has many processors, you should carry out load testing to ensure that Hawthorn makes efficient use of them. It may even be beneficial to run multiple Hawthorn servers (linked or, ideally, serving different channels) on the same machine.

Disk

Apart from the application binary and its configuration file, Hawthorn only uses disk space for log files.

Hawthorn needs write and read permission to a single log folder, and read permission to its binary and configuration file.

More information about logs can be found on page 33.

Memory

Hawthorn does not have large memory requirements. Any modern machine should have plenty of memory.

6 Standard server operating system with high-performance multithreading and networking.
7 Reputed issues with multithreading. Not widely used as server operating system.
8 Reputed issues with network performance. Configuration is different to other operating systems.
9 64-bit mode offers slight performance improvements (not just greater memory addressing capability) when using Intel processors. On other processor architectures it shouldn't make any difference.
10 Java 6 offered significant improvements to performance for server applications. Later Java 6 updates may offer additional performance gains.

Network

Hawthorn works via HTTP connections to a single server port.

The default port is 13370 [**recommended**[11]].

Because Hawthorn is designed to handle many hundreds of connections per second, your system kernel's HTTP performance may be more critical than usual, and more system effort may be spent initiating and closing connections.

Firewall configuration

Configure your organisation's firewall to allow access:

- From the public Internet to the Hawthorn port (TCP).
- Between any linked servers, also on the Hawthorn port (TCP).

Port 80

You may wish to configure this to port 80 in order to avoid any possible firewall problems, although this generally ought not to be necessary as Hawthorn works through most Web proxies.

Running on ports below 1024 generally requires running Hawthorn as root [**not recommended**[12]]. It may be possible to use the Apache Foundation's `jsvc` utility to initialise Hawthorn as root, then reduce its privileges; this has not been tested.

11 There's no particular reason to change it.

12 Running servers as root means that a bug in the server could allow attackers full access to the machine through various mechanisms. Hawthorn is unlikely to suffer from this type of bug, but it's better not to take the risk.

Installation

Installation requires the following steps:

- Create a Hawthorn user and group.
- Install the `hawthorn.jar` program file.
- Create a folder for logs.
- Create the configuration file.
- Run Hawthorn to test that it works.
- Set up your machine to run Hawthorn automatically.

Note for Windows users: Windows does not support standard POSIX permissions, so the security procedures and commands described here do not apply. Please use Windows security features to implement similar security restrictions.

Create Hawthorn user and group

For security reasons you should create a user account for Hawthorn. We suggest you call this user account `hawthorn`. Only the Hawthorn server will use this user account. It will have permission to read the Hawthorn configuration file and write Hawthorn logs.

You should also create a group. We suggest you also call this group `hawthorn`. Users in this group will have permission to read the Hawthorn logs.

All other users will not have permission to read or write the logs and configuration file.

- *Linux*:
  Create a group called `hawthorn` and a user called `hawthorn`, which is a member of that group. The user should be configured so that you can't actually log in with it from the console. Procedures vary for different distributions, but the `useradd` command may be the one you need.
- *Mac OS X 10.5*[13]:
  ```
  sudo dscl . create /Users/hawthorn
  sudo dscl . create /Users/hawthorn RealName "Hawthorn server"
  sudo dscl . create /Users/hawthorn UniqueID 493[14]
  sudo dscl . create /Users/hawthorn PrimaryGroupID 493
  sudo dscl . create /Users/hawthorn UserShell /usr/bin/false
  sudo dscl . create /Users/hawthorn NFSHomeDirectory /var/empty
  sudo dscl . create /Groups/hawthorn
  sudo dscl . create /Groups/hawthorn PrimaryGroupID 493
  ```

13 Apple keep changing the way this works. If using a different version, try Google for instructions.
14 The number 493 is an arbitrary value less than 500. If it is taken, try another one.

Install program file

The program file `hawthorn.jar` is provided in the standard download. This is the only program file you need to deploy.

- Install this file in a suitable location. For example purposes, these instructions assume it's `/UnixApps/hawthorn/hawthorn.jar`.
- Make sure it is readable by the `hawthorn` user. Since it is publicly available, you might as well make it readable by all users.
  ```
  sudo chmod a+r /UnixApps/hawthorn/hawthorn.jar
  ```

Create a folder for logs

Hawthorn needs a folder for its logs. You should create a new folder that is not used by other applications. If you run multiple Hawthorn instances, these must use different folders.

The folder can be local disk [**recommended**[15]], or in another file system location such as an NFS-mounted share.

- Create the log folder. For example purposes, these instructions assume it's `/UnixData/hawthorn-logs`.
- Change its owner to the `hawthorn` user, and group to the `hawthorn` group. Set permissions so that the user can read and write, and the group can only read. (The x permission is also required to access the folder.)
  ```
  sudo chown hawthorn:hawthorn /UnixData/hawthorn-logs
  sudo chmod u=rwx,g=rx,o= /UnixData/hawthorn-logs
  ```

15 Local disk gives very high performance for log writes, although Hawthorn log writes are buffered so it probably doesn't make much difference..

Create configuration file

Hawthorn uses a single XML configuration file. This file includes the magic number that provides authentication information, so it's important to keep it safe. It also contains other configuration settings. Here's a minimal configuration file:

```
<hawthorn>
<logfolder>/UnixData/hawthorn-logs</logfolder>
<magicnumber>23d70acbe28943b3548e500e297afb16</magicnumber>
<servers><server this='y'>192.168.0.100</server></servers>
</hawthorn>
```

- Save the configuration file in a suitable location. For example purposes, these instructions assume it's `/UnixData/hawthorn-conf/config.xml`. You can use the same folder as for logs if you prefer.
- Set the log folder and server address.
- Enter a new magic number (hexadecimal, 40 digits recommended). If you want Hawthorn to make one up for you, delete the `<magicnumber>` section entirely and then run Hawthorn with this unfinished configuration file. It will suggest a suitable one. **Do not use the example above on a production system**. The magic number must be kept secure as it would allow users to impersonate any other user and obtain unrestricted server access.
- Change the file's owner to the `hawthorn` user and group, and set permissions so that nobody but this user can read the file.
  ```
  sudo chown hawthorn:hawthorn /UnixData/hawthorn-conf/config.xml
  sudo chmod u=r,go= /UnixData/hawthorn-conf/config.xml
  ```

Other configuration options are available. See page 30 for a full list. In particular, if you want to route Hawthorn requests through a proxy or load-balancer, see page 22.

Check Java version

From the command line, run `java -version`. This reports the Java version in use. If you have multiple Java versions, ensure that Hawthorn is using the appropriate version (Java 6, reported as 1.6). You might need to give a specific path to Java if it is using the wrong version.

Run Hawthorn

To test that it's working correctly, you can run Hawthorn from the command line. Hawthorn takes a single command-line parameter: the location of the configuration file. You should also include Java configuration parameters:

- `-server` selects the server VM[16]. The server VM performs better than the client VM, which is default on some platforms.
- `-d64` selects the 64-bit VM. If you are using a supported 64-bit processor and operating system, add this flag.
- `-Xmx256M` sets the memory limit. The default is often 64 megabytes, which is rather low. Required memory depends on the `historytime` Hawthorn setting (default 15 minutes) and the quantity of usage. You can determine the required memory by load-testing.
- `-Xms256M` causes Java to claim the memory immediately rather than waiting until it is required. This can save memory-allocation time later, and makes the system's physical memory usage more consistent.

In addition, you should run Hawthorn as the `hawthorn` user. Doing this while testing lets you ensure that file permissions are set correctly. On most platforms you can do this by adding `sudo -u hawthorn` to the start of the command line.

Here is an example command line:

```
sudo -u hawthorn java -server -d64 -Xmx256M -Xms256M -jar
/UnixApps/hawthorn/hawthorn.jar /UnixData/hawthorn-conf/config.xml
```

- Run Hawthorn.
- Using another terminal window, check in the Hawthorn log folder. You should see a log file.
- Look at the log file to make sure it shows startup information and doesn't indicate any errors.
- Kill the Hawthorn process (for example by pressing Ctrl-C in its terminal).

Make Hawthorn run automatically

In order to deploy the server, you need to make it run Hawthorn automatically so that Hawthorn will come up if the machine recovers after losing power.

This script should launch Hawthorn using the `hawthorn` user account. You will need to include a mechanism to restart Hawthorn in case of problems or if you want to change the configuration file (which is only read on startup).

You can write this script yourself, or use the examples included in operating system sub-folders of the `bin` distribution folder.

16 The server VM may not be available by default on 32-bit Windows installs. If this is the case you will get an error. To get access to the server VM, install the latest Java 6 JDK and use the Java runtime installed in that.

Trying out your server

Now that you have installed the server, you need to put in place the systems that will use it for chat.

Testing with pure HTML

If you add 'test key' information to the configuration file and manually launch the server, you can test using a basic HTML page that can be loaded on any computer. See `readme.txt` in the `connectors/htmlexample` folder of the distribution.

Testing with JSP or PHP

If you have a JSP or PHP installation, look at the `readme.txt` in the relevant `connectors/` example project. These example projects cannot be used in production environments, but allow you to conveniently test the server.

Installing a real connector

At time of writing, the only system with a provided connector is Moodle.

To install the Moodle connector (or another one if more are provided later), see the instructions in the `readme.txt` in the relevant `connectors/` folder.

Writing your own connector

If you need to write your own connector, get started by trying one of the example connectors mentioned above first. The connector development guide, provided in the distribution as `connectordevelopment.pdf`, documents the process of creating your own connector..

Load testing

If you are expecting any serious load on your server, you should carry out load testing. Hawthorn comes with a built-in load testing usage simulator that you can use for this purpose, or you can use standard load testing software such as Apache JMeter.

See page 23 for full load testing instructions.

Finished

That's it – your server is installed.

Thanks for trying Hawthorn. I hope it's useful for you.

Reference

This section contains detailed information that was summarised earlier in the manual.

Proxies

It is possible, but not recommended, to run a Hawthorn system behind a proxy or load balancer.

Logging client IP addresses

Most proxies send the request to Hawthorn with an additional header that includes the client's genuine IP address. Otherwise, Hawthorn would log all requests with the proxy's address.

This header is usually called `Client-IP` or `X-Forwarded-For`. Set up your proxy to provide the header, then set the name of the header in Hawthorn's configuration file:

```
<ipheader>Client-IP</ipheader>
```

Multiple servers

If you run multiple connected servers, the server-to-server connections cannot go via a Web proxy or load balancer. Even though they go to the same port, server connections are not true HTTP connections and cannot be proxied.

Your server settings in the configuration file must point directly to the real server machines.

Load testing

Before using a proxy for a live Hawthorn system, ensure that load testing is done through the same proxy or (if the proxy is in use already) an equivalent test proxy. This will help you diagnose any problems that might apply, and ensure that your load testing is otherwise realistic.

If the proxy handles other systems, you may wish to run load testing for the expected level of load via the live proxy while monitoring its performance, to ensure that the expected level of Hawthorn usage does not degrade your proxy's performance.

Load testing

You should load test your Hawthorn system to determine the approximate user load that it can handle. If this load is below, or close to, the load you expect in real use, you should increase the performance of your Hawthorn server(s).

Test system

You need one or more computers to run a load tester. Do not use the server computer as a test system.

The server you are testing should be your live (but not yet deployed) system, or an equivalent system. Do not test a system while it is actually in use, unless you don't mind if it falls over!

A note about TIME_WAIT

Hawthorn relies on extremely short-lived connections. TCP restrictions mean that after it is closed, each connection consumes a port number on the **client** machine for up to 4 minutes, a time known as TIME_WAIT.

A single system is limited to approximately 64K ports, although typically only around 20K ports are actually used for these temporary port assignments. If 20K ports are used and TIME_WAIT is 4 minutes, this means it can support 20K connections per 240 seconds = ~80 connections per second.

This restriction does not apply to the server. Consequently it is unlikely to be a problem in real use, but it can get in the way of load testing.

While load testing, the TIME_WAIT limitation results in two possible outcomes:

- 'Lumpy' processing where a queue builds up, stays there for some time with all threads in C (connecting) status, then eventually clears. If you monitor CPU usage, this should be visible as obvious peaks and troughs.
- Exceptions: `NoRouteToHostException: Cannot assign requested address`.

You can verify that this is the problem by trying to use a Web browser on the client computer immediately after this exception occurs. Visit a common site, such as Google. The web browser will fail to connect because no temporary outgoing port is available.

Hawthorn can deliver performance far in advance of 80 connections per second, so a typical system is not adequate for use as a load testing client without configuration. The following solutions are available:

- Use more than one client computer to achieve the required test load while not going beyond the TIME_WAIT limitation.
- Temporarily configure TIME_WAIT to achieve a larger number of connections from a single computer. (This is usually more practical.)

Configuring TIME_WAIT on Linux

Remember: make this configuration change on the load-testing client machine(s), not the server!

Configure TIME_WAIT using `sysctl` as follows:

```
sudo sysctl -w net.ipv4.tcp_fin_timeout=5
```

If this doesn't work, search for documentation on your specific Linux distribution.

Remember to set it back to the previous value before you try to use the computer for Internet connections.

Configuring TIME_WAIT on OS X

Remember: make this configuration change on the load-testing client machine(s), not the server!

On OS X you cannot configure TIME_WAIT directly, but you can configure the 'MSL' time, and TIME_WAIT is expressed as a multiple of this time.

```
sudo sysctl -w net.inet.tcp.msl=1000
```

This short time is suitable only for local networks. Make sure to set it back (the default is 15000) before you try to use the computer for Internet connections.

If you need more connections, you can also configure the port range used for these temporary ports. The default starting port is 49152. You could supply a larger range such as:

```
sudo sysctl -w net.inet.ip.portrange.hifirst=20000
```

This makes less difference so is unlikely to solve the problem on its own.

Configuring TIME_WAIT on Windows 2000/XP[17]

Remember: make this configuration change on the load-testing client machine(s), not the server!

1. Open the registry editor (Start / Run / `regedit`).
2. Navigate to `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters`.
3. Look for the value name `TcpTimedWaitDelay`. If it doesn't exist, right-click on a blank area, choose New / DWORD value, and type that name.
4. Right-click on the value name and choose Modify, then type a suitable value such as 5.

This short time is suitable only for local networks. Make sure to set it back (the default is 240 decimal; or if the value didn't exist before, you could delete it again) before you try to use the computer for Internet connections.

17 http://technet.microsoft.com/en-us/library/cc938217.aspx

LoadTest simulator basics

Hawthorn comes with a simulator called `LoadTest`. This simulator is able to imitate real usage. It creates a large number of users who connect to the server, poll for messages, and say things.

Minimal parameters

To run a simulation, you need to decide:

- The number of simulated **users** who will have a chat window open (and possibly be talking in it) at any one time.
- The number of **drive-bys** per minute. Drive-bys are visits to other pages that include a Hawthorn `recent` call to display recent chat messages in a channel: the user isn't actively chatting, but a single request will be made to the Hawthorn server for each visit to that page.

If you know which pages of an existing system will include Hawthorn recent calls, you may be able to use your present Web logs to estimate a good peak value for drive-bys. Unfortunately, there will probably be a high degree of guesswork involved in the more important **users** parameter.

Testing strategy

There are many possible testing strategies. One would be:

- Estimate the parameters you really expect.
- Do a simulation run with these settings.
- If the server copes, double both parameters and try again. Repeat until the server can no longer cope.

Running a test

Once you've decided on your parameters and testing strategy, here's how to run an actual test. (You might like to try this with very small values first just to make sure it's working.)

1. Install `hawthorn.loadtest.jar` (from the `bin` folder of the distribution) onto your load testing client machine.
2. Make sure you have configured TIME_WAIT on the client machine as above.
3. On the client machine, run a command line like the following. Include your server's own host IP address and magic number, along with the parameters you chose.
   ```
   java -jar hawthorn.loadtest.jar host=192.168.0.1
   magicnumber=23d70acbe28943b3548e500e297afb16 users=1000
   drivebys=100 > loadtest.1000.100.csv
   ```
4. You should see screen output immediately. (Final output was redirected to a CSV file, but the test displays information as it runs.)
5. **Monitor CPU usage on the client machine**. If the CPU nears 100% after the warm-up period is complete, then you cannot run that many test users on the machine, so **the test is invalid**. Exit from the test (Ctrl-C) and decrease the number of users. You will need to use multiple client machines.
6. Monitor the displayed test information (see below). If the Queue figure gets high, that indicates that the system is unable to push events through fast enough. That probably indicates that a load limit has been reached; however, if the value isn't continuously increasing, you may be able to get more accurate results by using more threads in the load tester (add `threads=30` to the command line).
7. Monitor the server machine to determine how it is handling the load. During the test, take a look at CPU usage and at the server statistics (see page 36) to see how well it's holding up. (USER_REQUEST_TIME and MAIN_THREAD_BUSY are significant here.) You may also want to keep an eye on network traffic to see what additional traffic you could expect as a result of this usage.
8. The test lasts for five minutes. When it's finished, examine the report in the CSV file listed above. The CSV file contains information about the load test (time, parameters) and a single line at the end with the actual results.

Information displayed during test

During the test, a line of information is displayed every 5 seconds. This includes:

- Time since start of test, in seconds.
- Total number of events sent so far.
- Total number of errors and exceptions (should both be 0).
- Current queue size. If all test threads are waiting, the queue will grow.
- Mean event time so far (averaged over the whole test, not a current figure).
- Threads. Each thread is shown as a `.` (inactive), `C` (connecting), or `R` (receiving).

If the Queue figure gets high, that indicates that the system is unable to push events through fast enough. That probably indicates that a load limit has been reached; however, if the value only appears occasionally, you may be able to get more accurate results by using more threads in the load tester (e.g. add `threads=50` to the command line).

When there is a large queue, there might be a delay in stopping the test because the test doesn't stop until it completes all events that were scheduled to happen before the scheduled test completion.

Interpreting the results

You should expect a 0 in the **Errors** and **Exceptions** columns. If the number is not 0, make sure your machines are correctly configured (they can connect to each other, the magic number is correct, and so on).

The **Event mean** column includes the mean time it took to process an event, in milliseconds. An acceptable average time might be in the 10 millisecond range.

Note: The reported time will be higher than the time the server reports in its statistics pages. The server processing time does not include network transmission delays or the network connection overhead; and if the server is heavily loaded so that MAIN_THREAD_BUSY gets high, it won't be able to start timing at the right point either! Under normal conditions the server reported times more accurately reflect the server's effort in handling a request, but the load test times more accurately reflect a potential user experience.

A server has probably not reached load limits if **all** of the following are correct:

- Server CPU usage did not hit 100% during the test (or only for brief periods due to other system activity).
- The **Event mean** load test result column contains an acceptable value, such as 20ms.
- The **Seconds** result column shows the expected number of seconds (300 for a five-minute test).
- The **Queue mean** load test result column, indicating inaccuracies of the test, is near zero. (See example below.)
- The MAIN_THREAD_BUSY_PERCENT value in Hawthorn's statistics (see page 36) did not approach 100. (If you need to check it after the test, this per-minute value is available in the system logs; look at the values covering your test period.)
- Hawthorn performance during the test remained acceptable – for example, the Statistics page appeared quickly when viewing from another machine.

Watching the load test 'live'

The load test happens on channels with names like `loadtest2`. If you connect to one of these channels while the load test is happening, you can get an idea of what's going on. The example JSP web application provides one way of getting access to a load test channel.

Multiple load test clients

If your live servers are faster than the machines you have available to run load tests (or just because the load tester is not as efficient as the server), you will need to run multiple load test client machines in order to generate enough load.

To run multiple clients, simply set them both up as above. Get ready to run each test, then start them at the same time. During the test, make sure to monitor CPU usage of all client machines, especially if they are not identical. If any of them get into the 100% range, the test becomes invalid, so you need to reduce the user count on that machine and try again.

Testing multi-server setups

When using a linked multi-server setup, be sure to test that configuration. If you have two linked servers, run at least two load-testing machines simultaneously, pointing one to each server.

Advanced usage

If you'd like to change the way it generates data, the load test system has several other configuration options (for example if you think the frequency of `say` commands is not realistic). Please view the API documentation under `com.leafdigital.hawthorn.loadtest.LoadTest` for a current and complete command line parameter list.

Example results

I ran the load test for between 250 and 2,000 users on an old PC: a 1.6 GHz AMD Duron running Windows 2000 and 32-bit Java 1.6.0_13 (server VM).

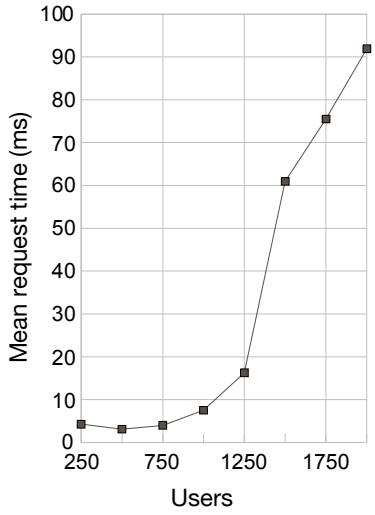
Hawthorn settings were all left at defaults.

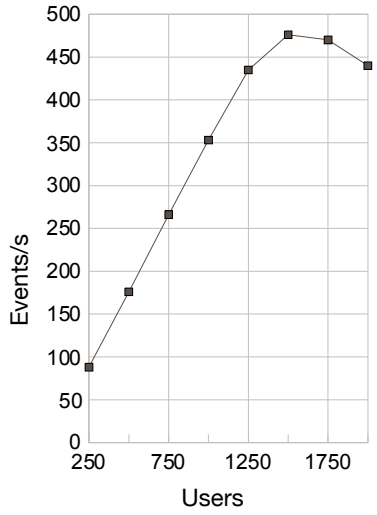
For each user count, I arbitrarily set the drive-bys value to one-tenth, so this test measures mainly the effect of active users.

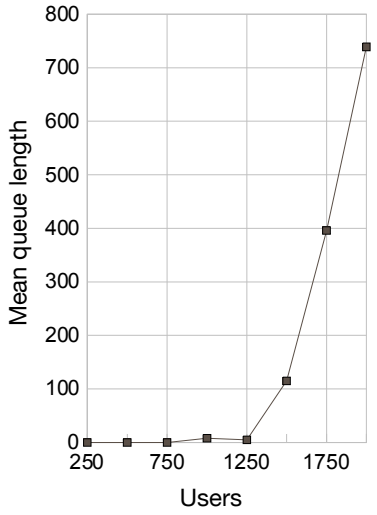
The following table and graphs summarise the results from these runs. Note:

- I omitted the 'Errors' and 'Exceptions' columns from the table, because these were always zero.
- Three columns were added that do not appear in the load test output: the user and drive-by counts, and my estimate (eyeballed) of typical CPU usage during the run.

Users	Drive-bys	Events	Seconds	Events/s	Event mean	Queue mean	~CPU
250	25	26,699	300	88	4.25	0.05	20%
500	50	53,092	300	176	3.07	0.08	35%
750	75	80,076	300	266	4.00	0.03	50%
1,000	100	106,194	300	353	7.54	7.89	70%
1,250	125	130,781	300	435	16.26	4.89	95%
1,500	150	145,318	305	476	60.97	114.84	100%
1,750	175	143,419	305	470	75.53	396.03	100%
2,000	200	134,492	305	440	91.92	738.85	100%







These results indicate that the system in question can happily sustain 1,000 active users (if they behave similarly to the simulated load-test users). 1,250 is about the absolute maximum. This is indicated by all relevant factors: event mean, queue mean, and CPU. The system can cope with a maximum of about 450 requests per second and, when it is running within capacity, typical request processing time is about 4ms including network delays.

One important value is queue length. If the mean queue length isn't near zero, then results from the load test are invalid (genuine performance would be worse). This is because the load tester uses a limited number of threads to simulate users, and waits for threads to become available if they are all busy; this is the 'queue'. Real usage does not have this limitation. In this case, the queue length indicates that results above 1,500 users are not accurate. The results for 1,000 and 1,250 users have small queue lengths; it might be possible to increase the load tester's thread count and rerun these tests to obtain a more accurate result.

If higher capacity is required, note that Hawthorn settings can be tuned. Increasing the `minpoll` setting to 5 seconds allowed this machine to handle 2,000 users.

Using other load testing systems

Because all Hawthorn requests are standard HTTP GET requests, you can also use any other web load tester, such as Apache JMeter, to test Hawthorn performance.

Configuration

This is the full list of options you can put in a Hawthorn configuration file.

Setting	Effect
Required settings	
`<magicnumber>`	Secret number used for SHA-1 authorisation key hashes. This number must be kept secure. If you suspect the number has been released, change it immediately.
`<logfolder>`	Folder used for Hawthorn log files. This folder must exist and be writeable by the Hawthorn system. All log files (system and all channels) are stored in this folder. The folder must not be shared by any other Hawthorn instance. You can use forward slashes in the path name, even in Windows.
`<servers>`	List of servers in the Hawthorn system. This must contain at least one `<server>` tag: `<server this="y" port="13370">192.168.0.1</server>` When using multiple linked servers, the other server(s) should also be defined here (but without `this="y"`, obviously).
Optional settings	
`<logdays>`	Number of days after which Hawthorn logs are deleted. If you set this to zero, Hawthorn will never delete its logs. The default is 7 days.
`<loglevel>`	Level of logging. Set this to `DETAIL` if you want every HTTP request to be logged (not recommended). `NORMAL` is the default.
`<logchat>`	Whether chat messages are logged. Set this to n if you don't want this server to log messages sent to chat channels. y is the default.
`<detailedstats>`	Whether per-verb command statistics are enabled. Set this to y if you want Hawthorn statistics to record details for each individual command verb. n is the default; this includes only the single standard `USER_REQUEST_TIME` statistic.

`<historytime>`	Time in milliseconds to retain channel history. Old channel messages are removed from memory after this time. The default is 900000 (15 minutes). This setting has a significant impact on memory usage.
`<minpoll>`	Minimum polling delay in milliseconds. Hawthorn tells clients to wait this long before making another poll request to check for messages, when a new message has recently been sent to the channel. The default is 2000 (2 seconds). This setting has a significant impact on the number of requests sent to the server by active users, so affects performance.
`<maxpoll>`	Maximum polling delay in milliseconds. Hawthorn tells clients to wait this long before making another poll request to check for messages, when there has been no new message on the channel recently. The default is 15000 (15 seconds). This setting has a significant impact on the number of requests sent to the server by users in idle channels, so affects performance.
`<pollscale>`	Time in milliseconds for polling delay to reach `maxpoll`. Immediately after a new message has been sent to the channel, the polling delay is `minpoll`. After `pollscale` milliseconds, the delay is `maxpoll`. (The polling delay is gradually increased between those points.) The default is 60000 (1 minute).
`<ipheader>`	Header to check for user's IP address. If Hawthorn is behind a load-balancer or other proxy, this should be set to the HTTP header used to provide the original IP address of the user, so that it can be correctly stored in Hawthorn logs. Typical settings would be `X-Forwarded-For` or `Client-IP`. The default is to use the directly-reported IP address and not check HTTP headers.

`<eventthreads>`	Number of event processing threads. Hawthorn uses event processing threads (in addition to a main thread and some other utility threads) to handle all requests. These may block but only very rarely, so the number of event processing threads should generally be just a few more than, the number of hardware threads supported by the machine. The default is the number of available logical processors ('logical' means that a HyperThreading core counts as 2), plus 2. You probably do not need to change this value unless you are trying to limit the server's CPU usage.
`<testkey>`	Generate key for testing. When testing the server (for example by using the HTML setup provided in lib/html), it may be useful to generate a test key you can use to access features. If included, Hawthorn will print this key to standard output when the server starts up. Example: `<testkey channel="c1" user="u1" displayname="User 1" extra="" permissions="rwma"/>` You can include this key more than once if required.

Logs

Hawthorn log files are all stored in a single log folder defined by the `logfolder` configuration option.

- There is one system log for the server for each day. This file has a name like `!system.192.168.0.1_13370.2009-05-15.log`. (`!system`, followed by the IP address and port, followed by the date.)
    - If your server has an IPv6 address, `:` symbols in the address are replaced with `!` so that the filename does not cause problems.
- There is one log file per channel per day. This file has a name like `channel.2009-05-15.log` (where `channel` is the channel name).
    - If the `logchat` configuration option is turned off, there won't be any channel logs.
- Log files are automatically deleted according to the value of the `logdays` configuration option.

Log format and behaviour

All log files are plain text. Each line contains the current time followed by other data.

Log files are written efficiently. As a consequence, there may be a delay of a few seconds before new information appears in a file.

System log

The system log includes:

- Server start-up information (logged whenever the server starts up). This lists all configuration options and some system information (Java version, Java VM runtime, and so on) which could be useful in diagnosing problems.
- Information about connection to linked servers, if any.
- System errors and warnings.
- Audit information whenever the following events happen on any channel:
    - Ban.
    - Log view.
    - Statistics view.
- Statistics, logged per day, hour, and minute. These are the same statistics shown on the server statistics page. See page 36 for details.

If the `loglevel` configuration option is set to `detail`, every single HTTP request to the server is logged. This is not recommended for production servers because it creates huge volumes of log entries, but may be useful when tracing problems.

Example lines

`15:20:00 STARTUP Hawthorn system startup`

- First line of startup information.

`16:12:29 SERVERTO 192.168.0.100:1338 ERROR Connect failure: Connection refused`

- Connecting to the remote server failed.

`16:12:30 SERVERFROM 192.168.0.100 CONNECTED`

- Received a connection from the remote server.

`16:12:59 SERVERTO 192.168.0.100:1338 CONNECTED`

- Connecting to the remote server succeeded.

`15:20:21 AUDIT BAN admin (192.168.0.100) banned u1 on channel c2`

- A user banned someone.

`15:17:47 AUDIT LOG b (192.168.0.100) obtained log for a on 2009-05-16`

- A user obtained a channel log.

`15:13:26 AUDIT STATISTICS admin (192.168.0.100) viewed statistics page`

- A user obtained the statistics page.

`16:16:00 STATISTIC M [MEMORY_USAGE_KB] 4222`

- Per-minute statistics (current data, or, for those which use averages, from the previous minute). The hourly and daily data are marked `STATISTIC H` and `D`.

Errors and warnings

If errors occur, you may see Java stack traces in the log. These are displayed on a single line, with `|` in place of line breaks.

A warning is displayed if an event thread blocks. This generally happens only when viewing the statistics page, and doesn't cause a problem. If it is logged frequently, something might be wrong.

Detailed log examples

`16:13:44 REQUEST 192.168.0.100 /hawthorn/say?channel=...` [*full path*]

- An HTTP request received from the given IP address.

`16:13:59 SERVERTO 192.168.0.100:1338 REQUEST SAY...` [*full message*]

- Information passed to a remote server.

`16:13:59 SERVERFROM 192.168.0.100 REQUEST SAY...` [*full message*]

- Information received from a remote server.

Channel logs

Channel logs include all SAY, JOIN, LEAVE, and BAN messages sent to the channel.

In addition to the message contents and basic user details, each line also includes the IP address of the user in question. This may be useful if tracking down inappropriate behaviour.

Example lines

`14:54:03 JOIN 192.168.0.100 u1 "User 1"`

- A user from 192.168.0.100 with user name u1 and display name User 1 joined the channel.

`14:54:04 SAY 192.168.0.100 u1 "User 1" hi there`

- The user from 192.168.0.100 with user name u1 and display name User 1 said ‘hi there’.

`14:54:17 LEAVE 192.168.0.100 u1 "User 1" explicit`

- The user from 192.168.0.100 with user name u1 and display name User 1 left the channel *explicitly* by clicking the ‘Close’ button. (The other possible leave event type, timeout, occurs if they leave without telling the server by closing the window in another way.)

`14:54:17 BAN 192.168.0.99 admin "Admin User" u1 "User 1" until 18:54:17`

- A user from 192.168.0.99 with user name admin and display name Admin User banned u1 until 18:54.

Statistics

The statistics page is available through a direct browser HTTP request to the Hawthorn server. You can access it through links provided by the connector in use:

- In the Moodle connector there is a link on the admin page for the Hawthorn chat block.
- For testing, the HTML, JSP, and PHP examples can all generate statistics links. (You need to request the *a* permission.)

When you view the statistics page it shows current information in two categories. `USER_REQUEST_TIME` is the latest information and is updated every time you reload the page, but other details were recorded the last time that the minute changed. For example, if you view the page at 17:54:39, you'll see values obtained at approximately 17:54:00.

Information for the current hour and current day appears below. In the example above, these would cover, respectively, from 17:00:00 or 0:00:00 up to 17:54:00. Most statistics are shown as averages (means) of the per-minute values.

The statistics page does not automatically reload; reload it manually to update.

If you want to see data for previous time periods, examine the server logs.

If the configuration option `detailedstats` is enabled, statistics for each verb, for the statistics page itself, and even for favicon[18] requests, are displayed in all three sections. These are shown with graphs in the same format as `USER_REQUEST_TIME`.

Statistic	Meaning
`USER_REQUEST_TIME`	Time taken to handle user HTTP requests. Shows the number of requests, the mean and medium processing times, and a histogram indicating the spread of times. This is counted from the moment when a user connection is accepted by the server, to the point where response data has been sent to the networking system. It is less than the delay experienced by clients because: • Networking delays (transfer time, the time in setting up a TCP connection) are not included. • If the server's main thread is busy (see `MAIN_THREAD_BUSY_PERCENT`), the connection may not be accepted immediately. One way to see a more accurate estimate of client times is to use the load testing system (page 23). When there is no message immediately available, requests for the `wait` verb only include the time taken to set up the initial request, not the time spent waiting.

18 The *favicon* is the icon that displays in the browser address bar while viewing the statistics page.

CHANNEL_COUNT	The current number of open channels.
CLOSE_QUEUE_SIZE	Number of connections waiting to be closed. When a connection is finished with, it is closed on a separate server thread. If the server is busy, this thread might get a little behind in closing connections.
CONNECTION_COUNT	Number of currently-open connections. Unless the wait verb is in use, this number is likely to be small because connections are closed immediately after sending a response. Does not include connections waiting to be closed (see CLOSE_QUEUE_SIZE).
EVENT_QUEUE_SIZE	Events waiting to be processed. Each request generates an internal server event. These events are processed on the server event threads. If these threads are busy, a queue might build up.
MAIN_THREAD_BUSY_PERCENT	Proportion of time that the server's main thread is active. The server has a top-priority main thread which accepts connections and reads request data before passing work on to the event threads. This design may limit system scalability. If the main thread is active for 100% of the time, it doesn't matter how many other processors might be available.
MEMORY_USAGE_KB	Current memory usage in kilobytes. The Java memory system relies on garbage collection. It is normal for memory usage to gradually increase over time; periodically, the system will do garbage collection that removes unnecessary data. If you want to know the 'true' memory usage, you can force garbage collection from the link at the bottom of the statistics page.

Verbs

Command verbs are issued as HTTP GET requests; each verb corresponds to an HTTP path (the `say` verb has the path `/hawthorn/say`, for instance). Verbs have parameters, given as HTTP parameters after a `?`.

All verbs require identity and authorisation parameters (user name, display name, extra data, permissions, key, and key expire time); verbs have additional parameters as below.

Verb	Parameters	Effect
`say`	`channel`, `message`, `unique`	The specified `message` (plain text) is added to the `channel` and sent to other users on request. A unique identifier ensures that messages aren't added twice (this could otherwise happen in certain failure situations). If the `user` is not present in the `channel`, they are automatically joined; a join message is sent. Requires `w` permission.
`ban`	`channel`, `ban`, `bandisplayname`, `banextra`, `until`, `unique`	The user with user name `ban` is banned from the `channel` until the time `until` (expressed in standard form, milliseconds since 1970). If the user is currently in the channel, they are forced to leave. `bandisplayname` and `banextra` are used to send information about the banned user to other channel users. The unique identifier avoids duplicates. Requires `m` permission.
`leave`	channel	If the `user` is currently in the `channel`, a leave message is sent. Leave messages are automatically sent after a timeout, but this verb sends one immediately. There is no `join` verb; other verbs cause automatic joins. Requires `w` permission.
`recent`	`channel`, `maxage`, `maxnumber`, `maxnames`, `filter`	Obtains recent messages from a `channel`, and information about users present in the channel. Messages are retrieved as far back as the `maxage` (in milliseconds). This must be at most equal to the server's `historytime` setting. Up to `maxnumber` messages are returned (the most recent) along with up to `maxnames` names of those present in the channel. If the filter is set to `say`, then only standard messages (not join, leave, or ban messages) are retrieved. Requires `r` permission.

wait	channel, lasttime	Obtains messages sent to a channel since lasttime. If there are no messages, this verb waits up to 50 seconds or until a new message is retrieved before returning. If the user is not already in the channel, they are joined to it; a join message is sent. This method works well and is much more efficient for the server than polling, but there are issues in browser support: some browsers force an hourglass cursor during the wait period. In addition, it's possible that web proxies might not like long-lasting connections. Consequently this verb is not used by default in current Hawthorn JavaScript. You can test it in JavaScript if you set a parameter when creating the HawthornPopup object (see JavaScript comments). Requires r permission.
poll	channel, lasttime	Obtains messages sent to a channel since lasttime. Even if there are no messages, this verb returns immediately. If the user is not already in the channel, they are joined to it; a join message is sent. The verb also returns a suggested poll delay based on the configuration settings minpoll, maxpoll, and pollscale. By default, clients are suggested to wait 15 seconds before polling again if there has been no activity in the channel. If there has been recent activity, the next poll action may be in 2 seconds, allowing for real-time conversations when the channel is active. Requires r permission.
log	channel, date	Retrieves (from disk) full logs of the channel on the given date (YYYY-MM-DD format). This verb is not currently used in the standard Hawthorn interface, but can be seen in the HTML example. Requires a permission.