Technical Overview of Comanche

Daniel Lopez Ridruejo, ridruejo@apache.org

This paper documents Comanche internals and the design philosophy behind it. It will help you get started if you want to write your own plugins or want to help develop the existing ones.

Technical Overview of Comanche 1
Daniel Lopez Ridruejo, ridruejo@apache.org 1
1. Introduction 2
1.1 Design principles 3
"Easy things should be easy and hard thing should be possible" 3
"There's more than one way to do it" 3
"Laziness is one of the virtues of a good programmer" 3
"Many acceptable levels of competence" 4
"There are probably better ways to do that, but it would make the parser more complex. I do, occasionally, struggle feebly against complexity..." 4
"Historically speaking, the presence of wheels in Unix has never precluded their reinvention." 4
End users 4
Experienced administrators 4
Developers 4
2. Architecture 4
2.1 Simple API for developers 5
2.2 Support for multiple extension languages 6
2.3 Easy addition and maintenance of configuration files 6
2.4 Easy maintenance of different interfaces (and languages) 6
Namespace 7
Console 7
Plugins 8
3. XML based configuration 8
3.1 String 10
3.2 Number 10
3.3 Boolean 11
3.4 Choice 11
3.5 Label 12
3.6 Structure 12
3.7 List 13
3.8 Alternate 14
4. XML User Interface 18
4.1 guiString 19
4.2 guiLabel 19
4.3 guiLabeled 19
4.4 guiBoolean 19
4.5 guiImage 19
4.6 guiChoice 19
4.7 guiList 19

4.8 guiStructure 20
4.9 guiAlternate 20
4.10 guiPropertyPage 20
4.11 How it works 21
5. Distributed architecture 23
5.1 Different models for distributed operation 23
Web based approach 23
Distributed application approach 23
5.2 Few, well defined API calls 24
Namespace / Plug-in 24
Namespace / View 25
5.3 Inter process protocol based on XML 25
5.4 Concurrency handling 26
5.5 Delegation of authority 27
6. Programmer tutorial 27
6.1 Creating a module. 27
6.2 init 29

1. Introduction

The main issue that Comanche wants to address was the lack of appropriate management tools for popular internet software. The Apache web server and derivatives runs on more than 60% of all internet servers (according to Netcraft). Sendmail runs in approximately 75% of all mail servers on the internet (Sendmail website). (Bind has a virtual monopoly for internet DNS servers. Big organizations running SMB networks (windows file and printer services protocol) often turn to a Unix Operating System running Samba for their mission critical file servers..

These programs are free. Their source code is available for inspection and customization. However, although they are often technically superior to their commercial counterparts, they usually lack good management software. This steep learning curve often hinders their adoption both by the casual user and large corporations who want to keep down their administrative costs.

The purpose of this project is to implement a management system, code named Comanche, geared toward improving the usability and widespread adoption of Internet related Open Source software.

Design goals:

- Avoid focusing on a single product. Create a framework for easily developing management programs
- Provide the ability to easily develop simple modules yet still make it possible to architect complicated ones.
- Build an intuitive GUI to hide complex tasks from casual users while providing full flexibility to advanced users.
- Enable remote administration
- Possibility to extend the framework in arbitrary programming languages

- Support localization of both Help files and interface

The following sections give an overview of the architecture of Comanche. Later on we introduce XML usage in Comanche and we conclude with a step by step implementation of a simple module.

1.1 Design principles

After downloading, compiling and using many configuration programs and management frameworks, it was felt that there was room for improvement, and thus the idea of writing Comanche was born. Some of the key points that motivated that decision follow. To describe them, some thoughts are borrowed from the author of Perl, Larry Wall.

"Easy things should be easy and hard thing should be possible"

Comanche, as any other open source project can benefit greatly if other people were able to contribute to it. The idea is to develop a framework that encourages other programmers to contribute. Successful open source projects have a modular architecture. To become productive, a developer only has to deal with the module API and a few concepts of the overall architecture. The learning curve is much higher for one piece monolithic programs.

Existing frameworks usually aim to be comprehensive solutions, providing special functions for registring and accessing configuration files, starting/stopping programs, etc. This increases the amount of knowledge required to build even the simplest of the modules. One of the goals of Comanche is to make it easy to develop plug-ins. The API should be kept simple. A developer should be able to write a simple module for Comanche in a very short amount of time, without precluding the development of more complex modules.

"There's more than one way to do it"

Some of the existing frameworks restrict the developer to the use of only certain extension languages, like C++ or Java. The design of the framework should not restrict the developer to the use of only certain languages.

"Laziness is one of the virtues of a good programmer"

Configuring complex programs can be a daunting task. Hard-coded interfaces make painful to maintain the program when new versions of the application are released and that requires changes to the syntax of directives, redesign of existing dialogs or creation of new ones. Usually all this requires changing the source code and recompiling with existing frameworks. It will be desirable to clearly separate the interface and directive description from the configurated engine itself, so interfaces can be generated dynamically.

"Many acceptable levels of competence"

If this separation of content from presentation can be achieved, then it would be possible to maintain different versions of the interface. New users would be presented with the minimum set of information required to configure the program, while experts would be presented with an advanced view, that exposes the more obscure details.

"There are probably better ways to do that, but it would make the parser more complex. I do, occasionally, struggle feebly against complexity..."

The design of the program should be kept as simple as possible while covering the most common cases and uses. This may involve some functionality tradeoff.

"Historically speaking, the presence of wheels in Unix has never precluded their reinvention."

In this case it is believed that it will be a better, faster, rounder wheel.

Now that it has been stated what the ideal configuration framework, it should be analyzed which problem it should/would solve for different target groups:

End users

It should help them set up and manage popular open source applications like Apache and Samba. It should make the process easy by hiding the complexity and guiding them through the different options using wizards. When they feel confident, they should be able to switch to expert mode and be able to configure the more obscure parameters.

Experienced administrators

Usually experienced administrators set up an infrastructure to automate repetitive tasks (like adding users, creating mail aliases, etc.). This is usually accomplished using Perl or shell scripts. This kind of infrastructure is often poorly documented and the know-how is difficult to transmit. Comanche can make it really easy to create administration scripts and provide a graphical interface to them that can be then used by junior administrators.

Developers

It should help them providing a simple API, so they can concentrate on their original purpose: configure the application. The framework should take care of presenting an unified interface and interacting with the user.

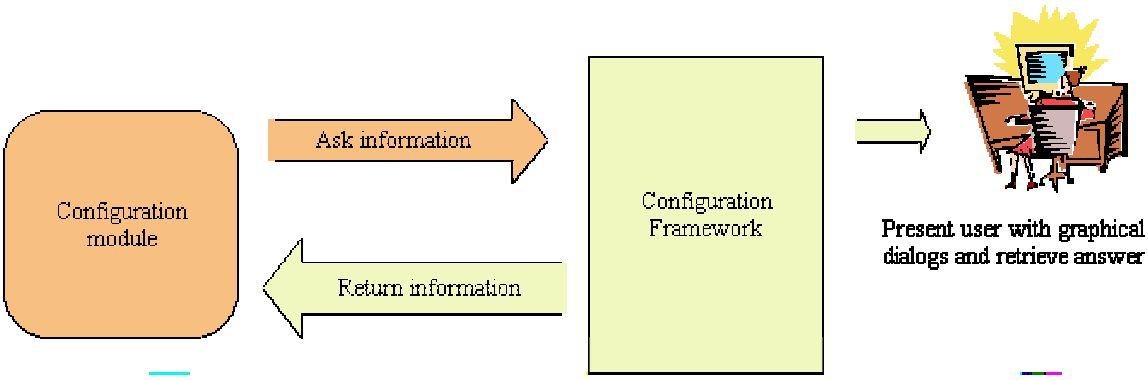
2. Architecture

This section is an introduction to the Comanche architecture.

Comanche is designed around a modular architecture and has the following characteristics:

Even the most complex configuration programs can be reduced to follow these principles:

- Present information to the user and ask him for input.
- Receive input from the user.
- Act upon that input.



Comanche takes care of the first two items and leaves the third to the module author. It can optionally provide libraries to help with this step, but it gives him absolute freedom. The module author, not the framework is who knows better how to parse configuration files or take certain actions (like execute a external program or check environment variables). As stated before, the goal is to provide as much freedom as possible to the module authors and present them with a small, simple API.

To present information to the user, instead of having to hard-code interface and dialogs in the program itself, they are described in a mark up language based on XML (XML is a mark up language to represent structured data). The interface can be easily manipulated or even generated on the fly. The XML representation is platform independent, it can be rendered using a traditional UI toolkit or in a web based interface.

2.1 Simple API for developers

A lot of time was spent in designing a simple API. The plug-in or management module has to know how to:

- Deliver the XML description of the configuration options. This information will be rendered and presented to the user, who will manipulate it.
- Receive the results from the user (also encoded in XML format).
- Extract the results and act upon them.

This can be reduced to the following APIs calls:

- RequestDocument
- ReceiveDocument

There are some others API calls for initializing the plugIn, etc. but these two are the main ones. The dialogs, if fixed, can be stored in text files or alternatively can be generated on the fly. In a way the plug-in acts like a traditional cgi-bin application. But instead of generating HTML , it generates user interface data encoded in XML, and instead of accepting form variables, it accepts XML encoded data.

2.2 Support for multiple extension languages

Comanche is developed mainly in Tcl, but that does not preclude future use of other languages for building extension modules (plugins). Since the interface is well defined, small and-text based it is possible to encapsulate the protocol into HTTP or Fast cgi. Most programming languages support these interfaces, including Tcl, Perl, Java, Python, C, C++, etc. This is still not implemented, but the basics are there.

2.3 Easy addition and maintenance of configuration files

XML is not only used for exchange of information. It is also used for describing the interfaces and the configuration directives. Since XML is a text based language, it is easy to change the definition of directives or rearrange user interfaces without need to recompile.

2.4 Easy maintenance of different interfaces (and languages)

Since the information can be separated from the presentation it is possible to maintain several versions of the interface, based on the mother tongue and/or skill level of the end user.

The architecture of Comanche gravitates around three blocks:

- Namespace. Can be thought of as a hierarchical database or a directory server.
- Plugins. They provide the configuration functionality. They register themselves with the namespace, and populate it creating nodes.
- The console. It is the graphical tool used by the user to connect to the different namespaces (represented by computer icons in the tree)

These elements are usually part of the same program, but the system has been designed such it should be relatively easy to have them running in separate machines or processes. They could be tied together using simple protocols like HTTP + SSL, thus someone running a console in one machine can securely administer remote boxes. The protocol is designed to work well over moderately slow links. Alternatively, the console instead of being a traditional application can be a web server module that renders the interface and interacts with the user. All of this is transparent to the module developer, which still sees the same simple interface described above.

Namespace

The namespace is the "central switch" of the Comanche architecture. It acts as a broker between the plugins and the console. It helps organize information about the plug-ins in a hierarchical manner, and to arbitrate communication between users and plugins (so an user can configure several plugins at the same time or several users can communicate with one plug in simultaneously). plugins register with the namespace and they create new nodes under existing categories (like network services or system).

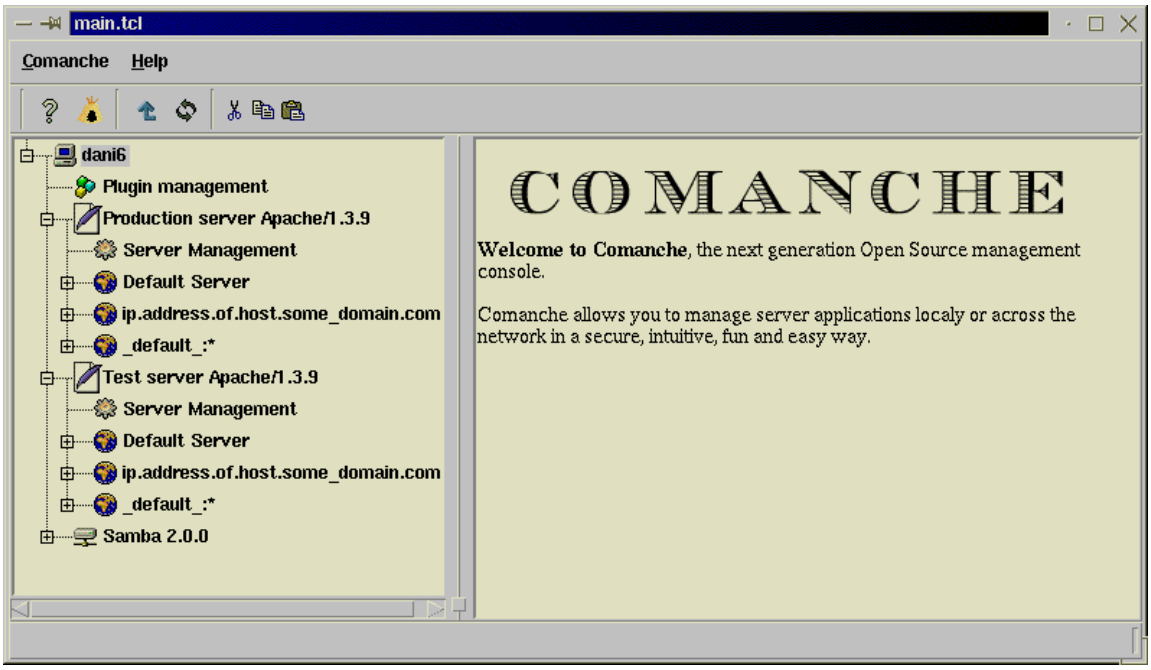
Plugins can also extend other plugins. To do so, plugins inform the namespace which nodes they are interested in. For example, there may be a need to add SSL (Secure Sockets Layer, necessary for secure communications between the browser and the server) configuration support to Apache. Instead of rewriting the existing Apache plug in, an extension SSL plugin is created. It tells the namespace its interest in all nodes created by Apache. Whenever the namespace receives a request for the property pages for a given node, the SSL plugin is also given an opportunity to contribute to the answer, so it can add new property pages. In case the SSL module is not installed or enabled, no new pages appear.

Console

The console is the end user interface. The console UI is divided in several areas, the main two are:

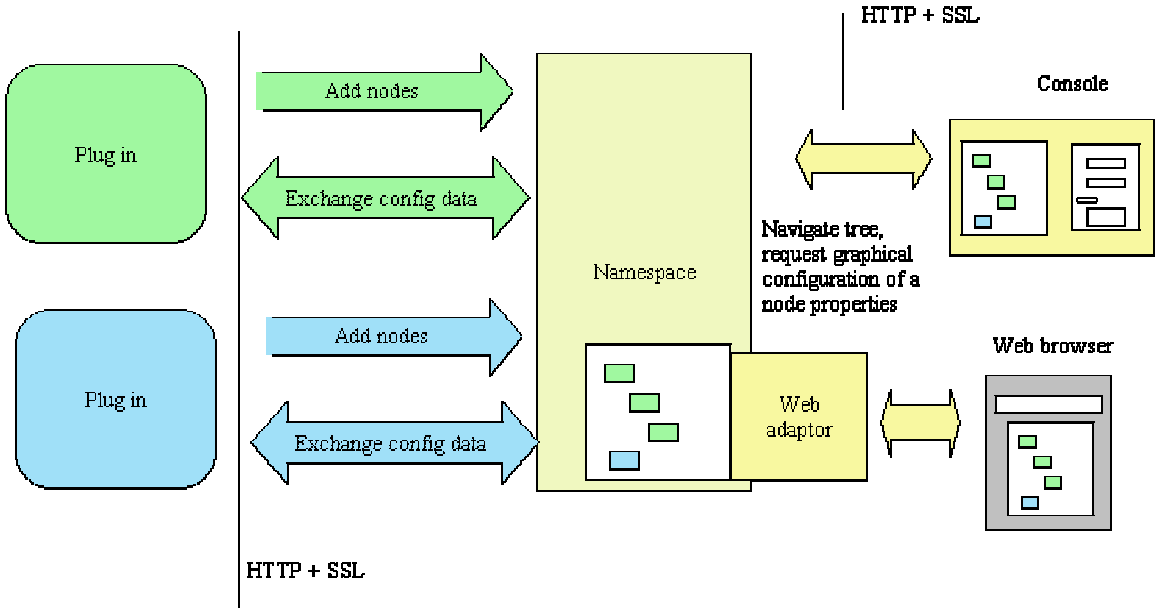
- **A tree structure on the left**. It allows the user to connect to the different managed machines, and navigate them by clicking and expanding nodes
- **A right pane**. It visualizes information related to the currently selected node.

The user navigates the tree on the left and configures the properties of the nodes by right-clicking on them and selecting the properties entry in the pop-up menu.



Plugins

Plugins are modules that implement the specific management behaviour. They are the ones that populate the namespace that is browsed by the console. They produce the content of the property pages that are delivered to the user and they act upon the received changes.



3. XML based configuration

This section explains introduces a configuration language based in XML. Traditionally different applications have stored their configuration settings in a variety of formats ranging from databases to text files, from the Windows registry to directory services like LDAP.

The most common configuration format employed by Unix applications is plain text files. The exact syntax employed inside the text file varies greatly from application to application: Samba uses a simple pair key/value Windows .INI format, Apache allows sections and nested subsections, Bind also structures information with sections delimited by curly braces, etc. There are advantages/disadvantages of having a text based configuration as opossed to a GUI, such as easy automation with scripts, remote administration, etc.

A XML based configuration language was designed with the following goals:

- **Simple**. An human should be able to read, modify or write from scratch the XML documents. Parsers for the configuration parser should be easy to write.
- **Universal**. The language should be general enough that it can be applied to describe the configuration settings of a variety of programs.
- **Extensible**. The language should admit easy generation of new configuration parameters.
- **Multilingual**. It should be possible to localize (translate) the information about the directives, help text, etc

- **Comprehensive**. By combining simple basic blocks it is possible to describe complex configuration directives.
- **Verbose**. The description of the directive should include information about the class of the directive (string, number, ...), range of values accepted, etc. This metadata helps the user interface and the parser when validating or displaying the data.

The XML configuration language is built on top of basic building blocks. These blocks represent parameters from a semantic point of view, for what they mean, not how they are represented. That is, if there is a directive "userName" that can accept the name of a person as its value, the directive is thought as being a directive of class string rather than thinking about the directive in terms of how would it be represented in a GUI (in an entry form). This decoupling between the meaning of the directive and the user interface representation is of great importance as it will become clear in the section about XML User Interface.

The current basic blocks are the following:

- string
- number
- boolean
- choice
- label
- structure
- list
- alternate

To understand the following sections, it is important that there are two components to consider:

- Configuration file: This is the actual configuration information of the program being manipulated. If it is a mail server it will have mail aliases, mailing list information, etc.
- Directive description: This describes what the tags found in the configuration file actually mean and how they can be nested, etc. If it is the mail server mentioned above, it will explain how the mail aliases are described or how to store the members of the mailing lists.

A simple XML configuration file of a fictitious server could be:

```
<server>
   <name>Mike's server</name>
   <ipAddress>10.0.0.1</ipAddress>
   <port>80</port>
</server>
```

But how it is possible to know, when a server tag is found, which elements are allowed inside, and which are the ranges of accepted values? We need a configuration directives metalanguage. The server directive could be described as follows:

```
<structure name="server" label="Server">
     </syntax>
         <string name="name" label="Server Name"/>
         <string name="ipAddress" label="IP address"/>
         <number name="Port" label="Port"/>
    </syntax>
</structure>
```

The syntax above will be explained in the following sections. It is important to note that although the XML standard has a similar mechanism for defining the structure of a document, called DTDs (Document Type Definitions) they are not suitable for this purpose. If only because DTDs define the structure of a document (which tag can appear where) but do not offer any information about the data hold by those elements. Other XML standards, like XML Schemas aim to solve this problem, but they were only early drafts at the time Comanche was written. As the standards mature a move to them will be considered.

3.1 String

A string element represents text values and has the following XML definition.

```
<string name="stringName" label="This briefly describes the
directive">
   <default>Some default value</default>
</string>
```

This directive will be represented in the XML configuration file as:

```
<stringName>This is some value</stringName>
```

If the directive stringName does not appear in the configuration file, it will be assumed to have the value "Some default value".

The default value and tags are optional. A graphical representation could be the following:

3.2 Number

A number elemenet represents numbers and has the following XML definition.

```
<number name="numberName" label="This briefly describes the
directive">
   <default>123456</default>
</number>
```

This directive will be represented in the XML configuration file as:

```
<numberName>123456</numberName>
```

If the directive numberName does not appear in the configuration file, it will be assumed to have the value "123456".

The default value and tags are optional A graphical representation could be the following:

3.3 Boolean

A boolean element can only hold two values, true or false.

```
<boolean name="booleanName" label="Boolean label">
   <default>1</default>
</boolean>
```

In the XML configuration, it will appear the following

```
<booleanName>1</booleanName>
```

A graphical representation could be the following:

3.4 Choice

The values can be one of a collection of fixed ones

```
<choice name="choiceName" label="Choose a fruit">
   <syntax>
     <option name="orange" value="Juicy orange" />
     <option name="lemon" value="Acid lemon" />
   </syntax>
   <default>orange</default>
</choice>
```

This directive will be represented in the configuration file as:

```
<choiceName>lemon</choiceName>
```

If the directive choiceName does not appear in the configuration file, it will be assumed to have the value "orange".

The default value and tags are optional A graphical representation could be the following:

3.5 Label

This element represents a fixed value. It will be used in other composite elements.

```
<label name="labelName" label="This is the value of the label" >
</label>
```

The directive will be represented as:

```
<labelName>This is the value of the label</labelName>
```

A graphical representation could be the following:

3.6 Structure

This is an element that is a composite of others. The directive is composed of other directives. The general XMl description of the directive is as follows:

```
<structure name="structureName" label="Description of the structure">
   <syntax>
      (... Here comes the XMl description of the structure
      components ..)
   </syntax>
</structure>
```

This is better explained through an example. Let's assume a directive "person", which is composed of three other directives: a name (string), a surname (string) and an age (number)

The description would be:

```
<structure name="person" label="Description of a person" >
   <syntax>
      <string name="name" label="Name" />
      <string name="surname" label="Family name" />
      <number name="age" label="Age">
         <default>21</default>
      </number>
   </syntax>
</structure>
```

A representation of an instance of this directive in the configuration file would be:

```
<person>
  <name>John</name>
```

```
  <surname>Smith</surname>
  <age>30</age>
</person>
```

There is no default for the directive itself, but rather each one of the elements defines its own default.

A possible graphical representation:

3.7 List

A list is a collection of elements of the same type. The list element definition (the part inside the syntax tags) can be itself be described in terms of other basic building blocks.

```
<list name="listName" label="Comment about the list">
   <syntax>
      (... XML description of the list elements  ...)
   </syntax>
   <default> (... depends on the list element ...) </default>
</list>
```

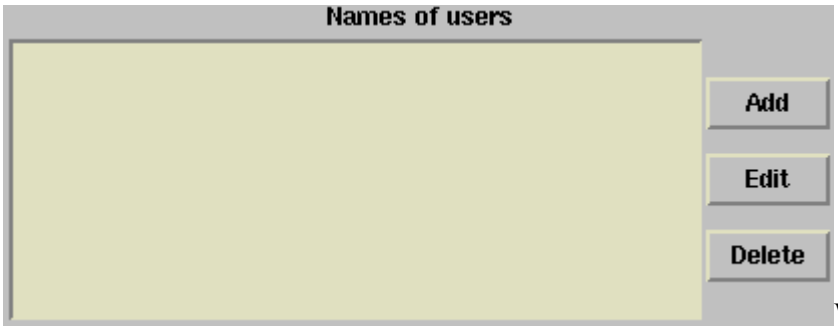
The following example illustrates the use of the list element:

```
<list name="userNames" label="Names of users">
   <syntax>
      <string name="user" label="Name of the user" >
         <default>nobody</default>
      </string>
   </syntax>
   <default>
      <item>dani</item>
      <item></item>
   </default>
</list>
```

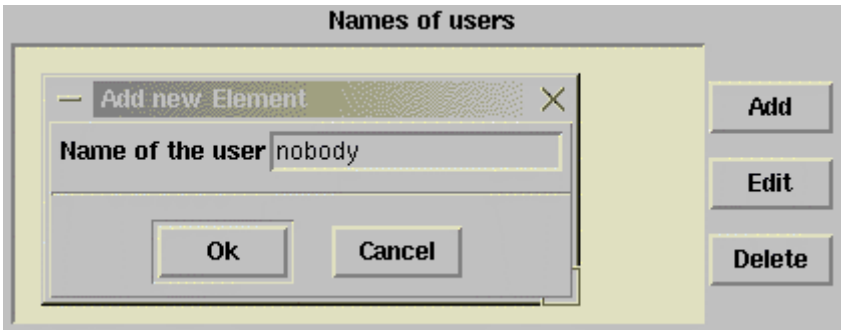
An entry io the XML conf file that uses this list would be:

```
<userNames>
   <user>user1</user>
   <user>user2</user>
</userNames>
```

A possible graphical representation:



When the user adds an element:



3.8 Alternate

This element is similar to the structure element, but in this case instead of being a collection of elements, it is one (and only one) of the elements.

```
<alternate name="alternateName" label="some text describing the
alternate" >
   <syntax>
      (... XML description of the possible elements  ...)
   <syntax>
   <default>
     (... XML description of the element and value   ...)
   </default>
</alternate>
```

As an example, the Apache bind directive is implementated as an alternate

```
<alternate name="bindAddress" label="Address to bind the server" >
   <syntax>
      <label name="allAddresses" label="All available addresses" />
      <string name="specific" label="Specific address/domain-name" >
      </string>
   </syntax>
<default><specific>127.0.0.1</specific></default>
</alternate>
```

A possible graphical representation:

In the XML configuration file, a bindAddress directive would look like:

```
<bindAddress>
   <specific>10.0.0.1</specific>
<bindAddress>
```

or if the web server is going to listen to all addresses available:

```
<bindAddress>
   <allAddresses/>
</bindAddress>
```

These are the basic building blocks that, combined, can be used to create arbitrarily complex directives. Yet the rules are very simple.

These XML elements are manipulated in Tcl in a special way, that isolates the XML syntax details (that may change in the future) and allows similar elements to be accessed in a similar way (i.e reading a string or numeric value is done in a similar way)

The process is the following: a XML document containing the directives is parsed using a XML parser. This XML parser's callbacks are used to construct a Document Object Model representation of the XML document. A DOM representation of a XML document is a tree like structure of the document that can be accessed programtically. For example, the following XML document:

```
<server>
    <name>Daniel</name>
    <surname>Lopez</surname>
</server>
```

Its DOM representation would be:

```
server (element)
|
|---name (element)
|     |
|     \_ text node (value="Daniel")
|
|---surname (element)
|
 \_text node (value="Lopez")
```

The DOM implementation allows navigation of the document. To get the value "Daniel" the steps are as follows:

- · Ask for the first node. It returns as a reference to "server"
- · Ask for the children of "server", get a reference to "name"
- · Ask for the children of "name", get reference to the text node
- · Get the value of the text node: "Daniel"

Manipulating XML documents this way can be cumbersome. A new method was devised to transition from XML documents to a DOM tree and from a DOM tree to xuiObjects. What is a xuiObject? It stands for XML User Interface Object. It is a Tcl object that encapsulates the functionality of its corresponding XML element. After those objects are created, it is possible to manipulate them easily, combine them and serialize them back to XML.

That is, a xuiString encapsulates the functionality of a XML string element. This functionality can be accessed invoking different methods:

If the variable xuiStr contains the name of an object instance of a xuiString class it is possible to do the following in the Tcl programming language:

```
$xuiStr getValue
```

The previous Tcl code invokes the method getValue on the object referenced by the xuiStr variable. The result of the invocation will be the return of the value of the string. If the string was defined as:

```
<someString>Some value<</someString>
```

And xuiStr contained a reference to an object that represents that someString XML code, getValue would return "Some value".

Similarly,

```
$xuiStr setValue "Another value"
```

Will set the new value for the string. If the object is serialized back to XML, the result would be:

```
<someString>Another value</someString>
```

If the xuiObject is a xuiList, it has methods available to create, add, remove certain elements, etc.

If the xuiObject is a xuiList, it has methods available to create, add, remove certain elements, etc. Assuming $myList is a xuiList of strings defined by the following XML declaration:

```
<list name="myListName" label="Some comentary">
   <syntax>
      <string name="someStringName" label="some label" />
   </syntax>
   <default>
      <item>bla</item>
```

```
   </default>
</list>
```

If $MyList has just been created it has a default content of one children with the value "bla", which XML representation would be:

```
<myListName>
   <item>bla</item>
</myListName>
```

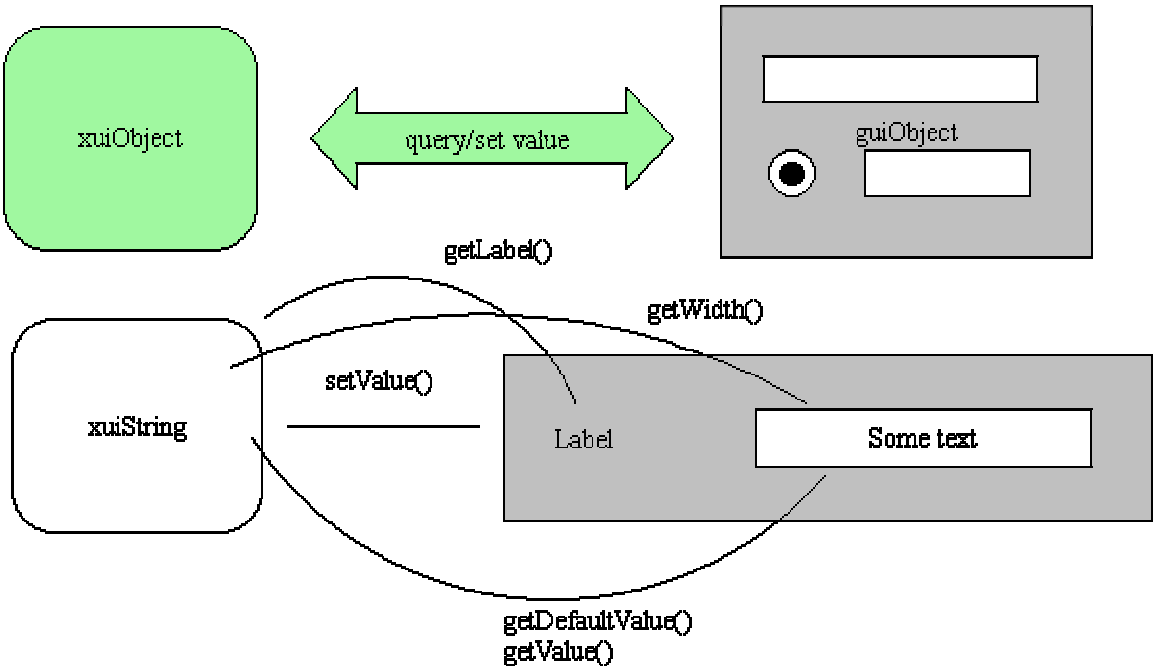
```
set childList [$myList getChildren]

# In Tcl, code in brackets [] is executed and the result is
substituted.
# The previous statement works as follows:
# - Invokes the method getChildren on the $myList object
# - Store the result on the childList variable
# Now childList contains all the children of $myList, in this case
only one,
# a xuiString element with value "bla"

puts [$childList getValue]

# puts is the Tcl command for printing a value to the standard output
# The code inside the brackets gets the value of the list element, in
this
# case it will print "bla" since we have a single element, which has
that
# value.

set newChild [$myList newChild]

# The newChild method invocation creates a new list element and the
# reference is stored in the newChild variable

$newChild setValue "foo"

# This new child, which is of the type xuiString, is asigned the
value "foo"

$myList insertChild $newChild

# Finally, the child is inserted at the end of the xuilist.
# In summary: A new children has been created and inserted in the
list.
# If the object was serialized, the result would be the following:

<myListName>
    <item>bla</item>
    <item>foo</item>
</myListName>
```

This distinction between the XML representation of the object and the object itself is very important for a number of reasons:

- The XML document description specifics may change in the future. The XML objects are used widely through Comanche. Small changes on the syntax of the XML document would have wide impact and would require a multitude of other changes. The encapsulation on Tcl objects provides a level of abstraction that isolates most of these changes from the rest of programs.
- Similar objects can be manipulated in a similar way (string, number, etc.). From the point of view of the programmer, it is assigning a value to an object via methods, with no required knowledge of how the XML output will look.
- Objects can be easily manipulated and then rendered back to XML (or to some other format, like HTML). The rendering logic is separated from the object logic. The same object (a xuiString object) can be rendered in different GUI controls. This is the topic of the next section.

4. XML User Interface

Tcl, together with the graphical toolkit extension Tk was chosen to provide a Graphical User Interface for Comanche. As seen in previous sections, the interface can be described in terms of a mark up language based on XML. The interface description can be parsed and abstracted into [incr tcl] objects (see Glossary on Tcl and [incr tcl]). This intermediate abstraction layer allows for different rendering engines. The rendering can transform the objects into DHTML (Dynamic HTML) that can be rendered by a web browser or into a traditional GUI representation. It is possible for the front ends to issue callbacks and manipulate the Tcl object. If XUI objects share the same interface (like string, number), the same GUI object class can manipulate them. Conversely, the same xui object can be rendered differently by different GUI object class: an element of the type structure can be rendered like a collection of property pages or as several groups of directives.



Examples of guiInterfaces are :

4.1 guiString

Can render xuiString and xuiNumber elements.

4.2 guiLabel

It is used to represent elements of the type label

4.3 guiLabeled

All the elements that have a label inherit from this guiObject class. It provides for text alignment of the labels so the interface looks nice.

4.4 guiBoolean

It is used to represent elements of the type guiBoolean

4.5 guiImage

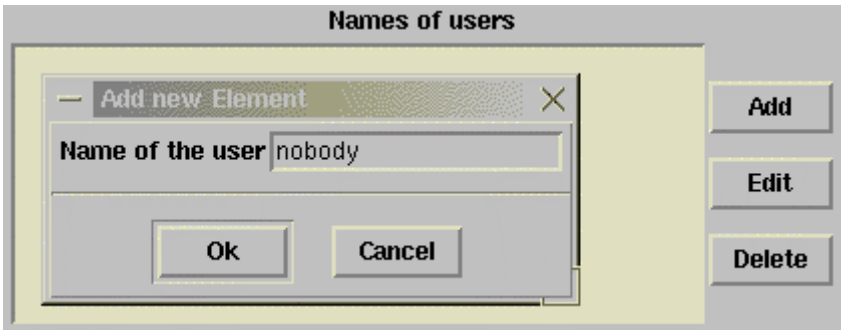
It is used to represent images

4.6 guiChoice

It is used to manipulate xui elements of the type choice. It is generally represented as a combobox. It could easily be represented as a collection of radiobuttons.

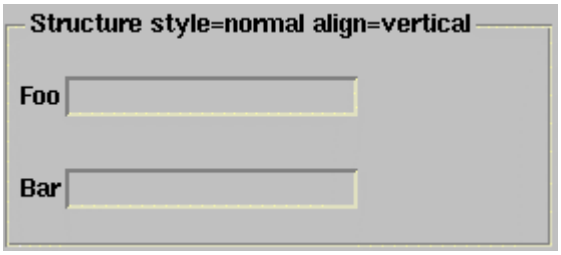
4.7 guiList

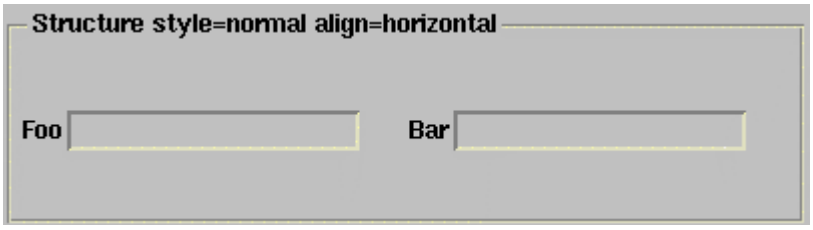
It is used to manipulate list objects



4.8 guiStructure

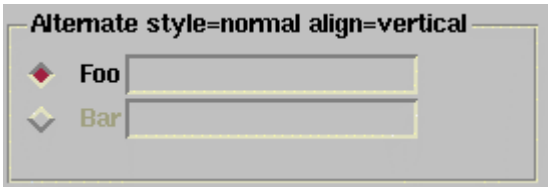
Is is used to manipulate xuiStructure elements. xuiStructure elements can represent compound directives or groups of other xuiElements. Depending on certain attributes being present (style) groups of directives can be represented in different ways: horizontally or vertically, surrounded by a labeled frame or with no decoration.





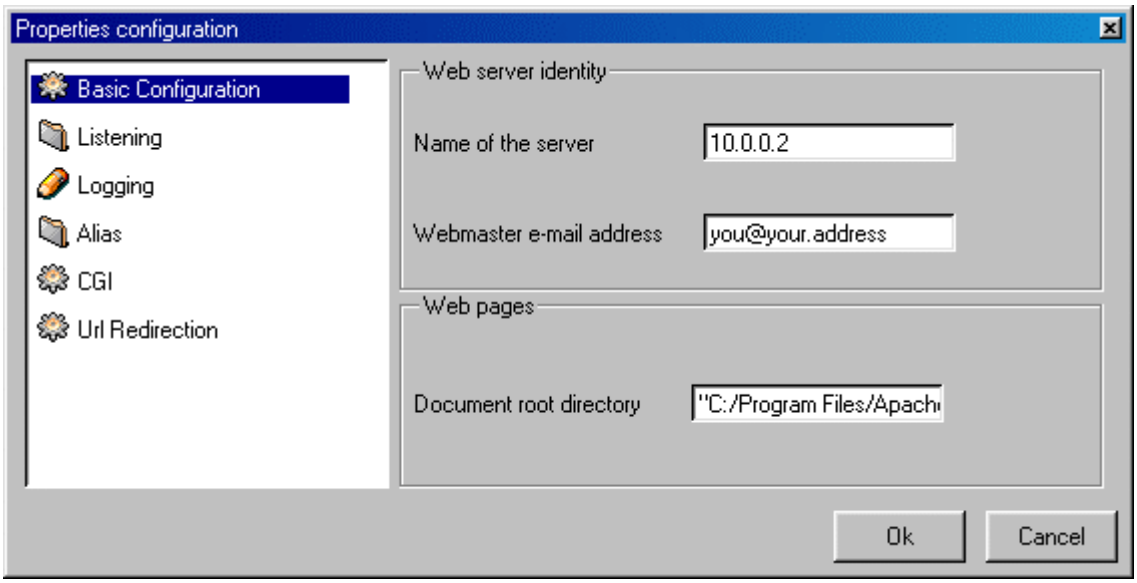
4.9 guiAlternate

Is is used to manipulate xuiAlternate elements.



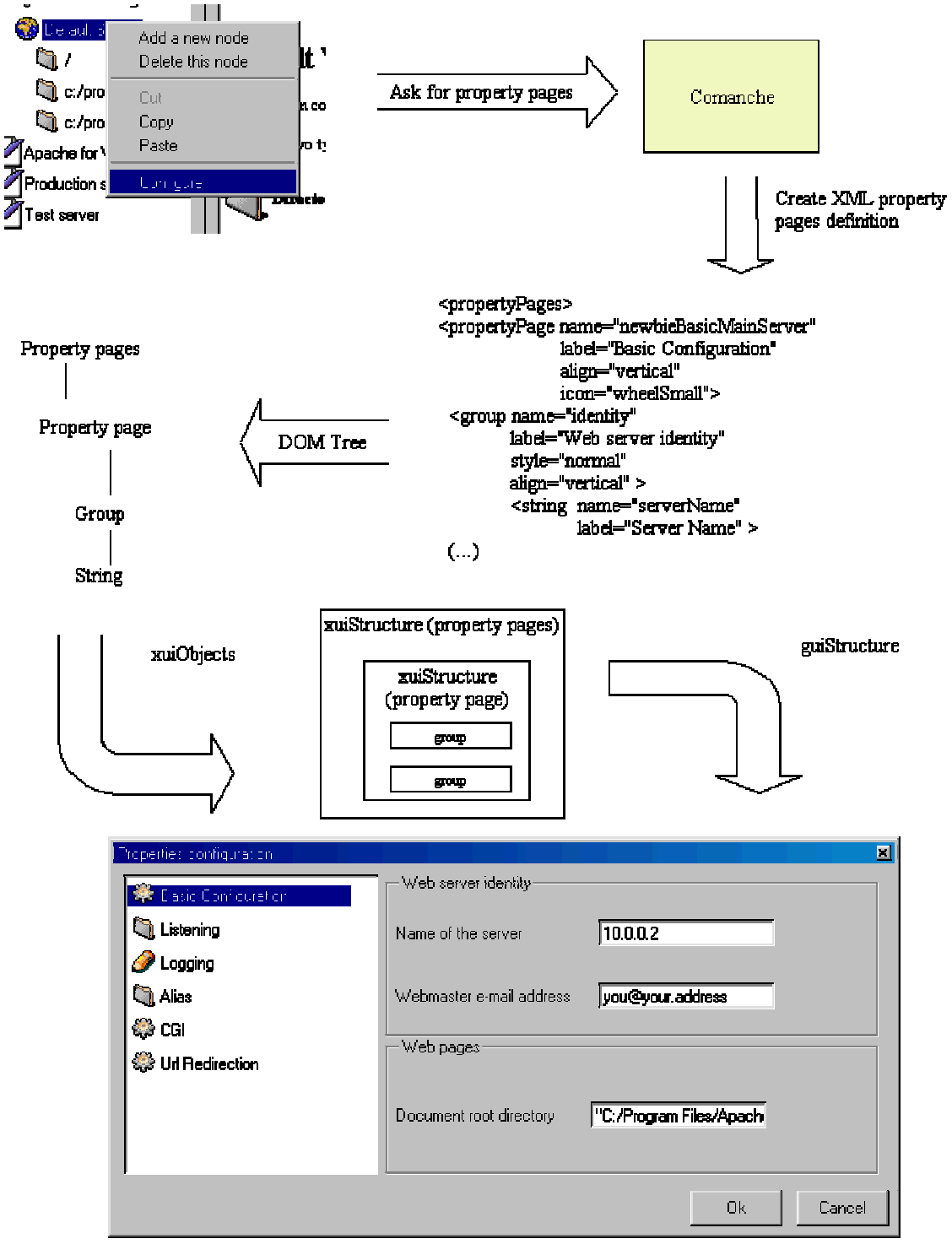
4.10 guiPropertyPage

This gui element manipulates xuiStructures interpreting them as property pages:



4.11 How it works

This section describes how the process works from the instant that the user requests some information to the point that the user is presented with a property page that he can fill and return the information to the plugin.



- User selects to view the properties of a given node, by right clicking on the node and selecting the "Configure node" entry from the menu.
- The message is sent to the namespace, which contacts the appropriate plugins and generates the property pages document, which it is a XML description of the property pages.
- The console receives the document, transforming it into a xuiObject of the type xuiPropertyPages (the XML definition is transformed into Tcl code).

- The console needs to allow the end user to manipulate the xuiPropertyPages object and for that purpose creates a guiPropertyPages (presentation) object and connects it to the xuiPropertyPages object (data).
- The guiPropertyPages object is passed the frame where it has to display its information, the propertyPages object and a reference to an object factory. It creates a listbox menu on the left and a notebook on the right. For each one of the property pages (elements of the xuiStructure), it creates an entry in the listbox (with an additional image if the attribute icon is present) and uses the guiObject factory to render the property page on the notebook.
- When an element is selected on the listbox, the appropriate property page is displayed on the right notebook.
- When the user is done, it can press the Ok button. The GUI representation can be destroyed then, but the xuiPropertyPages object will have the modifications performed by the user.
- The xuiObject is then passed to the namespace, which will select the appropriate property pages and deliver it to the plug ins so they can act upon it.

5. Distributed architecture

The architecture of the system has being designed with distributed operation in mind. This translates into the following ideas:

- Few, well defined API calls. The idea is to reduce the traffic over the net and to simplify the implementation of different front end clients.
- The inter process protocol is based on XML, which in turn is based in text and can be easily transported over other protocols and manipulated by other languages.
- There is no notion of a single user. Requests can be server concurrently in a web server like fashion. Little or no state is kept.

There are still issues that need to be addressed like delegation of authority and how to prevent administrators working concurrently on a configuration from interfering with each other

5.1 Different models for distributed operation

At least two models have been considered for developing an architecture that would allow for remote administration of machines

Web based approach

Namespace and plugins would reside in the remote machine. A web front end would also reside in the same machine and would accept requests from client browsers.

Distributed application approach

The other approach is to encapsulate the protocol between the console client and the namespace over HTTP and have them reside in different servers. This would require

installation of a Tcl client in the administrator machine, but would allow centralized administration from a single machine (the web based approach would require connecting to each machine that requires administration)

5.2 Few, well defined API calls

The API calls between the three architectural blocks are few and well defined, as explained in the following sections:

Namespace / Plug-in

The communication that takes place requires the following information to be exchanged.

The plugins need to register with the namespace and explain which nodes it is interested in extending, etc.

```
registerPlugInInterests
        The information that the plug in would provide would be
        name
        version
        description
        node types that it provides
        node types that it extends
        category the plug in belongs to: network services, user
management,
        system management
```

The plugIns needs to query the tree structure, add and remove nodes, etc The API functions to perform that are:

```
    getRootNode
    addNode
    configureNode
    removeNode
    getChildren
```

The namespace needs to request and deliver information from the plug in

- deleteNodeRequest (When the user wants to delete a node)
- requestXuiDocument (The user requests a document. it can be a property page, a wizard or a right pane content)
- answerXuiDocument (the user has filled some information and it has to be returned to the plug in for processing)
- populateNodeRequest (The user is exploring a node and double-clicks on it, the namespace takes note and urges the plugin to add nodes. This allows for dynamically generating trees (useful for navigating a server's filesystem, for example.)

Namespace / View

Similarly, the view needs to access the namespace, basically for the same purpose: query the tree structure, request information for display to the user and deliver back the user feedback. For those purposes it uses the previously detailed functions. In addition, the namespace informs the view when certain events occur: a node has been added or modified, etc

Also the namespace keeps track of which view has browsed which nodes and thus avoid informing the views of event regarding nodes the user has not yet browsed.

A future option may to request not to be notified of updates, and have the user refresh the display when necessary. This may be useful for slow links or the web based interface.

5.3 Inter process protocol based on XML

The interprocess communication that takes places is based on XML. The data ,object and method invoked are encoded in XML.

The interprocess communication is hidden in the infrastructure. The API offered to the module author is identical, no matter if the plug in is being used locally or remotely. When a component of the system talks to another component, it does so using xui objects. The system keeps track if the component that is being called is remote or local, if it is local, it directly passes the xuiObject to the called entity. If the entity is remote, it performs a remote call and serializes the object into XML. At the other end, the serialized object is transformed again into a xui Object.

How does the system know if the object called is remote or local? First, objects must register themselves before being able to invoke / receive any methods. If the object is accessible locally (for example, namespace and plug ins are living in the same Tcl interpreter) nothing is done and future communication takes place directly. If the object being registered is accessing the system remotely, a "fake object" will be created that will remember how to access the remote object. This fake object will then be accessed normally as a local object by the rest of the system. When a method is invoked in this fake object, it will in turn take the arguments, the method and the identity of the caller, serialize them and sends it to the remote object. This also involves timeouts (that can be tuned depending on the situation) so if the remote end becomes unavailable the application will get informed and the object will get deleted.

All this process is greatly simplified by the fact that arguments are passed as xuiObjects, which are composed of only a few building blocks. Thus arbitrary functions can be called with arbitrary arguments, since the system knows how to serialize them. This allows for greater flexibility, since this generic mechanism avoids:

- having to declare each one of the possible functions.
- having to explicitily distinguish between remote and local operations.

The XML protocol can be encapsualted in a variety of transport protocols:

- Over HTTP, both directly (using GET and POST methods) or over XMl-RPC
- Plug-In communication with the namespace could also be done using a Fast-CGI approach.

Note: none of the above is yet implemented. Comanche can only run as a local application right now.

5.4 Concurrency handling

The namespace server will act in a similar fashion to a web server, in the sense that it can serve requests simultaneously. In fact, initial feasibility tests where performed using the tclhttpd web server. In both cases, XML-RPC was used as the underlying communication mechanism. Requests are served in a first come first served basis. There is a single process running and speed is not likely to be an issue (the network part is usually the bottleneck. Because of that network transmission is done using fileevents (fileevents is a Tcl feature that allows serving of multiple requests using callbacks to detect when a socket has received new data or the data scheduled to send has been effectively transmitted).

Concurency means several problems need to be addressed: what happens when two different users are configuring the same application or where the same user configures the application using different windows (in the case of the web interface, opening several browser windows). There is the potential for the following scenarion to happen: Administrator A selects property pages for virtualhost v1. Administrator B selects property pages for the same virtual host. Administrator B presses OK and commits the changes. Administrator A presses OK and commits the changes.

The following can happen:

- Administrator A will overwrite administrator B changes. Administrator B does not even know that. This is Bad
- Changes could be merged a la CVS style. But the concept of merging changes is more ambiguous here. Merging could also lead to inconsistencies.

Alternative, more desirable solutions are the following:

- Only one admin can be editing a node/service at a given time. Users with enough rights should be able to kick out other admins to avoid deadlock situations (the administrator browser crashed, but the admin appears to the system as still logged, preventing anyone else from administering the machine). Some schema of auto-logout after a period of inactivity could also be implemented.
- More than one admin can be logged and editing the same node. If the previously described situation with admin A and admin B occurs, the solution is to prevent Admin A to commit its changes, informing it that the node has been modified in the mean time and the information is no longer valid.

5.5 Delegation of authority

Currently there is no concept of users or privileges in Comanche. It needs to run with the privileges required to edit by hand the configuration files of the programs it is configuring. It would however, be interesting to have some authentication and delegation schema for certain situations: an ISP may be hosting hundreds of web sites as virtual hosts for their customers. In the current situation, the customer must explain what changes it needs to make to the configuration files and the ISP staff performs that for them. This has an obvious administrative overhead and slow turn around time. This problem is partially solved currently :

- **Using Frontpage server extensions**. This allows customers to use proprietary Ms tools to configure and maintain their web servers. This extensions have a track record of security problems and messy code, so they are not very popular with ISPs, which however have to install them due to customer demand.
- **Use .htaccess files**. These files allow per directory configuration files If its used is enable, Apache will look for every one of these files and apply the parameters that it finds. They are used for example to allow users to specify password protected pages and directories. .htaccess files are however, a serious performance hit for highly loaded servers

In summary, delegation of authority is an interesting feature, but poses a series of challenges that are out of the scope of a first implementation of Comanche. The architecture, however, if flexible enough to implement such hooks for authentication and delegation. These controls could be placed when views register with the namespace, when xuiRequests and xuiAnswers are requested, etc.

6. Programmer tutorial

This section guides a programmer in the process of writing a simple plugin for Comanche. The purpose is to describe the APIs that module authors should know and give examples of how they can be used. Although the module is written in Tcl, knowledge of Tcl is not strictly necessary or assumed. The code is extensively commented and explained to guide the reader.

6.1 Creating a module.

The main tasks that a Comanche module has to carry out are:

- Read any internal configuration and initialize itself
- Answer requests for information
- Accept answers from the client

We will develop a simple module. This module queries the hostname of the machine and allows the user to change it. To do so, the plugin will rely on the "hostname" system command. In certain operating systems, like Red Hat Linux, changing the hostname permanently involves changing some text files that get read at startup. Since this is just a demonstration of how to write a simple module for Comanche, we will

not worry about that. This simple plugIn will add a node to the Comanche console. When the user clicks on the node, a page on the right will appear that gives the current hostname. When the user right clicks on the node, a menu will appear that allows the user to pop up a property page to change the hostname value.

Every Comanche module should be designed as a [incr tcl] module (this is not necessary if it is done via the remote plugin interface, which is not implemented in this version and that allows plugins to be written in a variety of languages)

[incr Tcl] is an object oriented extension of Tcl. It allows you to create classes which define objects. Objects have functions that can be called on the object and that are called methods. We could define a class dog, which represents dogs in abstract. We could define a method, bark, that when invoked would print "Barf!" on the screen. In [incr tcl] this is done in the following fashion:

```
class dog {

   method barf {} {
       puts "Barf!"
   }

}
```

We can create an object called scooby, which is an instance of the class dog.

```
dog scooby
```

Now we can tell scooby to bark:

```
scooby bark
```

and we get:

```
Barf!
```

The skeleton of the plug in looks something similar to the following:

```
class hostnamePlugIn {
    inherit plugIn
}
```

In a similar fashion to the above, we are going to be creating an object of the class plugin. When we have a plugin, we can tell it to do certain things for us: we can tell it to add nodes to the namespace, we can ask it information about nodes that belong to it, etc.

The kind of information that we ask is usually property pages for displaying/modifying the plugin settings. Most of the work in a plugin resides on the design of these property pages.

We are inheritting from the plugIn class, which implements the following methods:

```
  method init { args }
  method requestXuiDocument { xuiData }
  method answerXuiDocument  { xuiData }
  method deleteNodeRequest  { xuiData }
  method populateNodeRequest { xuiData }
```

From all these, the only ones that we need to implement are the first three ones, since we only have one node (populateNodeRequest is the way the namespace tell us to add nodes that are children of another) and we do not want to delete it. The remaining three functions (init, requestXuiDocument, answerXuiDocument), deal with initialization routines, and passing/getting information to/from the user.

6.2 init

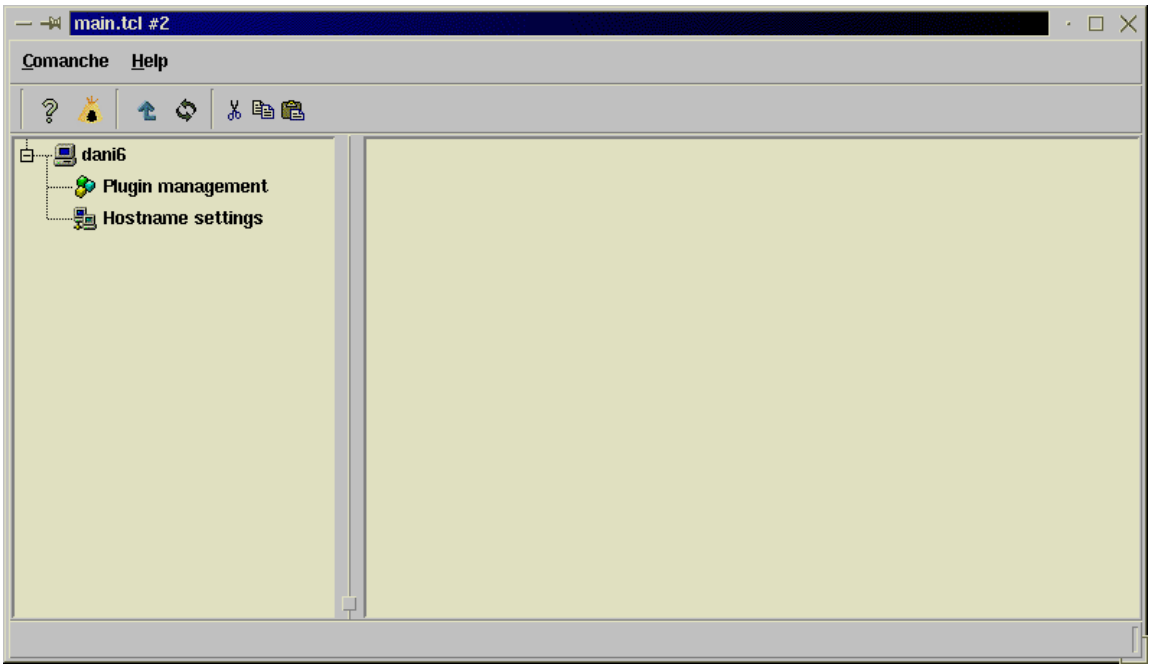
This function will get called at initialization time. It gives our plugin a chance to initialize internal data structures, read external files, etc. and finally add nodes to the namespace if necessary. There are several helper objects that can be used when managing many nodes. Since we are adding a single node, it is easier to add it directly and keep track of where we added it in a variable.

```
# args contains the options that are passed to the plugIn at
initialization
# time.
#
#  -namespace    contains the name of the name space server


method init { args } {

    # args is a list of pair/value options
    # The following is to convert the list to an array, called
options

    array set options $args

    # This is the way Tcl assigns a variable value
    # Now namespace contains the value of the element -namespace
    # of the array options

    set namespace $options(-namespace)

    # The [] tell Tcl to treat the text contained in the brackets as
a
    # command, execute it and substitute the result. So the sequence
of
    # events is as follows:
    # - Ask the namespace for the root node (will return a xuiNode
object)
```

```
    # - Get the unique id number for that node
    # - Assign that value to the hostnameNode variable
    #

    set parentNode [[ $namespace getRootNode] getId]

    # Add a node to the namespace, we need to tell the namespace:
    # - Who we are: $this
    # - Which namespace we want to hook up under: $namespace
    # - Which node we want to hook the new node under: $parentNode
    # - Several icons for open and closed
    # - Classes: List with node classes. Leaf means that it cannot
have
    #           children. Hostname means that it belongs to our
plugin.
    # - Label: Text that will be displayed next to the icon

    set hostnameNode [::plugInUtils::addNode $this $namespace
$parentNode \
        -classes {hostname leaf} \
        -openIcon networkComputer \
        -closedIcon networkComputer \
        -label {Hostname settings}]

    # We remember the Id of the node that we just added

    set hostnameNodeId [$hostNameNode getId]
}
```

We add a couple of variables to the plugIn, to also store the id for the node just added and the name of the namespace

With just the above, the plugin will add the node to the namespace:




```
 class hostnamePlugIn {
```

```
    inherit plugIn
    variable namespace
    variable hostnameNodeId
}
```

By declaring the variables in the plugin class, we make sure that they are persistent and accessible when other methods are called.

Next step is to implement the rest of the functions required for displaying menus, right pane contents and a pop up property page. When we receive/send a XML document using

`requestXuiDocument`
or

`answerXuiDocument`
we have to specify the kind of document we are receiving/transmitting (menu, property page, etc)

This involves processing xuiStructures for storing the answer, etc. We can save ourselves a lot of trouble if we directly inherit from the basePlugIn class, which already takes care of many of those details.

The basePlugin class defines the following methods:

```
method _inquiryForPropertyPages  { node }
method _inquiryForMenu { node }
method _inquiryForWizard { type node }
method _receivedWizard { type node }
method _inquiryForRightPaneContent { node }
method _receivedPropertyPages { node xuiPropertyPages }
method _receivedCommand { node command }
```

We are going to provide content now for each one of the functions and we will be one step ahead in building our plugin

`_inquiryForRightPaneContent`

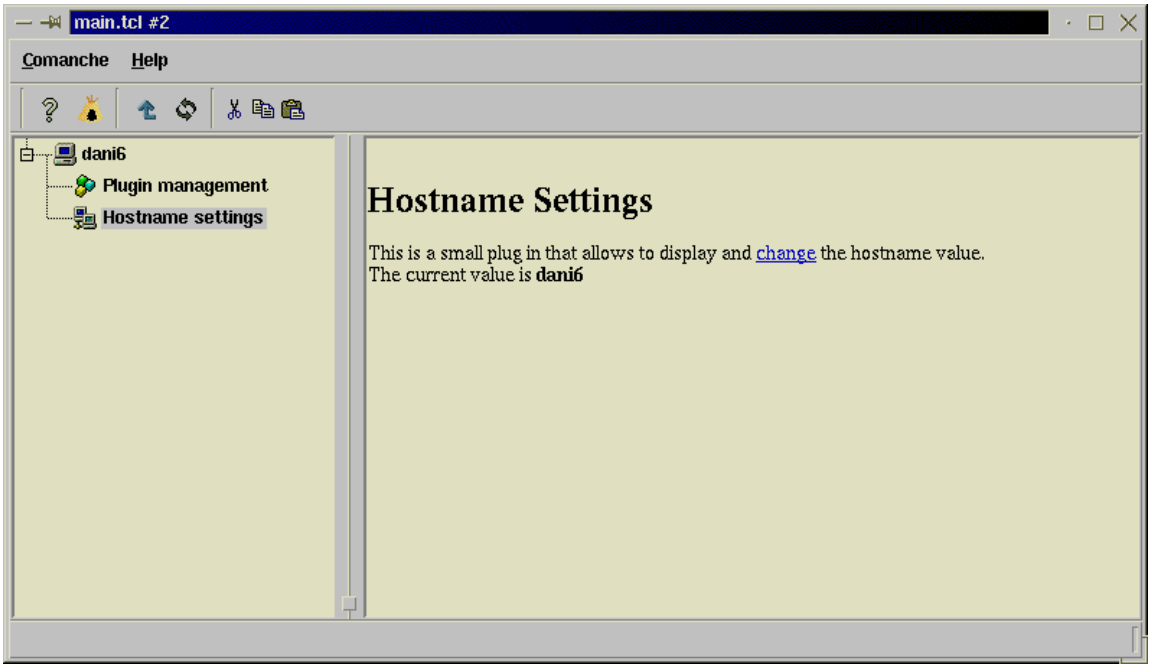
This function takes as an argument the node for what the content is being requested. It must return the HTML-like text to be displayed in the right pane portion of the interface. Since our plugin only has one node, it is safe to assume that when the function is called the node is the right one, so we do not need to double-check it. If a plugin had more than one node, it would be necessary to distinguish between them.

The function is then, simply:

```
body hostnamePlugIn::_inquiryForRightPaneContent { node } {

    # Set the variable result to a snippet of HTMl-like code
    # The link, instead of a normal HTML link is a command directed
```

```
    # to the console. In this case it tells the console to show
    # the property pages for the selected node when clicked.

    set result {
        <h1>Hostname Settings<h1>
        <br> This is a small plug in that allows to display and
        <a href="command propertyPages"> change</a>the hostname
value.<br>
        The current value is }

    # Current value is given by executing the system command hostname

    append result [exec hostname]
    return $result
}
```

The previous addition will show in the console as follows:



Menu generation is still not implemented, there is a generic menu in place. When the user clicks on the menu entries, nothing will happen except when the user selects "Configure node". This will trigger the

`_inquiryForPropertyPages`
method

For returning property pages, instead of creating a new property page object per request, we will keep a property page and update it every time it is requested.

We add the following to the plug in definition

```
variable hostnameXuiPP
variable hostnameEntry

constructor {} {
```

```
   # We create a global  object of type xuiPropertyPage
   # the #auto keyword will assign an arbitrary name.
   # This is necessary because if we hardcode the name, this would
   # prevent having two instances of the same plugin

   set hostnameXuiPP [xuiPropertyPage ::#auto]

   # Set default icon, title and name of the property page

   $hostnameXuiPP configure -icon network
   $hostnameXuiPP setLabel {Configuring hostname}
   $hostnameXuiPP setName hostnamePP

   # Create the xuiString object that will hold the hostname value
for the
   # user to modify ...

   set hostnameEntry [xuiString ::#auto]
   $hostnameEntry setLabel "Hostname"
   $hostnameEntry setName hostname

   # ..and add it to the property page

   $hostnameXuiPP addComponent $hostnameEntry
}
```

The constructor method is called everytime a hostnamePlugIn object is created. It set ups a XUI property page object. Every time they ask us for a property page, we fill the current hostname and we return the property page back. When it comes back, it will contain the data, probably modified by the user.

The following methods perform just that:

```
body hostnamePlugIn::_inquiryForPropertyPages  { node } {

    # User is asking for a property page to display for this node
    # We set the current hostname in the entry

    $hostnameEntry setValue [exec hostname]

    # We return the property page

    return $hostnameXuiPP
}

body hostnamePlugIn::_receivedPropertyPages { node xuiPropertyPages }
{

    # We extract the appropriate property page from the xuiStructure
    # containing the property pages.
    # (there is only one page, but we ask it by name)

    set pp [$xuiPropertyPages getComponentByName hostnamePP]

    # From that property page, we get to the string containing the
hostname
    # and get its value
```

```
    set newHostname [[$pp getComponentByName hostname] getValue]

    # Change the hostname to the one supplied by the user

    catch {exec hostname $newHostname}
}
```

And that is all, the plugIn is completed, we do not care about the rest of available functions by now (asking for wizards, etc...), since the plugIn is a simple one. We need now to package the plugIn in a certain way so Comanche can discover it and load it at start up. Comanche stores modules under the subdirectory modules/ Under modules, each directory contains a plugIn. For each plugIn, a special file called init.tcl will get sourced.

The module needs to define certain functions that will get called at the appropriate time:

```
modulename_init
modulename_restart
modulename_info
modulename_unload
```

Using the module name is a convention. It will probably be replaced by use of Tcl namespace facility, just not yet.

By now we only define the modulename_init function, in this case hostname_init, that will get called with the following arguments -namespace namespaceObject

```
# This file will get sourced when Comanche starts to load the module
#  and declare the hostname_* functions

# Determine my current directory

set currentDir [file dirname [file join [pwd] [info script]]]

# Load the file containing the class definition

source [file join $currentDir hostname.tcl]

# Will get called each time we want to add a plugin. In this case, we
are
# only adding one

proc hostname_init { args } {
    array set options $args
    set hostnameInstance [hostnamePlugIn ::#auto]

    # Hook up the plugin to the namespace

    $hostnamePlugIn init -namespace $options(-namespace)
}
```

```
# This function is used to provide information about the installed
plugins

proc hostname_info {} {
    array set info {description {Example module that changes
hostname}}
    array set info {name {hostname}}
    array set info {version {1.0}}
    array set info {icon network}
    return [array get info]
}
```

Where do you go from here? Have a look at the other documents at the docs/ subdirectory and at the source code for the modules at plugins/. Writing a XML definition to support an apache module (such as PHP) is really easy. have a look at plugins/apache/modules