Chalks, collaborative text editing for everyone

Design documentation

Rodrigo Benenson, Ricardo Niederberger

Technical draft as of
18th September 2004

Contents

Chapter 1

Chalks

Chalks is a software *under development* for crossplatform realtime concurrent text editing. The primary focus will be ease of use and minimum requirements.

Chalks is available at `http://chalks.berlios.de`

1.1 Requirements

These are the requirements for Chalks:

Realtime Concurrent Edition

- As defined in Chengzheng Sun's works, see the "Research Notes" chapter.

Easy to install

- The user should not know about python existence
- Lightweight
- Ideally only one executable file, without requiring an installation process

Easy to use

- Scan local network (also guessing local network) for Chalks servers
- Collaborative NotePad for dummies
- Very simple gui (figure 1.1), notepad/wordpad look'n'feel, so it should have similar menus and basic features:
  - Search, Find, Replace, Print, Save as, Insert date/time.
- Undo/redo operations
- Chat space among users
- Background text coloring for indicating the user who made changes
- Cursors for indicating other users current position on the text, as an additional measure to avoid conflicts
- Only text, no pretention of text formatting or any multimedia support
- One app instance can manage only one document, but can run many instances in a computer. In other words, Single Document Interface approach.
- One app instance can do only one thing at a time: connect to a remote server or serve a local file.

Descentralized data flow

- Nearby users will have better ping (responsiveness) than remotes one

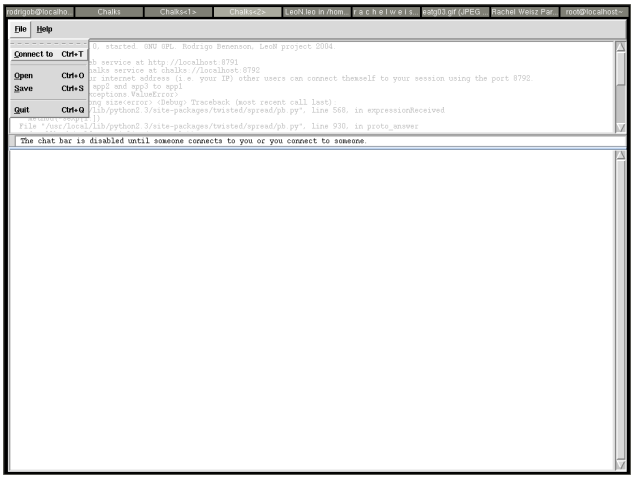

Figure 1.1: Chalks Gui Concept

1.2 Development difficulties

The development of Chalks has some specific difficulties, it is important the have them in mind in order to allow reducing risks:

- Debugging of distributed applications
- Creating tests for the concurrent editing algorithms
- Separating GUI frontend from concurrent editing logic in order to ease automatic tests

1.3 Aspects

Chalks is a software with three aspects. Identifiying these aspects helps understanding the application structure:

1. Concurrent Editing (figure 1.2)
   - Managing Concurrent Realtime editing of just text
     - Define the memory/cpu performance

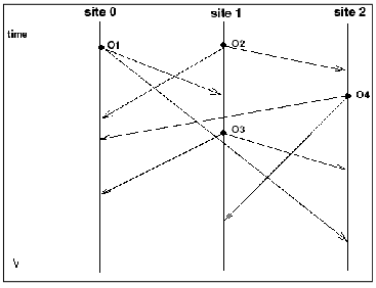

Figure 1.2: Operations in the time

   - Receive tagged atomic operations over the text and manage them
   - Generate tagged atomic operations over the local text and pass them to the network layer
2. Network

- Create a transparent layer between the concurrent editing of objects
  - Creates the ilustion of an "all to all" network (figure 1.3)

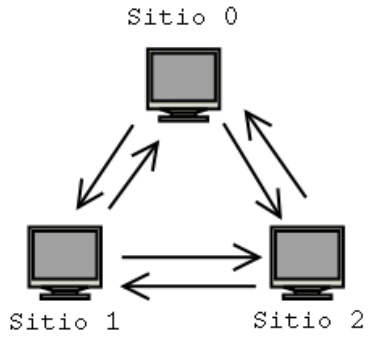

Figure 1.3: All to all network

  - Define the real network topology (figure 1.4)

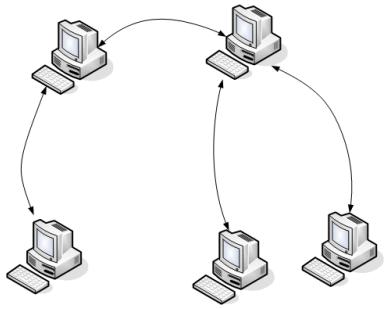

Figure 1.4: A possible real network topology

  - Define the protocol
  - Define the network usage performances
- Insertion/deletion of new nodes into the network ← *tricky point*
- How to discover Chalks servers on the current local network automatically

  Figure out local ip address and netmask, then scan ip range for default chalks server port. No need to code everything by hand, we will use the project Multicast DNS Service Discovery for Python by Paul Scott-Murphy. This module seems impressive, since it's compatible (well, claims to be) with Apple's opensource implementation for Darwin which should be pretty much industry standard.

  There is preliminary support for multicast DNS on Twisted (http://svn.twistedmatrix.com/cvs/trunk/sandbox/ita but right now it's very experimental, only suppports querying and doesn't seem to be evolving fast enough, so *pyzeroconf seems to be the best option.*

  We also need to think about IANA standard port numbers when coming up with a default server port and registering it on the network with rendezvous.
- *Translation/transport of atomic operations*

3. Gui

- Simple gui
- Reflect the local and remote operations over the seen text (figure1.5)
- Allow user entry of operations, pass the atomic operations to the Concurrentediting layer ← *tricky point*

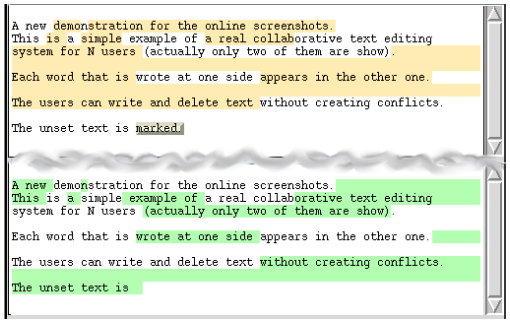

Figure 1.5: Remote users operations are colored

1.4 Research notes

For definitions of Real-time Cooperative Editing Systems, and their basic properties: Convergence, Casuality-preservation and Intention-preservation and how to achieve them (Search at http://www.researchindex.com for these files and related files):

- Achieving convergence, causality-preservation, and intention.. - Sun, Jia et al. - 1998 ← *the must*
- A Generic Operation Transformation Scheme for Consistency.. - Sun, Jia (1997)
- Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements (1998)
- Reversible inclusion and exclusion transformation for string.. - Sun, Chen et al. - 1998
- Realtime Collaborative Edition explained in plain English

Chapter 2

Design Proposal

2.1 Introduction

This text covers every defined aspect and proposes a technical solution to implement all requirements. This is mostly a documentation of the initial Chalks design.

Root concept: *Keep It Simple*, minimal complexity to acomplish the strictly necessary requirements.

2.2 Concurrent Editing

Direct implementation of the linear undo/redo algorithm described in Chengzheng Sun's works. This algorithm is the simpler aviable. It has a good compromise in the memory/cpu usage, with a some charge on the memory. Anyway as the objects managed are strings, the memory rarely grows more than a few tens of megabytes.

Chengzheng Sun's algorithm defines three objects: the operations, the history buffer (HB), and the resulting text.

Operations are specific transformations over the text (insertion, deletion) originated from a specific version of the text. The history buffer is a buffer of all the operations applied over the text. And the resulting text is the product of applying all the HB operations over the original text.

The concurrent editable algorithm defines what to do with a new operation. The processing of the received operation will have effect over the HB and over the original resulting text.

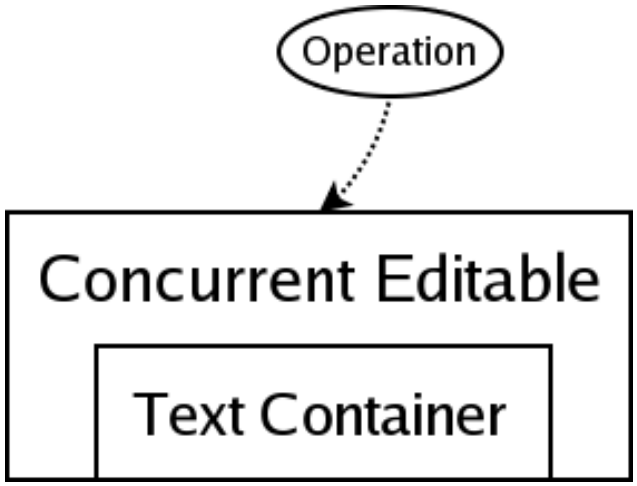

Figure 2.1: ConcurrentEditable, TextContainer and the Operations

Note: the actual WIP version does not separate in a clear enough way the TextContainer from the ConcurrentEditable object. I think this separation should be enforced to ease later integration with GUI

Using Python power the operations can be easilly represented as objects, following an easy API similar to Chengzheng Sun's notation.

In a similar way the Concurrent Editable algorithm can be implemented as an almost direct translation from the Chengzheng Sun papers to python code.

The `TextContainer` class is a simple one that contain an unicode string, and have methods for insertion, deletion of chars and the retrieval of the actual content.

2.3 Network

The definition and implementation of the networking system is the most difficult aspect of the software. Distributed bugs arises and issues not covered in the papers have to be afronted.

Essentially the network has to take care of three things:

- Sent operations have to arrive to every connected user
- If a new user logs in, everyone has to know about it
- If a user logs out, everyone has to know about it

Every operation has a specific identifier. This identifier is the number of operations that the emitter site has received from other sites (including himself). This identifier defines without ambiguity the *version* of the document over which the operation was generated.

To simplify the implementation instead of enumerating the sites (as in the paper) the sites are individualized by an unique id (ip+port). Then the operations are tagged with a dictionary of the kind `id1: ops_from_id1, id2: ops_from_id2, ...` (including site own id).

This approach becomes less efficient if many sites just connect once, send an operation and then disconnect, but this is a rare case. This approach is less efficient than using an ordered list, but it is simpler and functional. It is simpler because we do not care about the order of arrival, the disconnections and reconnections, and when new users are added to the session the management becomes trivial.

2.3.1 Network topology

Following the KIS principle which is the simplest network architectures that does the job fine ?

The proposal is to use a "Connect to One, Accept N" topology, also known as a "simple tree". See the image 2.2 to get an idea.

The concept of this topology is, very simple. One user only want to connect himself to one computer, so let's do that. If more than one user want to connect to the same computer let's do that. After the first connection no more connections are made.

This is topology is inefficient. In the figure, when site 4 edit the text, the operation will be relayed by site 0 and site 1, adding inecessary delays. This is true, but this topology is very simple implement. It has also an interesant advantage, it gives to the users the control of the network. In the figure, if sites [0,2,4] are in the same LAN, and sites [1,3], in another one, then the topology is not so bad. Also if an user is working in a slow connection (example, low SNR wireless conneciton), it would prefer to have only one connection to the nearest (in ping measure) machine. Having more than one connection add transfers costs to keep the TCP/IP links alive. If the users control the conection they can choose the computer with overload. A tipical user case will be simply N-to-One (central server receiving all the conections).

The tree topology is not only simple to implement but also simple to analyse.

When a site receives an operation it applies it locally and spread it to all the other known sites, excluding the node that sent the operation. As the topology is an acyclic graph, no special precautions has to be taken (to avoid repeated reception).

Let's see what should happen when a user connects himself to a session.

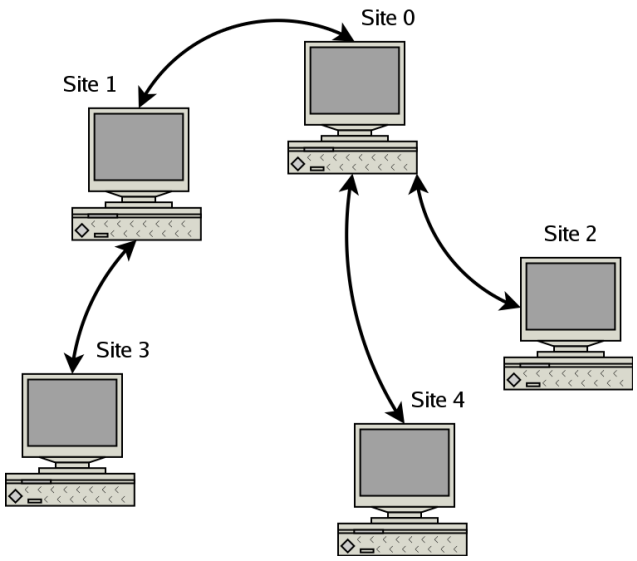

Figure 2.2: Connect to One, Accept N ilustrative topology

- New user enters the network

  A user has choosen a machine to connect to. This machine receives a new connection and has to send back, the current text and the HB. We have to take care of new operations that could be generated during the connection process.

  Chengzheng Sun's algorithm includes a purge procedure to avoid the HB getting unncessarily big.

  This HB purge could be implemented as a method which calls itself repeatedly, using a call like "reactor.callLater(self.HBPurge(), purgeDelay)". This "service method" pattern could repeat a lot on such networked application, especially when using a networking framework like Twisted, with deferred execution.

  Upon the first text editing, the new user will generate an operation with a new entry in the tag. When a site receives it for the first time it will include it in the list of know sites. If a received operation does not include (not a key in it's id-¿version map) a known site id, obviously this node has received zero operations from that site.

- A user quits the network

  No special precaution has to be taken when a user quits. We have to simply disconnect from the parent node, and kill all children nodes. This will create a cascade that will close the branch adequately.

Note: this could be an inconvenience, depending of the usage scheme. Maybe we could create a transaction system to invite all children nodes to connect themself to the parent node. But synchronisation issues may arise ("what if a packet arives to the parent while children-childrens are connecting to them ?"). This is a non trivial topic that we should check. For an initial implementation branch die seems fine to me, but is it the desired behaviour ? (let's check the requiriment)

  HB purge should also purge sites info. If a site id does not appear more in the HB, then it should be deleted from the known sites list, thus shrinking back the packet sizes since no memory traces are keep in any connected site.

2.3.2 Protocol

`Operation` objects serialized and transfered via Twisted Spread.

- Well documented API
- Simple and eficient binary serialization (low bandwith usage)

- Well documented Serialization protocol (Banana)
- Fast serialization (C modules)
- Pythonic

2.3.3 API

Twisted

- Well documented API
- Full featured, spread, authentication, web services, diverse protocols, etc.
- Pythonic
- Easy to embed in the software distribution

2.4 Gui

Show the text container, generate new operations over the text, allow sending chat messages to the other users.

The idea is to provide to the user all what it need to see in only one window, splitted in two areas: the text area and the chat area.

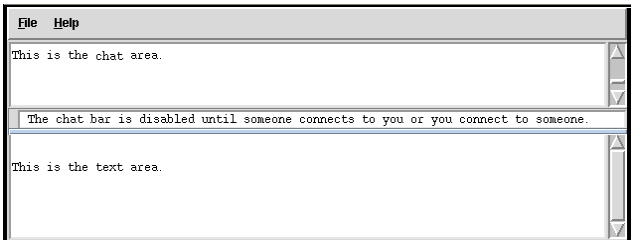

Figure 2.3: Text and chat areas in the same window

The chat area is used as a log window, to receive message, and have an input section to send messages. This input section can also be used to enter advanced commands (to enable/disable debug logs, to select special options, etc...). Different colors in the chat bar are used to differenciate diferent kind of messages.

The chat area is resizable (click and drag). If it is totally colapsed it transform itself into a status bar, that show the last line in the chat area.

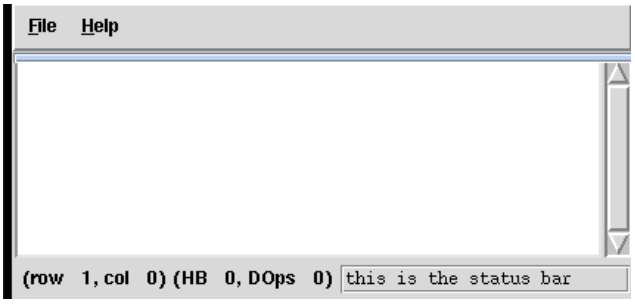

Figure 2.4: The chat area is colapsable, transforming itself in a normal status bar

In the text area color codes are used to differenciate the users entries. Also the local entry are marked in such a way that the local user know when a group of chars have been sent or not.

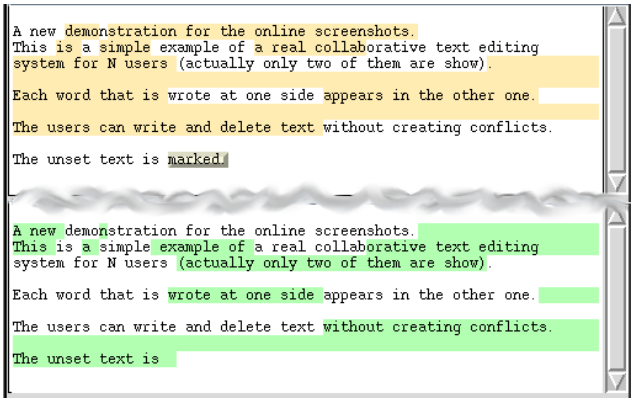

Figure 2.5: Text from other users are colored. Text not yet processed is raised

The color of the users is randomly choosen at the local site. The color generator choose in a family of smooth agradable colors.

2.4.1 Menus

The menus should be strictly minimal, siilar to NotePad ones, but with less options. The number of dialogs is also minimal, and they complexity very low.

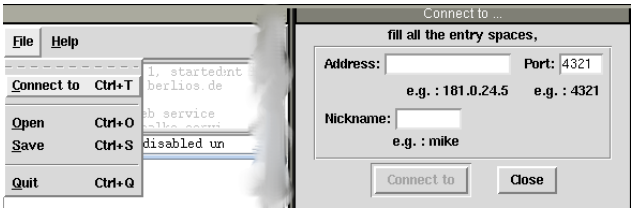

Figure 2.6: Simples menus and dialogs

2.4.2 API

Tkinter

Pros
- Easy to include in the python distribution
- Ligthweigth (? compared to wxwindows ?)
- Python default gui, crossplatform
- Good coloring facilities
- Documented, easy API
- Full Unicode support
- Stable API

Cons
- Manage text as a bidimensional object
- Ugly: The gui is not similar to the OS default one (this may change in the future)
- Low mindshare among other programmers, wxPython seems to be the mainstream python gui programming toolkit nowadays
- Increasingly few projects using it and as such there is less people reporting bugs, etc. (development is not too active lately)

- But anyway we believe we should keep on using tkinter for the first release and see how things play along. Provided we honour our intent of keeping GUI code clearly separated from concurrent/networking code, we should be safe

Note:the gui code implementation should be as well defined and separated as possible to allow future implementations using diferent gui engines (natives, scintilla, etc.)

Chapter 3

Notes

Misceleanous notes about sections or aspects of the code.

3.1 About the connections process

The Children start a connection. The ChalksNode object request a connection to the parent.

The parent receive access to the children as a reference to ChalksNode and the children gets access to the parent obtaining access to a ChalksAvatar instance created in the parent side.

ChalksNode can receive calls to the remote_ (view_) methods. ChalksAvatar can receive calls to the perspective_ methods.

We want the childrens able to call the ChalksNode methods. Thus when the Childrens call the collaborate_in using his avatar on the parent side, he receive back (in a huge list of params) the parents ChalksNode reference.

The connection process occurs in this order:

1. ChalksNode require a connection to the parent using the `connect_to_parent` method. Thus a parent ChalksAvatar object is created and passed to the parent as an access to the Children. This parent ChalksAvatar instance has no `mind` object.

2. The parent receive through his `ChalksRealm` the children connection request. A children ChalksAvatar instance is created in the parent side. The parent return to the children a reference to this avatar.

3. The children receive this reference (at `logged_in`) and register it as the parent perspective (the access of the children to the parent side). It also register this perspective as the mind of the children side parent ChalksAvatar instance.

4. Now that the children is connect he call in the parent (througth his avatar in the parent side, i.e. throught the parent perpespective) the `collaborate_in` method.

5. The parent receive the call, register the children perspective (local children avatar mind) as one more site in the collaboration session and return to the children all the required information to start collaborating.

It worth noting that the ChalksNode object keep only *remote references* of the avatars (of the parent and of the children). When a node call a remote method in the called node receive a call from the local Avatar instance. Thus *the mind* of the local Avatar is compared to the childrens and parents remote references. For a clear example look at the `send_message` ChalksNode method.