OPEN SOURCE APPLICATIONS FOUNDATION

Chandler Architecture

1. **An overview of design values**
2. **Block Diagram**
3. **Chandler architecture overview**

An overview of design values

As is often the case in the early stages of software projects, our goals for the Chandler Interpersonal Information Manager are ambitious, given our limited resources. But by choosing an architecture that uses the best technologies for what we're trying to achieve, and being smart about the development process, we should have a great leg up. We hope to streamline the development process via two main strategies: First, we'll build on as much existing open source software as possible, getting help from the open source community to evolve and expand it as necessary. Ideally, the vast majority of code in Chandler will be code we don't write. Second, we'll combine sound architectural decisions with improved development methodologies to make software a lot easier to write. Key to our success will be having a world -- class team of developers head up the effort.

As with most human endeavors, good software architectures evolve, and what I present here is only an overview, or summary of our architecture values. As we get further along in development, I'll revise this document accordingly, including more details of the design.

Here's a summary of our primary architectural decisions:

- Build on open source software that supports open standards, choosing projects that are reliable, well documented, and widely used
- Use the Python language at the top level to orchestrate low level, higher performance code

- Design a platform that supports an extensible modular architecture
- For the desktop client, choose a cross platform U/I toolkit that provides native user experience
- Use RDF and a persistent object database
- Build in security from the ground up
- Build an architecture that supports sharing, communication, and collaboration

Guiding Principles

Use existing open source software that supports open standards, choosing projects that are reliable, well documented, and widely used

Much of our work involves figuring out which of the vast array of open source software is best suited for our task, then adding the code necessary to make it callable from Python. Some of this software doesn't currently meet our requirements for quality, performance, or feature sets, so we'll contribute toward improving projects we depend upon, as necessary. Hopefully these contributions to the open source community will make it easier for others to write quality application software in the future. The Mozilla project stands out as potentially having a number of key pieces we hope to reuse, the most obvious being the Gecko HTML layout engine and the HTML e-mail editor.

Use an interpreted high-level language to as the glue that holds together low level, high-performance pieces

Perhaps one of our more unconventional decisions is to use Python -- a dynamic, interpreted language. Compared to static languages like C++ or Java, dynamic languages tend to have poor performance and lack features for developing large programs like Chandler. But Python has many other features of modern programming languages, and its dynamic nature makes it easier to program than its static counterparts. To mitigate performance deficiencies, we plan to use Python only at the top level to orchestrate lower-level code written typically in C/C++. Much of this lower-level code already exists in open source form and in many cases is already accessible to the Python programmer. As for supporting large programs, Python is better equipped than most other dynamic languages and we hope to face only minor compromises that will be far outweighed by the advantages.

The Python standard library contains a vast array of modules for developing applications. Some of these are written completely in Python and others are written in C/C++. They include a number of important protocols like SMTP, POP, IMAP and HTTP, and a host of useful libraries, including an XML parser, socket libraries, a regular expression package, and much more.

Build a platform that supports modules at a variety of levels

By designing a platform with well-defined APIs, we hope to enable both ourselves and the open source community to easily write modules we call parcels that can be plugged in at a variety of levels. This will make it possible to replace portions of Chandler to make improvements or add functionality that we could never anticipate -- and conveniently share them with others users.

Examples of modules include the "viewer parcel," an area of the desktop user-interface which can view various kinds of items, for example, the calendar view for viewing calendar event items, an email view for viewing email items, a contacts view for viewing contact items, or a generic table view that can view items of any type. If you have some experience programming Python, we hope it would be relatively easy to add a new kind of data, for example your music collection including information about each song and write a music viewer that you could use to search and view your music.

Parcels can be used at a many different levels. They might include an API for filtering email that can be used for spam filtering, an API for plugging in new protocols used to enhance or improve communication, an API for extending the search capabilities in each view, or even an API to attach a script to a button in the user-interface.
Finally, we hope that these parcels and other building blocks like our persistent object database can be reused in unrelated software projects. This will make it easier to develop software and increase the quality and choice of available application software.

Use a cross-platform U/I toolkit that provides native user experience

One of our goals for the desktop client is to have a version that runs on Linux, Macintosh and Windows. We want each of these versions to be better, in terms of the user experience, than the existing counterpart on the same platform. This means the user interface must conform to each platform's application conventions. To implement native user interface elements on each platform, we could either use a different U/I toolkit for each platform, or try to find a single toolkit that allows implementing native elements for all three platforms. We chose the latter approach, and among the open source options, wxWindows/wxPython appears to be the best candidate.

There are limits, however, to the number of different user interface elements you can abstract to a single cross-platform U/I toolkit. For elements whose semantics aren't the same on all platforms, we'll have to write special-purpose code for each platform. Also, wxWindows has many limitations that will need to be addressed. In particular, the Macintosh port is new and still needs work, and on all three platforms, we'll need to improve and add support for various U/I widgets.

Choose data storage that's easy to use and evolve

Chandler has a repository, or database that stores a potentially huge volume of e-mails, attachments, contacts, documents, and so on, the structure of which will have to evolve over time. This database is probably our biggest architectural challenge.

Early in the project, we decided to use RDF (Resource Description and Framework; www.w3.org/RDF) because of its ability to describe data in a very flexible format and exchange semantic information between applications in a standard format without loss. Because RDF is a World Wide Web Consortium standard, we hope to gain benefit from the existing tools, validators and applications that have been developed or will be developed.

For the repository, we've decided to use an object database. Historically, most databases haven't stored data in a form convenient for object-oriented programming. This meant extra work for programmers who had to convert back and forth between different data representations, and it introduced a variety of other difficult housekeeping challenges. Object databases try to solve this problem by transparently storing data in a form that's convenient for object-oriented programming, and they often solve many housekeeping tasks with built-in garbage collection, transactions/undo, and support for weak object references. While object databases can be difficult to use because of the limitations of static languages like C/C++, Python's interpreted nature makes it easier to implement a simple object database. There's currently a Python special interest group chartered with the task of coming up with a standard Python object database, which will likely be based on ZODB (the ZOPE database;

www.zope.org/Wikis/ZODB/). We think there's a good chance that using a standard object database and language like Python that are tailored to work well together will make it much easier to write applications like Chandler.

So you might ask, are RDF and ZODB compatible? The short answer is almost, but not completely. Although people typically store RDF as triples, there is nothing fundamental about RDF that requires it. If you look at the Python data structures, you can usually find a simple way to represent them as RDF. We plan to provide a way to import and export RDF to ZODB, and also to store the schema for our data in the database itself.

If we design the interface to the database correctly, it should function equally well as a server or as a database local to the machine accessing it. One of the Chandler's design goals is to not require a server. However, if the server is available, Chandler could provide some extra features that rely on access to certain shared data always being available. It's not our intention, however, to provide the vast array of business features that server-based enterprise businesses require.

There are admittedly a number of risks with an RDF/ZODB approach. First, ZODB isn't finished, and it has a number of performance problems. Second, combining two technologies like RDF and ZODB that weren't designed to work together, might lead to unforeseen problems. And there are still other significant tasks to tackle, most important of which are probably searching and indexing, and replication.

Improve and simplify the experience of sharing, communication, and collaboration

Now that we live in such a highly networked world, where many of our friends and family as well as workmates have a presence online, we can add several important new features that improve sharing, communication, and collaboration. For example, based on people's calendar entries, schedules can be displayed in a friendly way making it easy to see the if someone is available for an instant messaging, voice, or video conversation. A student planning a trip home for the holidays can overlay her calendar with her parent's to find the best time to schedule the visit. And when people update their contact information, the changes can be propagated to everyone who has subscribed to a copy of the contact.

At the highest level, items in repositories are accessible via their RDF URL. However, the communication path varies depending on the availability of the client storing the data that needs to be accessed. By default, the best communication path will be chosen automatically by the system accessing the data. Each communication path, available via a common API, might have different capabilities for availability, latency, and capacity, and communicate using a variety of protocols. We use Jabber as a low-capacity transport to discover clients, which once identified, can attempt to establish a direct high-capacity connection between themselves as peers. When a client is unavailable via Jabber, the fallback is to communicate transparently via SMTP (since everyone will have an email account). In this case, SMTP acts only as a transport mechanism -- data payloads can be included as MIME attachments in the email, but nothing would show up in your inbox.

Many people now have multiple computers: a desktop computer at work or school, a laptop, and a computer at home. Supporting work on a common set of data from these different computers requires either a server to share or store and forward data, or the ability to occasionally connect the different computers as peers to synchronize and replicate data that's changed since the last synchronization. We plan for peers to be able to modify data, whether connected to other peers or not. When they reconnect after being off-line from one another, an automatic mechanism will synchronize replicated data between each computer using the communication paths described above. A version and/or timestamp can identify which items need to be replicated, and on the rare occasion that multiple peers modify the same data, this versioning can identify which changes conflict so the user can choose the one to accept. When data resides on a server, or the peers can always communicate, such conflicts would not occur.

Another type of synchronization will allow users to make an "offer" of an item in their repository, such as their contact information, to which other users can subscribe. Changes to the item being offered can then automatically propagate to subscribers, ideally using the same underlying mechanism described above for synchronizing replicated data.

Use a Model View Controller architecture

The Model View Controller (MVC) architecture separates the back-end data and application (model) from the front-end user-interface (view). The controller links the view and model. One big advantage of MVC is that it makes it easier to add a new front-end, for example, a Web or PDA interface for Chandler, without modifying the model. MVC is also useful in large applications because the data model can be developed largely independent of the user interface code.

Security is hard

We're just beginning to work out our security strategy, and based on our preliminary exploration, it doesn't look like there's a completely transparent, 100 percent secure and easy-to-implement solution that interoperates with other clients, especially for the product without a server. Inevitably, we'll have to make some compromises. In any case, it has to be easy for users, otherwise they won't use it, and that would be worse than having no security at all.

For example, to ensure that only certain people or groups have access to data, we need to provide a mechanism for permissions or capabilities. At the implementation level we hope to provide a flexible fine-grained control over data access, but at the user level we plan to simplify access, in part, by associating items with a group of people and defining access for the group. We haven't yet worked out the details of implementation or even which model -- permissions, capabilities, or something else -- to use.

Managing Risks

Probably our two biggest risks are schedule and unforeseen problems with the open source software components we're using.

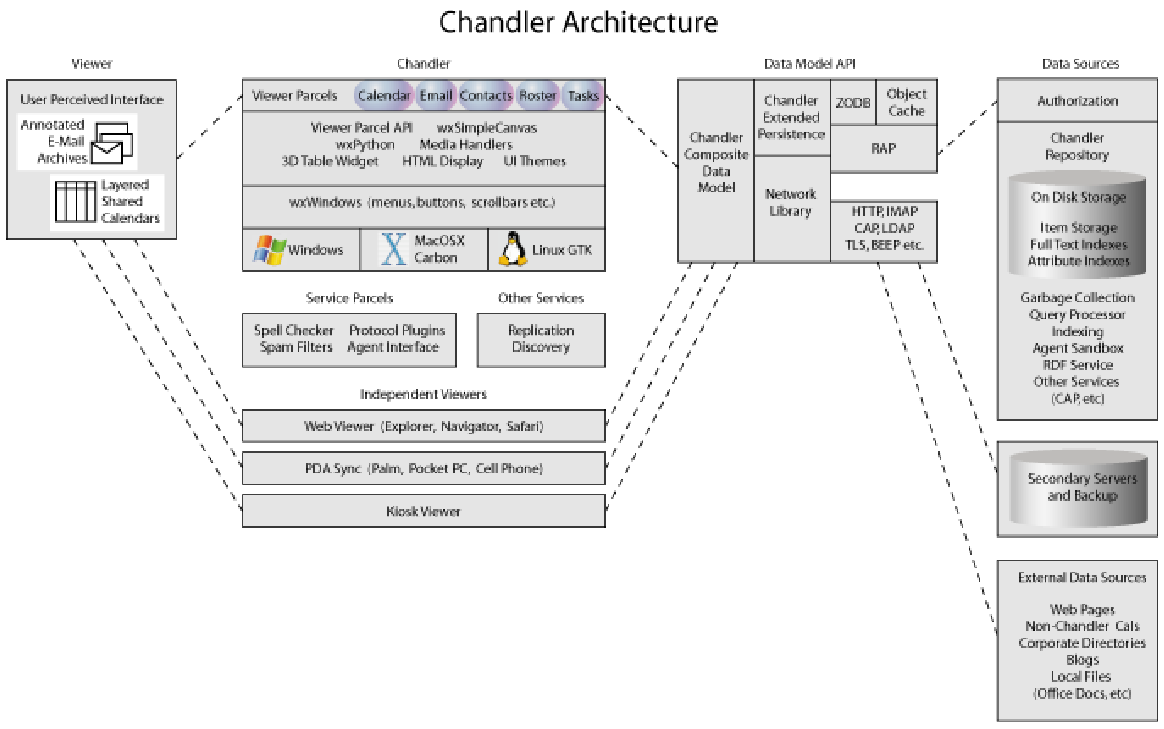
Scheduling is hard

Software projects are notoriously difficult to schedule, especially efforts that haven't been done before. I've seen two scheduling techniques used successfully. One is to estimate the total lines of code of a similar project and plan on writing somewhere around 20 to 40 lines of code per programmer per day. We don't have precedent of a similar project, so we'll attempt the second technique. This is to break down the project into pieces you can imagine finishing in less than a day or two, and totaling estimated hours for each piece. As developers complete their pieces, you adjust remaining estimates to reflect any average differences between estimated and actual times. We're only just beginning to break down the project in detail and track our progress, and much work remains, so we don't yet have a realistic schedule. And once we have one, it will of course be subject to change as we make new discoveries and learn from experience along the way.

Open source software needs revision

Our project depends on certain open source software that's either incomplete or doesn't meet our requirements. The projects that appear to present the biggest risks are wxWindows/wxPython -- particularly on Macintosh -- and ZODB. To mitigate these risks we've contracted with the author of wxPython to make the necessary improvements, and we hired an engineer to work on ZODB.

We plan to use a number of standard development techniques that can help minimize risks, identify problems as soon as possible, and increase software quality. For subsystems we rely on heavily, like ZODB, we're writing test programs to measure performance and identify bottlenecks, so we can code the necessary improvements. Unit tests will be written as part of the subsystem coding and can be automatically run during the build process to identify problems as soon as they're introduced. I also want to make code reviews part of our culture. Too often skipped because of schedule delays, code reviews are one of the most efficient ways to discover problems.



From www.osafoundation.org/architecture.htm#block_diagram 31 August 2003

An Overview of the Chandler Architecture

by Michael Toy, OSAF

Introduction

As Chandler is still in the early stages of development, the architecture is still evolving as we learn more about the problem space. Our vision of Chandler as a platform for a variety of information-centric applications requires a serious upfront investment in planning and design. This document is a snapshot of OSAF thinking as of the 0.1 release of Chandler.

Overview

The diagram is divided into two pieces labeled Viewer and Data Sources because Chandler makes more sense seen this way then across traditional client/server boundaries. Some code that is over on the Viewer side of the picture may run on a server, and some of the Data Storage facilities may happen on the client.

The diagram begins in the upper left corner with types of data items a Chandler user will typically be interacting with. The items will be perceived by the user to be a single object or piece of information but will often be composed of data from many different sources.

The piece labeled Chandler Composite Data Model provides a unified view of objects whose constituent pieces could come from a variety of attached and detached sources. For highest performance, this piece needs to be close to viewers of the data. This "middleware" may be instantiated in different places depending on the type of Chandler installation. In the case of a standalone Chandler client, it would exist in the client. In the case of a web based Chandler client, it might exist in the repository server or in the web based client code.

Viewers

As the diagram moves roughly from top to bottom on the Viewer side, we see the implementation stack of the interaction space. There will be a variety of ways to interact with Chandler data, the Chandler application is just one of them. The code used to build the Chandler application is split into modules that can easily be included in other applications to provide Chandler behavior on a variety of platforms. In the diagram we see; Web Viewer and PDA Sync applications subscribing to the Chandler Composite Data Model and using other services.

ChandlerView

The program which most people with think of as Chandler is the *Chandler View* application. This application is written in a combination of Python (for the high level interaction code) and C++ (for the platform specific and performance sensitive pieces). This application is built on a set of high level info-centric modules. Some of the modules are shared with all Chandler viewers, and some are specific to the Chandler View.

These modules are packaged in a Parcel, which is a piece of code for which extend the capabilities of Chandler. The Service Parcels are the modules which are shared across all applications which view Chandler data, while Viewer Parcels are modular user-interface components with domain-specific interactions for different data types. In the 1.0 of Chandler, we will provide parcels for manipulating calendar, e-mail, contacts and task lists.

Viewer Parcel API

Chandler provides a rich API for implementing info-centric applications. The foundation of this is the Viewer Parcel API, which provides simple access to common services shared by all parcels. This architecture makes it relatively easy for programmers to create new parcels that can display new item types, or provide a specialized view on existing item types. Since parcels are written in Python, they can be dynamically loaded, allowing the user to extend Chandler without having to re-compile.

wxPython

Chandler's UI framework is based on wxPython, which is in turn based on wxWindows, a cross platform UI toolkit that uses native widgetry on all three platforms, but provides a common API.

Table Widget, HTML Display, etc ...

Chandler will extend the off-the-shelf interaction components available with wxWindows to include various other pieces that are useful in building data-centric applications. There will be host of Media Handlers for displaying images, sounds, HTML, etc., that are used in various parts of the user-interface.

We are planning to develop a custom wxWindows user-interface Table Widget. It will be used to display information items in a tree view, where each node in a tree is a table. An HTML layout engine will be used for both displaying web pages and editing HTML e-mail.

Other Services

Here is a peek into the application independent portion of Chandler services. As with the viewer parcels, this list of services can be extended dynamically by importing new parcels.

Discovery

Chandler will use a variety of techniques to learn about the network, allowing Chandler users to interact and share data without requiring central administration. These features are still in early stages of investigation, but we expect to depend on whatever the emerging standards are in these areas.

Replication.

Although the servers may perform replication, the control over replication lies with the client. Replication is an action performed at the user's request, not a transparent action performed on the user's behalf.

Data Model API.

Left to right, across the top of the diagram, we see a rough outline of the implementation stack for the Chandler Data Model.

Chandler Extended Persistence

Object persistence has long been favored as a highly productive way for programmers to deal with data. Chandler extends the concept of persistence to include network transparency, versioning, and both attribute based and full text indexes.
ZODB.

Python has an excellent persistent object store, ZODB. Chandler will use ZODB as the base for implementing extended persistence. As data moves out of memory, Chandler's object store will use RAP (Repository Access Protocol) to transfer data between clients and Chandler servers.

Data Sources

A Chandler client will provide access to data from a myriad of sources. The top portion of the Data Sources diagram shows a Chandler data repository operating as a server responding to protocol requests from several sources. A Chandler client may act as a server to other clients. A user may have their repository on their own laptop, or they may be accessing their repository remotely through a Chandler information kiosk. There is another document **Data Sources Configurations** that shows several of these configurations.

Chandler Repository

A Chandler repository starts with an authorization layer, which provides authentication and access control. Garbage collection, or compaction of the data in the repository is provided transparently to the Viewer. The repository has a query interface that is used to return references to objects based on access to the various indexes stored in the repository. Our intention is that this query interface will be based on a Simple But Currently Unspecified Query Language, which for fun (because I have been working on documents too long) I will now call that SBCUQL.

The creation of indexes has to happen in the portion of the Chandler cloud where the data is fully introspectable, and this is currently a fuzzy boundary between the client and server portions, but the data for the indexes will be on the server, and so the intention is to have the code which creates the indexes also live on the server.

In a universe where data is distant, it makes sense to be able to have code running near the data. We are still in the early stages of thinking on this, but there is a desire to have a way for the user to delegate authority to pieces of code (or agents), which are sent to the repository to perform functions and report back later. Some sort of protection will need to exist in order to be able to provide security and privacy guarantees in the face of mobile code.

Repository Service Interpreters

Chandler repositories may also be interesting data sources to other network clients, and so there may be other "services" which provide an interpretation of the repository. On the diagram we have drawn a service that allows CAP calendar clients to connect to the Chandler repository and a service that provides an RDF view of the item store.

External Data Sources

As Chandler is trying to erase the boundaries that exist between different types of data, Chandler with have to inter-operate with a variety of other existing sources of information. Initially this will be limited mostly to import and categorize style of interaction, but as we gain experience our hope is to extend the full Chandler capabilities of sharing and data-composition to as wide a variety of users as possible.