**University of Alberta**

# A Controlled Accessibility Scripting Environment for Web Applications

**By**

**Sunil Jeevananda Kamath** ©

A thesis submitted to the Faculty of Graduate studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta

Spring 2000

0-612-60132-3

**University of Alberta**

**Library Release Form**

**Name of Author:** Sunil J Kamath

**Title of Thesis:** A Controlled Accessibility Scripting Environment for Web Applications

**Degree:** Master of Science

**Year this Degree granted:** 2000

Sunil Kamath

Apt A, 9574 -87 Street
Edmonton, Alberta
Canada T6C 3J1

**Date:** **<u>Dec 23, 1999</u>**

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Controlled Accessibility Scripting Environment for Web Applications**, in partial fulfillment of the requirements for the degree of **Master of Science**.

P. Gburzynski

Dr. Pawel Gburzynski
Supervisor

[illegible]

Dr. Ioanis Nikolaidis
Co-Supervisor

[illegible]

Dr. Janelle Harms

[illegible]

Dr. Horacio J Marquez

**Date:** Dec. 22, 1999

# Abstract

Over the years the astounding growth of Internet and Intranet technologies have inspired many industries to turn their corporate database applications into online systems. Typically in any Web database application design, the database server and the application-specific components (web pages, navigation, etc.) are tightly coupled. This approach pinches the corporate budgets especially when we have to build a new application each time from scratch. Therefore in this thesis project we build a generic Web database application server in a scripting environment using Tcl/Tk that not only provides specialized services and functions, but also can cater to a wide application domains facilitating rapid application development. Our design focus targeted dynamic page/content generation, specification of user accessibility rights, easy to maintain, portable across various platforms and more importantly scalable. The tool that we have built will now allow novice programmers to build Web applications rapidly by coding just the application-specific components and gluing it to our server module.

# Table of Contents

## Appendices

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview

The explosive growths of Internet and Intranet technologies have inspired many industries to turn their database applications into on-line systems. As this industry matures, it will become more important, and more difficult, for Web developers to meet the growing challenges that will allow them to remain competitive in Web application development.

Some of the critical challenges facing Web developers include:

- Time
- Budget
- Specialized database features and quality
- Finding and keeping qualified people
- Ongoing maintenance, enhancements, and changes
- Building generic web servers

These factors are, of course, inter-dependent. As a general rule, the longer it takes to build a certain Web application, the greater the budget required. Likewise, as the requirements for the application become more sophisticated, more talented (and expensive) staff, as well as more development time, are required. In this volatile industry, staff turnover can contribute to increased development costs, when retraining staff is required. Cost is also related to the enhancements the application requires, and the specific process involved.

The developers are now also expected to include specialized features such as the following in their web server:

- dynamically-generated Web pages and contents
- ability to easily specify the user accessibility rights and classes of users
- Web site organization and intelligent navigation
- easy-to-construct, intelligent forms for data input
- ease of maintenance and updating
- scalable architecture
- low bandwidth requirement

The above requirements and issues facing the current web developers demand us to design a generic, specialized web database application server that not only provides these specialized and sophisticated services, but also facilitates rapid application development, and cater to a wide application areas. We now discuss the various possible ideas in trying to design a generic web application server and also present the limitations of each of them. We introduce the concept of scripting and explain how scripting, especially with Tcl [1] allows us to overcome several of the above issues and provide specialized database services to the users.

## 1.2 Existing solutions and their limitations

Today, building Web database applications with interesting and sophisticated features requires one of several common approaches, or a combination of approaches, each of which has its unique limitations:

1. Limitations of cobbling together scripts using the CGI (Common Gateway Interface) protocol.

- Difficult to write and maintain large programs using the CGI.
- Difficult to coordinate and maintain libraries of CGI scripts.
- Usually results in performance degradation due to the launching of multiple processes, each of which taxes the server's resources.
- Difficult to hand over maintenance to new or replacement programmers.

2. Limitations of using programming languages such as C++ or Java.

- Requires the skills of a programmer with computer science training
- Bringing higher-skilled programmers into the picture significantly increases development and maintenance cost
- Requires a break from the HTML page mode in which the Web application is grounded
- Greater development time since the code must be recompiled to test changes, fix bugs, etc.

3. Limitations of specialized applications from component vendors.

- Since an application or tool may provide only one or two of the desired components, Web developers must work with tools from multiple vendors.

- Software from different vendors may be difficult to coordinate and maintain, requiring additional time and effort on the part of the Web application developer.

- Forces the Web developer to invest in one or more vendors' proprietary technologies. This may place too many restrictions on the development process, limit the desired outcome, and increase staff-training requirements.

- Requires a dependency upon the third party vendor for features, support, bug fixes, etc.

- May require learning a proprietary scripting language or programming in C++, Java

- May entail a large performance overhead or memory "footprint". These applications typically generate CGI code "behind the scenes," creating further dependence upon the CGI method. Developers who want to access and control the source code that comprises their Web sites may find it difficult or impossible to do so.

## 1.3 Introduction to scripting

Over the last few years, there has been a fundamental change in the way people write computer programs, i.e., from system programming languages like C, C++ to scripting languages such as Javascript, Perl, Tcl, etc [8]. This change however does not mean that programmers have stopped using system-programming languages. It only means that in recent years, a new style of application development based on components has become more and more popular. In this style programmers don't start from scratch; instead, they build new applications by reusing existing components and applications. Programmers write just enough new code to connect the components and provide additional facilities that aren't already available from the components. The component-based approach allows existing code to be reused and thereby provides much faster application development.

A scripting language is not a replacement for a system programming language or vice versa [9]. Each is suited to a different set of tasks. For gluing and system integration, applications can be developed 5-10x faster with a scripting language; system programming languages will require large amounts of boilerplate and conversion code to connect the pieces, whereas this can be done directly with a scripting language [8]. For complex algorithms and data structures, the strong typing of a system programming language makes programs easier to manage. Where execution speed is key, a system programming language can often run 10-20x faster than a scripting language because it makes fewer run-time checks.

Thus scripting languages are highly useful in the following scenarios when the application:

- connects together pre-existing components
- manipulates a variety of different kinds of things
- includes a graphical user interface
- does a lot of string manipulation
- functions evolve rapidly over time
- need to be extensible

Today's online database applications demand more from web technology than C-based CGI programming [12]. The scripting languages such as Perl, Tcl/Tk, PHP3 [13], and ASP's [14] have thus gained a lot of popularity over the years. PHP3 and ASP's are server-side HTML-embedded scripting languages. This means that instead of writing a program with lots of commands to output HTML, you write an HTML script with some embedded code to do something (i.e., generate dynamic contents). This is perhaps different from a CGI script written in other languages like Perl, Tcl or C, which would contain lots of commands to output the HTML code. The PHP or ASP code will be enclosed in a special start and end tags that allows to jump into and out of the PHP or ASP mode. These systems also differ from the client-side scripting languages like Javascript in which the code that generates the HTML will be executed at the client-side. However, in server-side scripting languages like PHP3 and ASP, the code that produces HTML contents will be executed at the server. Thus the clients would receive the results of running that script and would have no way of determining what the underlying code may be.

In our server we adopt the server-side scripting model to generate HTML contents. However our system differs from the PHP3 and ASP systems, specifically with respect to not having to embed scripts within an HTML program. In other words, we have a straightforward script that would generate the entire HTML contents and therefore would not require switching between HTML mode and the PHP or ASP mode.

In the next section, we introduce a tool for scripting, Tcl/Tk and study the various merits of using this tool.

### 1.3.1 Advantages of Tcl

Tcl (Tool Command Language) [1] is a programming system developed by John Ousterhout at the University of California, Berkeley. This scripting language was primarily designed for gluing: they assume the existence of a set of powerful components and are intended primarily for connecting components together.

We have selected Tcl as the initial script language because it has several advantages that are important for our design:

- It is structured and English-like making it easy to learn and program as a next step for an HTML coder [2, 7, 12].

- It is an extensible language and facilitates Rapid Application Development environment [2].

- It is a "glue" code and therefore extremely suitable as a platform from which to launch code written in Java or other languages [2, 7].

- It has the constructs of a programming language including the ability to work with subroutines, conditional elements, variables and complex structures [2].

- It is easier to maintain and update code [2, 7].

- It enjoys wide industry support [2, 12].

- It is portable across various platforms [2].

- It allows the use of an already existing tool SICLE [4], for developing and running reactive scripts (see chapter 2).

One may argue that many of the above features are also provided in other scripting tools like Perl. However we choose Tcl over Perl mainly because Tcl is easy to learn and program for a beginner [11] and also in order to use SICLE (implemented in Tcl) that acts as a back-end for running our server. We also see that in the bargain we inherit the various other advantages of Tcl. Because of these advantages we can offer developers significant reductions in development cost and time by employing Web developers who may not necessarily have computer science backgrounds. At the same time, they can enjoy the technical benefits of software developed by more expensive programmers. According to Dr. John Ousterhout, designer of Tcl, "...many people have reported tenfold reductions in code size and development time when they switched from other toolkits to Tcl..." [8].

## 1.4 Research scope

In this section we try to clearly specify our research scope and define our key objectives. The thesis project primarily aims at designing and developing a generic web application server, based on a scripting environment using Tcl/Tk, that can easily cater to a wide application areas. Specifically, we are looking at applications requiring specialized database services such as facilitating different user classes in the system, the ability to clearly define the accessibility rules, generating dynamic contents to the users based on different conditions, and allowing image uploading. The reason for making it generic is to facilitate the re-use of the server to suit many different application areas. The idea is to make the server an application-independent module and whenever we need to develop a new application, it would only require coding the application dependent part and glue it to the server module that provides some common services.

Note that in addition to the above features, we also need our server to enjoy various properties to be easily maintainable, extensible, scalable, portable, and operate under low bandwidths. By choosing Tcl/Tk as our scripting environment, we are able to achieve most of these desired properties and by not having any frames-support we try to cater to low-bandwidth clients. However this does not completely mean that the system is designed to work under low bandwidths. This only results in less transferred volume and most importantly in no particular demands from the client. Therefore the end result of this project is to come up with a tool that would allow novice programmers to create and build web applications rapidly while providing sophisticated services to his/her clients.

## 1.5 Report Organization

This thesis report is organized into three key chapters. Chapter 2 explains the design of our generic web server. Details regarding the implementation of our server are provided in Chapter 3. This chapter discusses the operation of all the server components, and the driver program used as an interface between the user and the application server. Chapter 4 introduces a sample application, Blood Database system, and presents our implementation for this specialized system using our tool. This application is used as a media to demonstrate the effectiveness, robustness, and various other promises that our tool provides. Finally, in Chapter 5 we discuss our conclusions.

# Chapter 2

# Application Server Design

## 2.1 Overview

In this chapter we present the design of our generic server. We also present the details regarding the need for session rules, session templates and form templates. The chapter is organized as follows. The following section 2.2 presents the basic functionality of the application server. This section clearly distinguishes between the application dependent and application independent components. Section 2.3 explains the organization of the various components of our package. In section 2.4, we present a brief discussion on the operation of the application server. This section also explains the functionality of the CGI driver program. Section 2.5 presents the attribute naming rules and the associated types that the application server supports. In section 2.6, we introduce the concept of session linkage. Then in section 2.7 we briefly run through the session processing. In section 2.8 we explain our design of the attribute protection mechanism. We then discuss the processing of form templates in section 2.9. Section 2.10 explains the session description. Here we define the role and the functionality of the session rules and session templates. The chapter concludes with section 2.11 where we present a brief summary.

## 2.2 Basic Functionality

The main goal of this thesis project is to design a generic web application server that is simple to use, maintenance-free, low-bandwidth, and yet provides specialized database services over the Internet. By generic, we mean that the application server should easily cater to wide application areas that need some specialized database services over the Internet. These services come in-built into the server, and whenever we have to develop a new application, it will require for the developer to only code the application-dependent part, facilitating rapid application development.

Accordingly, our package consists of two components: the generic server module and the application dependent modules.

The application server is inherently an independent component and is built using Tcl and SICLE. SICLE [4] is a software package that is built using Tcl for developing and running reactive scripts. It can also be used to implement lightweight Web servers. In our design model SICLE implements the backend for running our server. It also provides DES encryption/decryption [15]. The application server assumes that the database transactions are most commonly a series of related actions, and therefore implement sessions understood as sequences of related web transactions. A session is executed within the context of a given user. To initiate a session, the user must log on to the system. The session will be terminated when the user logs off. As users have a natural tendency to abandon web sessions, these sessions can timeout. However, at any instant of time the server allows only one active session per user.

The application server supports different classes of users in the system. These different classes are assumed to operate in different environments, in the sense that they may see different forms (possibly from the same page template) during their sessions. The application server also provides the ability to link to different classes and view their private attributes. In addition, it has the ability to impose restrictions (read-only) on certain attributes depending on the confidentiality requirements of the application. These specifications should be definable and used accordingly by the application.

The application-dependent modules are constructed by having the notion of rule templates, session templates and form templates. The server requires each class of users to have its own rule and session template. The rule template defines the set of rules that are applicable to a particular session class (without their implementation) and the session template specifies the class linkage and attribute properties. The implementation of all the rules defined in the rule templates of each user class will be collectively provided in a separate rules program. The form template represents a straightforward Tcl program that the server executes to produce the required HTML form. Most of these HTML forms are generated dynamically, based on the dynamic conditions in the form template. The server interprets and executes these form templates differently (depending on user accessibility rights) for each user class.

## 2.3 Organization

In this section we explain the different components that compose a complete application. The design of our tool splits the entire package into two different components: Web Server and Application Server. The Web Server is a part that is accessible to the entire world and the Application Server is the actual application that services client requests. These two components communicate with each other by means of a CGI driver script.

The CGI driver is a straightforward Tcl script that, in contrast to the server script, doesn't use SICLE. Its role is to transform a form submission (or a special URL reference) into a request passed to the server. In most cases, such a request is sent to the server over a stream socket. There are situations, however, when the driver script must reference the application server component directly. One such situation involves an image upload transaction. As images can be potentially big, the driver does not pass them to the server over a socket, but stores them temporarily in the application server.

Owing to the fact that the driver script must have access to the application server component, yet the web server (which usually and rightfully executes with very limited access rights) invokes it, the script is not invoked directly, but rather called from a wrapper. The role of the wrapper is to make sure that the CGI script executes with the privilege of the application server's owner, i.e., it is allowed to read and write to the application server component.

The driver script is also responsible for simple load balancing, in the situation where there are multiple copies of the application server. Such multiple copies are visible via different sockets, possibly opened on different machines. Essentially, clients are assigned to servers based on their IP addresses. This assignment is described by a set of patterns (regular expressions) in the driver script. The issue is complicated by the fact that an active client may change its IP address in the middle of a session. Since the load balancing is done solely on IP-address patterns, this approach does not guarantee optimum load at the server. The technique is followed only to divert specific groups of clients to each server to make use of different servers available to service the user requests.

### 2.3.1 Application server components

The application server component comprises of application-specific and application-independent components as shown in figure 2-1. When we say application-independent component, we are referring to the generic application server (application server script) with specialized features that can be used for different similar applications. However with respect to a particular application in hand, this application server may be called as an application-specific component.

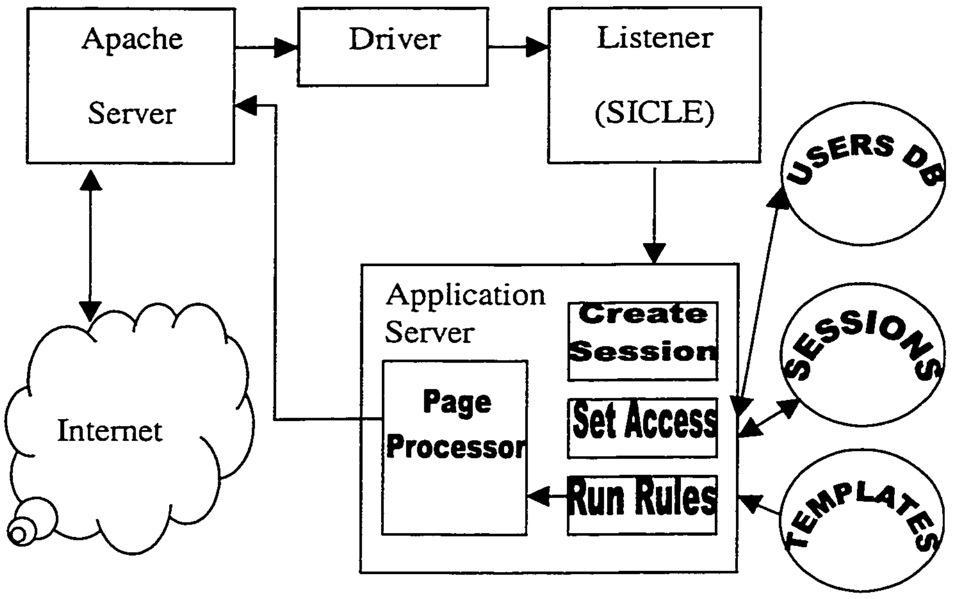

**Figure 2-1: Server architecture**

In the previous sections we saw that the application-specific components are constructed by having a notion of session template, rules template, and forms template. Therefore they all represent the components of the application server. We also said that there would be a session template and rules template defined for each class of user in the system. All these templates are organized in the application server component as **TEMPLATES**. The session template lists the session attributes, their default values and accessibility. The rules template that is defined for each class of user specifies just their names in the order in which they should be applied to a session belonging to the given user class. The method or implementations of the rules that are defined for all the different classes of users are collectively provided in a separate rules program of the **TEMPLATES** component. Thus the **TEMPLATES** component comprises of session and rules template for each user-class, and a single rules program that contains the implementation of all the rules defined in the rules template for each user class.

Another component of the application server is the **USERS** database. This can be viewed as the proper database and holds application-specific attributes. Each user known to the database has an entry there. Each user of the database will also have two sub-components: **authenticity** storing the user class and a DES-encrypted password with salt [4], and **attributes** specifying the contents of the user's record in the database. The salt is basically a 2-character string (12 bits) chosen from the set [a-zA-Z0-9./]. The salt string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string.

The application server also has a component called **SESSIONS** that stores some information about active sessions and is primarily used for locking. Every active user, i.e., currently engaged in a session, has an entry in SESSION whose name exactly matches the user name. It is illegal to start a new session for an already active user, but a user can re-login into his/her active session. The SESSIONS component identifies the specific copy of the server handling the session. A user re-logging into an existing session will be assigned to that server, even if based on its (new) IP address the session should be assigned to another server.

Any images uploaded by the user during a session are first put into the SESSIONS component. They will be moved into the corresponding database (USERS) when the session is eventually committed at the end. In our tool, a session must be committed (properly completed) to be considered valid. Otherwise, for example, if the session is abandoned or intentionally uncommitted, the user's record in the database, possibly including uploaded images, will not be modified.

The last component of the application server is the **form templates**. They are straightforward Tcl programs that will be executed by the application server (page processor) to dynamically generate the HTML code to the client. We also have **form inserts** that can be viewed as reusable (common) fragments of some form templates. Further the application server also uses special forms that shouldn't be treated as regular forms of the session. Two such forms that the application server supports are used to display error messages, and the other to present the contents of a selected textarea in a separate window.

## 2.4 Operation

In our server model, almost all the web pages (except the login form) are presented to the user as a part of the active session of the web application server. The user initiates each transaction by either clicking submit buttons on each form or by referencing a special HTML link, which invokes the CGI driver script on the web server's side. Although it is imperative that forms may also contain straightforward links e.g., pointing to additional information, clarification, www links, etc., the only transactions relevant from the viewpoint of our application server are those that pass through the CGI driver script.

Every form submission belongs to an active session and will trigger a specific action by the application server. Therefore any form belonging to a user session can be submitted anytime or perhaps multiple times during the course of the active user session and yet expect the same action by the server. Although the application will require a specific (possibly dynamic) ordering of forms, the present version of the rule set is intentionally very liberal in this respect to allow the usage of "Back" button of the browser and resubmitting it without a problem. Of course, it is also possible at any time to explicitly reference a specific form and submit it via a transaction involving the web application server.

It is also possible (and natural) to open several active sessions for different users at the same time and from the same browser. Such sessions are most naturally handled using multiple browser windows, but (with the assistance of "Back" and "Forward" buttons) they can be carried out from a single window. According to our rule, every form presented to the user belongs to exactly one session. Therefore every form submission contains a corresponding session identifier which enables the server to identify the session and understand the submission. It should be noted that the application server would only be able to recognize the session as long as it is still active (section 3.4).

### 2.4.1 The CGI driver script

The main function of the driver script is to transform a form submission (or a special URL reference) into a request passed to the server. It is a straightforward Tcl script [5], and provides the only link between the web server and the application server. It passes the user request to the application server over a stream socket. This driver program can be invoked in one of two possible ways.

1. In response to a form submission. In this case, the script is called without arguments; its standard input will contain the submission information.

2. In response to a (special) link request pointing to the CGI script. In this case, the script is called with arguments describing the parameters of the link request.

If the driver script has been called without arguments as in case 1, it reads the standard input converting the submission data into a list of pairs: *<attribute, value>*, where *attribute* is the name of the corresponding form field, and *value* is the submitted value of that field. The result of this conversion is independent of the encoding format, which is recognized based on the first line of the standard input.

The list of pairs sent to the server is preceded by two values, which in a sense arrive as a pair of its own. These two values are the *session tag* and the *IP address* of the client. The session tag is a unique eight-character identifier that the application server assigns to uniquely distinguish among different sessions. This value of the session tag is for the duration of the session, i.e., until the user logs off the application server. The entire submission is combined into a string that looks like a Tcl list whose elements are pairs. The length of this string is encoded into a four-byte integer and prepended at the front. This way, having read the first four bytes the application server knows exactly how many more bytes to expect.

On the other hand if the driver program is called with arguments as in case 2, the script recognizes that it has been invoked to process a link. It then transforms the request into a dummy form submission with the specified session tag and one attribute **Display** whose value specifies the linkage parameter. Thus special links are processed in the same way as form submissions.

When the driver script processes the form submission or special linkage requests, it tries to determine the session to which the submission belongs. If the form is a login form, the session tag attribute is **login**, which can never be a legitimate session identifier. So having examined this tag attribute, the driver program understands whether the submission is a request to start a new session (if tag is login), or whether it belongs to one of the existing active sessions.

The operation of initiating a new session is not as simple as it may look at the beginning, because the login request may be a re-login attempt into an interrupted existing active session. Therefore the driver script checks in the SESSIONS components to see whether an entry exists for the particular username. Note that all active sessions will be listed in the SESSIONS component with the username for each active session (section 2.3.1). If this is the case, the script tries to identify the specific copy of the application server handling the session and relays the submission information to that application server.

If the driver script passes through all the above tests, the submission is understood as a new-login request. The driver script then determines which application server should handle this new session. The application server chosen will then create a session and generate a unique session ID for that session that will be used to tag all subsequent forms presented to the user.

## 2.5 Attribute types and their naming rules

In this section we see the various attribute types that the application server supports and also explain their naming rules. All properties of a session attribute are determined by its name. This approach is both simple and convenient in a scripting environment where variable names are naturally available during program execution.

All the database attribute names should start with a lower case letter. Such a name directly corresponds to the name of a field in the user's database record. When a session is started, the user's database record is read (file **attributes**) and all fields found there are turned into session attributes with the same names. When the session is closed (and committed), the current values of the database attributes are written back into the user's record.

Names of database attributes are also naturally used to identify form fields. Typically, the purpose of a form is to assign new values to some fields of the user's record. When such a form is submitted, the list of pairs <attribute, value> received by the server directly reflects the current contents of the corresponding database fields. Image submissions are treated a bit differently (see below).

If an attribute starts with a upper-case letter, the server treats this attribute as a temporary attribute, i.e., one that is needed during the session, but whose value should not be stored in the database. Such an attribute can also correspond to a form field, i.e., it can be set automatically with a form submission, but it cannot represent a database field. Two names are reserved: *Display* and *Status*; they cannot be used as attributes settable from forms.

Any database attribute whose name includes the string "_img" is an image attribute, i.e., it represents an image or a list of images. The value of such an attribute (as stored in the session data structure or in the database record) is a Tcl list of pairs: <label, filename>, where *label* is the image label, i.e., a descriptive piece of string, and *filename* is the name of the file (usually stored in the user's directory in USERS) containing the actual image. With this approach, a single image attribute represents a whole class of images whose population is not explicitly limited. Different images on the image list of a single image attribute must have different labels.

Any attribute whose name includes the string "_lbl" is an image label attribute. It represents an identifier assigned to an image. Label attributes are not stored directly in the database (although their names look like names of database attributes), but they are used for image submission - to identify labels of submitted images.

Other attributes of a similar purpose are those names that include the string "_imd". They are used to pass an image deletion request to the server.

Finally, the last type of attributes the server supports are the ones that include the string "_mlt". The server interprets these attributes as multi-valued and adds all the values to the list representing the actual submitted value of the attribute. Declaring an attribute as multi-valued notifies the server that a single form submission may include several pairs of <attribute, value> with the same multi-valued attribute name and different values. This is done by the server rather than the driver script, i.e., the driver script does not interpret multi-valued attributes.

## 2.6 Session linkage

A user who has logged into the application server enters a session whose attributes (their initial values) are read from the user's database record. The default values of those attributes for a new user are determined by the session template for the given user class (section 2.10.1, 3.8.1).

In some cases, a user would like to open a record of another user and possibly modify the contents of the record. The kind of accessibility needed for this operation is implemented in our server through the concept of *session linkage*. According to the specification in session template (section 2.10.1, 3.8.1), it may be legal for a user of one class to start a session of another class and link to that session. For as long as user A is linked to user B, he/she effectively becomes user B and sees the same forms and operations as B would. In other words, the rules of A are overwritten by the rules of B. Later, when user A closes user B's session, the server would again reset the rules that are applicable to User A class. Through the design of our server model, it is possible to specify that the linked user has different access rights to some attributes of the linked-to session than the original user. Moreover, our tool allows altering the layout of forms, depending on whether they are presented in a linked-to or straightforward session.

It should be noted that the session linkage is not symmetric, i.e., if two sessions are linked, one of them is the *master* session (the one that requested the linkage) and the other is a *slave* (the one that has been created by the master session). In principle, it is possible for a linked-to (i.e., slave) session to link to (i.e., become a master of) another session, and so on. However it should be noted that if session A is a master of a slave session B, and session B is a master of another slave session C, then it is not possible for the slave session C to become a master of session A. In our design we do not allow the two active sessions for the same user at the same time. Hence when a user is having an active session, no other user is allowed to open a session for that user. This rule is implemented to overcome probable deadlock situations.

## 2.7 Session processing

In this section we explain the different phases involved in completing a submission cycle starting from the point when the application server receives the form submission from the driver script and ending when the application server send its response back to the client. The figures 2-2, 2-3, 2-4, and 2-5 shows the different phases and their actions involved in each submission processing cycle.

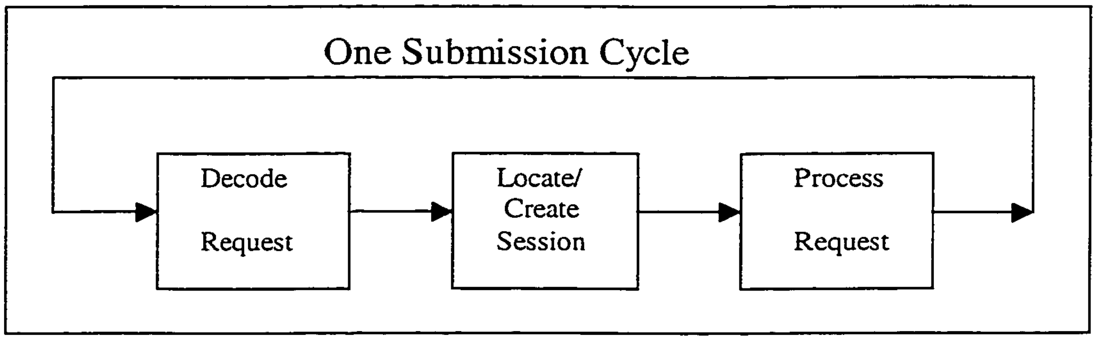

**Figure 2-2: Session processing cycle**

Initially we saw that the driver script combines the entire form submission into string and prepends two values: session tag, and IP address. We also said that a four-byte integer representing the length of this string would also be prepended at the front. So the application server first enters the decode phase in which it extracts the first four bytes and reads the contents into the Buffer. It also extracts the tag value to determine whether it is a login form or a submission into an existing session.

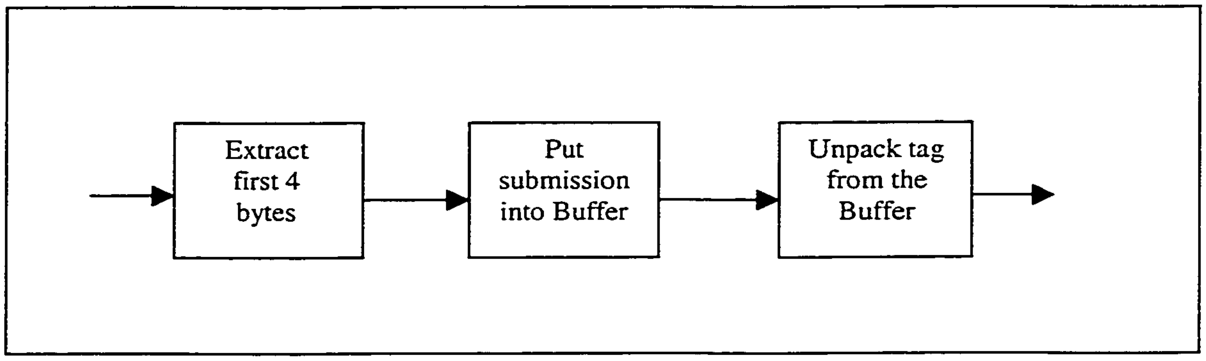

**Figure 2-3: Decode phase**

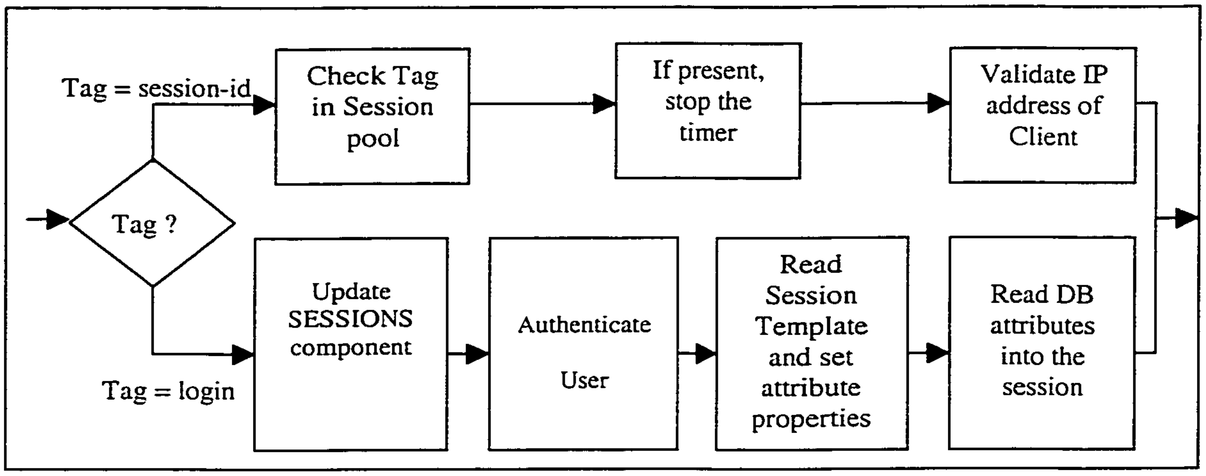
Tag = session-id
Check Tag in Session pool
If present, stop the timer
Validate IP address of Client
Tag ?
Tag = login
Update SESSIONS component
Authenticate User
Read Session Template and set attribute properties
Read DB attributes into the session

**Figure 2-4: Locate/Create Session**

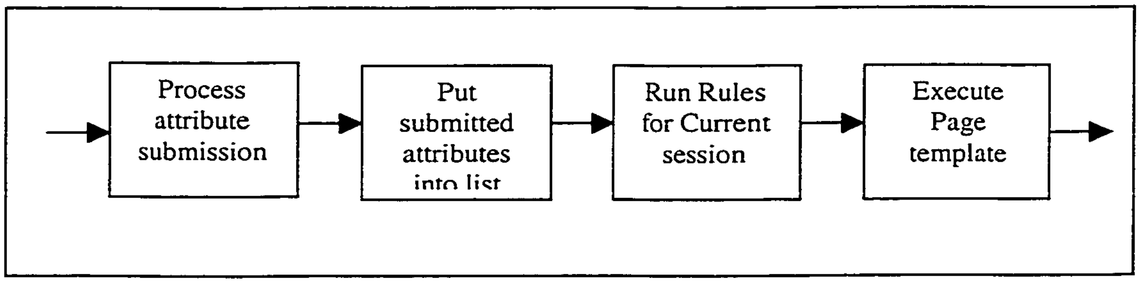
Process attribute submission
Put submitted attributes into list
Run Rules for Current session
Execute Page template

**Figure 2-5: Process Request**

So accordingly, in the next phase the application server tries to either locate an existing session or create a new session for the user (figure 2-4). It is during this phase that the application server creates a session and assigns a unique eight-character code to it for future identification. The application server then processes the request and runs the rules that are applicable for the present session class using the TEMPLATES component. When the rules return, it will determine the response page template that should be sent to the client. It then executes that page template to dynamically generate the HTML contents to the client. Also note that the rules are responsible for determining whether the session should be closed or committed and accordingly sets the response page to a corresponding form template. The cycle continues until the rules determine if the session should be closed or not.

## 2.8 Attribute protection mechanism

In applications requiring different classes of users and user accessibility rights for each of the attributes, there is a need for the server to provide an easy and consistent scheme that defines the accessibility rules at an attribute level according to the privacy and confidentiality requirements of the organization. Specifically, we require specifications when a certain user class tries to access information or attributes belonging to another class or perhaps to its own class. This is because, certain attributes may be deemed inaccessible for another class of user or to its own class. In our design of the server, we allow a high-level specification of user accessibility rights and they are defined for each attribute in the session template.

## 2.8.1 Attribute protection specification

In our previous sections, we mentioned that a session template is maintained for each user class. This session template is used to specify the attribute protection mechanism as required by the application.

The design of our tool currently facilitates three classes of users in the system and allows specification of each of the attributes to five probable user attribute types (A, B, C, D and E). Each of the class type has a predefined meaning associated with it. The meaning of each of these five user attribute types in the context of accessing his/her own attributes or linking to and accessing attributes of the other classes is illustrated in the following figure 2-6.

|   | Owner | Class2 | Class3 |
|---|---|---|---|
| A | no | full | no |
| B | full | full | full |
| C | full | full | no |
| D | no | full | full |
| E | no | no | full |

no: no access

full: full access

**Figure 2-6: Accessibility Matrix**

If an attribute belongs to type A, then the owner of the attribute cannot access the data. But when linked, class 2 has full access and class 3 no access. For example, if an attribute should not be accessible by its owner but should be accessible by a certain user class, Class2, and not by user class, class3, then the attribute will belong to type A specification according to the above table.

To demonstrate this example, let us consider that we have three user classes, APPLICANT, COMMITTEE, FACULTY, representing Class1, Class2 and Class3 users. Also, let us assume that there is an applicant-class attribute called *Status*, representing the status of the applicant. If this needs to be protected from tampering by the APPLICANT and only the COMMITTEE-class user should be able to have full access over this *Status* attribute, then the appropriate classification for these types of attribute would be type A.

If an attribute belongs to type B, then the owner of the attribute can access the data. But when linked both class 2 and class 3 users have full access.

If an attribute belongs to type C, then the owner of the attribute can access the data. But when linked class 2 has full access and class 3 no access.

If an attribute belongs to type D, then the owner of the attribute cannot access the data. But when linked both class 2 and class 3 users have full access.

If an attribute belongs to type E, then the owner of the attribute cannot access the data. But when linked class 3 has full access and class 2 no access.

Lastly if no specification is explicitly done, it means that the attribute has no access to whichever user classes that are initialized at the top of the session template.

## 2.9 Form Templates

As mentioned in our earlier section, a form template is nothing but a straightforward Tcl program that gets executed by the server to produce plain HTML document. In other words, the Tcl program consists of a sequence of procedure calls with specified arguments that are used to generate various components of an HTML document. All these procedures that generate the HTML content are in the *server* script. The template also contains various dynamic conditions required for the application server to compute and generate differently for different classes of users. The dynamics of these pages are coded by using conditional statements on different session attributes. The reference manual provided in Appendix A deal with these various server procedures that are used to generate almost all of the HTML code. The manual also illustrates with an example how to construct a page template.

## 2.10 Session description

A complete description of the application-dependent module comes in three parts:

- Form templates
- Session templates
- Session rules

The first part has already been discussed.

### 2.10.1 Session template

In the previous sections, we briefly discussed that the server maintains a session template for each class of user defined in the system. We also saw that the server uses this template to set various properties of the attributes. Now, in this section we target this session template and explain the role and importance of each of the information present in this session template. The session template is the application-dependent part of our tool whose role is to provide the following information to the application server:

- Initial values of session attributes.
- Patterns for assessing the validity of submitted values of those session attributes for which this is possible and makes sense.
- Accessibility rights for session attributes.
- Linkage rules for the session.
- Session timeouts

If an attribute has a verification pattern associated with it, this pattern is matched to the value of the attribute before the session rules are executed. If the value is incorrect, it is nonetheless stored in the session attribute, but the specified error message is presented to the user.

### 2.10.2 Session rules

The session rules are also an application-dependent part of our tool. These session rules are described by the contents of one global file containing the complete set of rules for the entire application, and multiple files (one per every user class), specifying the subset and ordering of the rules applicable to the given class.

Session rules can be viewed as a collection of procedures executed in the prescribed order. The primary purpose of the set of rules is to code the functional behavior of the application and determine the next form to be presented to the user. In addition, they also keep track of the Session State and decide when the session should be committed and/or terminated.

In the next chapter, we discuss the organization and implementation of each of the components discussed in this chapter.

## 2.11 Summary

In this chapter we presented the design of our generic server. First, we discussed briefly about the basic functionality of our application server. We then explained the role and objective of the CGI driver script. We then presented the attribute protection mechanism and briefly dealt with the design of form templates, session templates and rules template. In the next chapter we will present the implementation of our design.

# Chapter 3

# Implementation

## 3.1 Overview

In this chapter we present the implementation of our generic server program and also present the implementation of application-specific components. The chapter is organized as follows. The following section 3.2 presents the practical organization of the various components of our package. Then in section 3.3, we explain the form submission processing, special link processing and session linkage processing. Section 3.4 describes the method that the application server follows to generate unique session ids. Then in section 3.5 we briefly explain the session processing cycle; from the moment when the session is created until the server sends the desired page to the user. Section 3.6 presents our implementation of the attribute protection mechanism proposed in chapter 2. We then discuss the processing of form templates in section 3.7. In section 3.8 we explain the implementation and method of execution of session rules and session templates. The chapter concludes with section 3.9, where we give a brief summary of the chapter.

## 3.2 Organization

In this section we explain the implementation of the organization of all the components that compose a complete application. The installation of the tool logically divides the entire package into two parts or directories: the part accessible by the web server (called *web directory*), and the part accessible by the application server (called the *application server directory*). Figure 3-1 shows the organization of the entire package. For security reasons, the contents of the application server directory should not be readable by the web directory. The application server directory can (and perhaps should) be located outside the directory hierarchy visible by the web server, but the CGI driver program must be able to access it. In our organization, for simplicity and convenience, the application server directory is in fact a subdirectory of the web directory, but has been made unreadable by the web server. We now explain the contents and files that would be present in the web directory and the server directory.

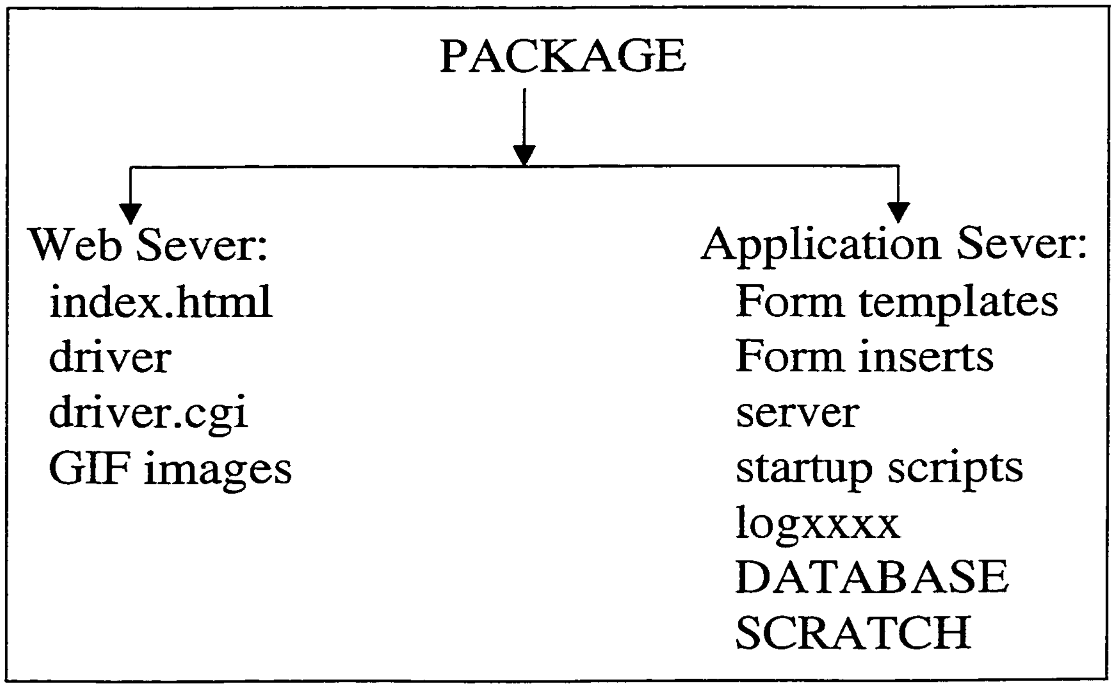

**Figure 3-1: Organization of Web server and application server**

### 3.2.1 Web files

The files that are accessible by the web server are the program *driver* and *driver.cgi*, HTML file *index.html*, and a number of gif images used by the forms. The two program files *driver* and *driver.cgi*, form the only interface between the web directory and the application server directory.

The web directory of the package determines the URL address of the application. The HTML file that will be returned when this address is referenced from a browser is of course *index.html*. This file is the banner page of the application: it contains the primary login form. This is the only HTML document that is presented directly, i.e., without being preprocessed by the application server. All the other documents sent to the user are generated by the application server based on form templates that are kept in the application server directory of the package.

The CGI program is a straightforward Tcl script (file *driver*), whose role is to transform a form submission (or a special URL reference) into a request passed to the server. In most cases, such a request is sent to the server over a stream socket. However, in chapter 2 we did see that there are situations, however, when the driver script must reference the application server directory directly regarding image upload transaction. This is implemented by having a SCRATCH subdirectory in the application server directory. The driver script initially stores the images directly (as files) in this SCRATCH subdirectory rather than passing it over the stream socket. Then the driver script notifies the application server about the image. The application server can then move the image from the SCRATCH directory to anywhere it pleases without performing actual data transfers, as long as the application server directory is entirely stored on a single file system.

Owing to the fact that the driver script must have access to the server directory, yet the web server (which usually and rightfully executes with very limited access rights) invokes it, the script is not invoked directly, but rather called from a wrapper. The role of the wrapper is to make sure that the CGI script executes with the privilege of the application server's owner, i.e., it is allowed to read and write to the application server directory. The wrapper is a very simple C program whose executable (named *driver.cgi*) is installed as SETUID. The SETUID function allows the driver script to access the server directory with the privileges to read and write into the application server directory. This is required since the driver script uses a temporary sub-directory (SCRATCH) in the application server directory to pass the image uploads to the application server rather than pass it over the stream socket.

### 3.2.2 Server directory

The server portion of the package occupies a single directory whose location need not be related to the location of the web portion, except for the requirement that the CGI program (driver) must be able to access it. This directory contains a collection of files and two subdirectories. The files found in the web server directory and the application server directory are of the following types:

We have already discussed the files of the web server directory in the previous section. Therefore we now deal with the contents of the application server directory alone. All files that end with the suffix **".htpl"** are form templates. They represent a Tcl program containing a set of procedure calls with arguments that must be executed by the server before the resultant form is presented to the client.

All files ending with the suffix **".ins"** are form inserts, i.e., they are *inserted* in the outgoing form templates by calls to special procedures. They can be viewed as reusable (common) fragments of some form templates.

All files ending with the suffix **".nfrm"** are special form templates that shouldn't be treated as regular forms of the session. Two such forms that the server supports are as follows: *error.nfrm* used to display an error message, and *gentxt.nfrm* used to present the contents of a selected textarea in a separate window.

The file **server** represents the application server that serves all the client requests. It is responsible for authenticating user login requests, creating and managing users, and processing form templates before presenting them to the clients. The server performs the above operations by using session and rule templates that will be dealt with in detail in subsequent sections.

The other files that would be present are the **startup scripts** for running the application server. The startup script is a SETUID wrapper invoking the application - to make sure that the application is executed with the proper owner.

Files named "**logxxxx**", where *xxxx* is a number, are log files generated by the servers. The number corresponds to the port number (socket) via which the server is visible on its host.

Apart from these files, the server directory has two subdirectories as mentioned earlier: **DATABASE**, containing the database files, session rules and session templates, and **SCRATCH** used to pass image files from the CGI script to the application server. The **DATABASE** directory has the structure as shown in figure 3-2.

The subdirectory **TEMPLATES** under the DATABASE directory is the only application dependent part of our package with respect to the application developer. It also comprises the static part of the database, i.e., the part that doesn't change as new users are added, or user records modified, and so on.

File **rules** contains the so-called session rules for the application at hand, which can be viewed as the application-specific part of the server program. Note that in a scripting (interpretative) environment, the server can easily read-in and execute a piece of text as part of its program.

```
DATABASE
    TEMPLATES
        rules
        CLASS1
            rules
            templates
        CLASS2
            rules
            templates
        CLASS3
            ..
        SUPERVISOR
            ..
    USERS
        ...
        username
            authenticity
            attributes
        ...
    SESSIONS
        ...
        username
            server
        ...
```

**Figure 3-2: Organization of database directory**

For each class of users, **TEMPLATES** contains a subdirectory, which is named as the corresponding user class. This directory contains exactly two files: **rules** and **templates**. The first file specifies the rules (just their names, as declared and defined in the global rules file from TEMPLATES) in the order in which they should be applied to a session belonging to the given user class. The second file contains the session template for this user class. It lists the session attributes, their default values and accessibility.

Directory **USERS** can be viewed as the proper database and holds application-specific attributes. Each user known to the database has a subdirectory there; the name of this subdirectory exactly matches the user name. This subdirectory contains at least two files: **authenticity** storing the user class and a DES-encrypted password with salt, and **attributes** specifying the contents of the user's record in the database. This record is simply a Tcl list of pairs: *<name, value>*, where *name* is an attribute name and *value* gives the value of the attribute. If the record includes images, those images appear as separate files in the user directory. The names of those files are referenced in the *attributes* file as values of the corresponding (image) attributes.

Directory **SESSIONS** stores some information about active sessions and is primarily used for locking. Every active user, i.e., currently engaged in a session, has a subdirectory in SESSION whose name exactly matches the user name. File **server** in the session directory identifies the specific copy of the server handling the session. A user re-logging into an existing session will be assigned to that server, even if based on its (new) IP address the session should be assigned to another server.

Any images uploaded by the user during a session are first put into the SESSIONS directory. They will be moved into the corresponding database directory (a subdirectory of USERS) when the session is eventually committed at the end. In our tool, a session must be committed (properly completed) to be considered valid. Otherwise, for example, if the session is abandoned or intentionally uncommitted, the user's record in the database, possibly including uploaded images, will not be modified.

## 3.3 Form submission processing

Most of the form templates used by the application server and represented by Tcl programs are sent to the user after executing (producing HTML code), with the following standard sequence included in them:

```
<form method="post" action="driver.cgi">
<input type="hidden" name="tag">
```

The first line indicates that when the form is submitted, the web server should invoke *driver.cgi* (i.e., the wrapper of the proper driver script). When this happens, the form information will appear on the standard input of the driver script.

The second line defines an invisible attribute of the form named *tag*. Every form submitted to the server must have this attribute. When the form template is turned into an actual form and sent to the client, the server assigns a value to the *tag*, i.e., it turns the above prototype input specification into:

```
<input type="hidden" name="tag" value="session-id">
```

where, *session-id* is a unique identifier assigned to the session by the server. The value of tag will be returned to the server (as part of the submission information) when the form is submitted. This way the server keeps track of multiple sessions.

Web browsers, including reasonably recent versions of Netscape and Internet Explorer, support at least two types of encoding for a form submission. The standard encoding type, assumed by default, is application/x-www-form-urlencoded (we will call it application encoding for short). A form that can be used to upload images must specify a different encoding type, which we will call multipart encoding. This is the only possible encoding type for an image submission. For such a form, the form method specification looks as follows:

```
<form method="post" action="driver.cgi" enctype="multipart/form-data">
```

Both submission formats are acceptable by the driver script. In fact, any form (not necessarily including an image submission) can be submitted with multipart encoding. However, it is recommended to avoid the multipart encoding for those forms that do not need it, as it is less efficient (longer) than the default (application) encoding. For details, the reader is referred to an HTML manual (version 3.0 or higher) [3].

### 3.3.1 Special links processing

Form templates may also include special link requests that must be processed by the application server. The format of a typical link request is as follows.

```
driver.cgi?tag&parameter&snumber
```

Where *tag* is the complete session tag, *parameter* identifies the link to the server, and *snumber* is a serial number of the link. The serial number is added by the application server to make sure that subsequent editions of the same link (e.g., in different versions of the form) appear different, so that client and proxy caching is effectively disabled.

For example,

<ahref="driver.cgi?11stauffer:8011CzEApmB6&Sel_comlogout&15226">

Looks like an anchor with a special link.

## 3.3.2 Session linkage processing

In our application server, the session linkages are implemented by following a special HTML link (section 3.3.1) specifying a linkage request. The linkage request is only successful if the indicated user exists and the session template for the requesting user declares this operation as legal (section 3.8.1.1).

The following is the format of the special URL.

driver.cgi?*tag*&LINK-*user*&snumber

where *user* is the name of the user of the slave (linked-to) session. The remaining components, i.e., *tag* and *snumber*, were discussed in the previous section. Note that the above parameters are sent as plain text to and from the server.

The next sub-section explains how the *tag* is determined and used by the server.

## 3.4 Session Identification

Until now we only said that there would be a session identifier for each session determined by the server. We now go a step forward and explain how the server computes and assigns a session Id for future identification.

When the driver script processes the form submission or special linkage requests, it tries to determine the session to which the submission belongs. If the form is a login form, the session tag attribute is **login**. So having examined this tag attribute, the driver program understands whether the submission is a request to start a new session (if tag is login), or whether it belongs to one of the existing active sessions.

The operation of initiating a new session is not as simple as it may look at the beginning, because the login request may be a re-login attempt into an interrupted existing active session. Therefore, the driver script checks whether the username attribute of the submitted form matches one of the directory names in SESSIONS. Note that all active sessions will be listed in the SESSIONS directory with a directory representing the username for each active session (section 2.3.1, 3.2.2). If this is the case, the script tries to open the server file in that directory, which contains a single line of text identifying the server (host name and port number) handling the session. The submission information is then relayed to that server, which is responsible for verifying its validity.

If the driver script passes through all the above tests, the submission is understood as a new-login request. The driver script now has to determine which server should handle this new session. This is achieved by matching the IP address of the client to the list of patterns (regular expressions) assigned to the servers when the package was installed. The first server whose pattern matches the IP address of the client is chosen to handle the new login request. The server will generate a session ID that will be used to tag all subsequent forms presented to the user. Note that such forms pass through the server, which can assign distinctive values to their tag attributes.

At first sight, it might seem that because the allocation of servers to clients is based deterministically on the client's IP address, the driver script doesn't have to check for a re-login attempt into an existing session. The driver script can simply pass such a request to the server (determined by the IP address of the client) and the server will check whether the session is already in progress. Unfortunately, this doesn't work if the IP address of the client has changed in the meantime, e.g., the IP address is dynamically assigned by the Internet Provider. It may happen that based on the (new) IP address the session should be handled by a given copy of the server, yet it must be continued on a different server, namely the one on which it was started. This is why the server to which a re-login attempt is passed is determined by the contents of the server file in the session directory.

The complete tag of a form that belongs to a definite session has the following structure:

nnhost:portxxxxxxxx

where, *nn* is the length of the host:port part in characters, *host* is the name of the host on which the server handling the session is running, *port* is the port number of the server, and *xxxxxxxx* is an eight-character unique session identifier generated by the server when the session was started. This eight-character code is generated using the following SICLE function.

rndinit *seed*

The above function generates a random number based on a specific integer seed. Therefore the method that is followed to determine the session-id is as follows:

rndinit [expr [[clock seconds] * [pid]]

**[clock seconds]** is a Tcl function that returns the current date and time as a system-dependent integer value. **[pid]** is a Tcl function that retrieves process-id.

All information is textual, i.e., the session tag can be safely embedded into an HTML document.

For illustration, this is a possible session tag:

11stauffer:8011CzEApmB6

Note that having received a submission with a session tag in the above format, the driver script knows immediately to which server this submission should be relayed (stauffer in the above case). The server itself will validate the session identifier (CzEApmB6 in the above case), locate the session, and perform the necessary processing. One should note that the security provided by these session-id's are weak and it is possible for malicious users to hijack existing sessions. This is because the session tag that the sever uses to identify an existing session is sent and received by the application server as a plain text information.

If the *Obsessive* parameter is set to 1, meaning that the client's IP address should be included in the session identifier, the server makes sure that the IP address of the client issuing the request matches the IP address of the session originator (stored as a special session attribute). Note that the client's IP address is sent by the driver script in the header of every submission.

## 3.5 Session processing cycle

In this section we explain the step-by-step process of the server starting from the point when the user first logs into the system and ending when the user session is terminated. We have already seen that all requests that ultimately arrive at the server can be viewed as form submissions, although sometimes those submissions are in fact special requests (e.g., special links or linkage requests) (section 2.4.1). The first submission in a session usually comes from an authentication form and is recognized by the server by the special name of the session tag (login). The attributes that arrive with such a submission are not stored as session attributes but they are used to locate the user and perform the required authentication. These attributes are *Username* and *Password.*

However, if the session tag arriving in the header of a submission indicates an existing session, the server tries to locate that session based on the 8-character session code (section 3.4). If the session exists, the server identifies the session object, stops the session inactivity timer, and processes the submission within the context of this session. If the session does not exist, which means that the session has been timed out or been closed normally, the server checks whether there exists an alive master session for the referenced session. This is because, it may happen that a slave session has timed out, but its master session is still active (section 2.6). In such a case, the user will be automatically switched to the master session. The current submission will be erased (i.e., it is assumed to be empty), as it cannot be considered valid in the context of the master session. A warning message will then be prepared and presented to the user to notify him/her what has happened. Note that there are two standard cases when the session rules receive an empty submission. The first case is immediately after login and the second case is when the master session is resumed after the termination of its slave. During login, the attributes/values that arrive at the server are used internally for authentication purposes only; the submission perceived by the rules is empty. The rules can tell the difference between these two situations by setting a flag (a temporary session attribute) after the first empty submission (received after login).

### 3.5.1 Attribute submission

Once the session has been identified, the server reads the submitted sequence of pairs <attribute, value> and performs the following operations:

- If attribute is "*Display*", it means that the entire submission is a special link request (section 2.4.1). Two types of such submissions are processed entirely by the server (i.e., without presenting them to the session rules), **LINK-** and **IMG-**.

If the value part starts with "LINK-", the special link is a session-linkage request. In such a case, the server checks the validity of the linkage request and, if everything is OK, creates a new slave session for the indicated user and links the present session to it. The current session is then set to the new slave session. The *Display* attribute is erased, so that the new (slave) session will start with an empty submission.

If value starts with "IMG-", the link represents an image fetch request, i.e., the user has clicked on an image link. In such a case, the server will identify the image by examining the remaining part of value and store its file name in the variable *OutgoingPage*.

Any other format of value indicates a special link that should be processed normally by the session rules.

The server treats *Display* as a temporary attribute (because it starts with an upper case letter), whose value will be available to the session rules (section 2.5). Note that *Display* always arrives as a single-attribute submission. The attribute is erased at the beginning of every submission cycle. If, as perceived by the session rules, the attribute has a value, it means that the current submission is a special-link submission.

- If the attribute is inaccessible in the current session, the submission is deemed illegal and diagnosed as such.

- If the attribute is an image attribute, image label attribute, or image deletion attribute, it is stored (together with its value) in a special list and ignored for now.

- If the attribute is a multi-valued attribute, it is stored (together with its value) in a special list and ignored for now.

- Any other attribute arriving with the submission is simply added (with its value) to the list of current session attributes. If the attribute already exists as a session attribute, its previous value is overwritten. The server checks whether the specified value of the attribute matches the value pattern associated with the attribute in the session template.

Having exhausted the submitted list of pairs, the server processes images and multi-valued attributes. Note that every multi-valued attribute arrives as a number of <attribute, value> pairs with the same first element (attribute name). The server combines all such pairs into a single <attribute, value> pair, where the *value* is now the list of different values of the attribute, and the *attribute* represents the multi-valued attribute name.

We are now left with the explanation of how the server processes the images. As discussed before, every image submission (upload) is described by two <attribute, value> pairs. The *attribute* element of the first pair identifies the image attribute to which the new image should be added. Note that the name of the attribute should include the string "_img" (section 2.5). The submitted value of the image attribute is the name of the file in directory SCRATCH containing the image (section 3.2.1). The second pair identifies the image label, i.e., a piece of text describing the image. Its attribute element should be the same as the attribute name of the first pair, with the string "_img" now replaced by _lbl".

For every image submission, the server matches the actual image attribute with its label. A submission missing a label or with an empty label is illegal. Then it looks up the image attribute in the session. The stored value of the session attribute is a list of pairs <label, filename>, i.e., a single image attribute may point to several images. Next the server determines the database file name for the new image (as opposed to the temporary name assigned by the CGI driver script). If the submitted image label matches one of the existing image labels associated with the session attribute, the new image will be stored using the file name of that existing image, i.e., the new image will overwrite the old one. Otherwise, the server selects a new unique name for the image file and adds a new <label, filename> pair to the session attribute. The submitted image file is moved to the session directory.

Note that the image is first moved to the session directory (which is temporary), rather than the user directory (representing the permanent database record of the user) (section 3.2.1). It will be moved to the user directory only when the session is completed and committed. While fetching images and presenting them to the user, the server first checks the session directory and then looks in the user directory. This way uncommitted images (i.e., those images that are not yet moved to the user directory) are presented correctly as current, also when they replace old permanent images. But they will not physically overwrite the old images until the session is committed. The same logic applies to regular (non-image) attributes whose current values are taken from the session object rather than the database record.

For processing image deletion requests, when the form is submitted, for every image marked for deletion the server receives an attribute whose name is the same as the name of the corresponding image attribute with the "_img" replaced by "_imd". The value of this attribute is the label of the selected image. Image deletion attributes, i.e., all attributes that include the string "_imd" in their names, are completely processed by the server (similar to image label attributes) and they never end up as actual session attributes. They are also invisible to session rules.

## 3.5.2 Session completion

In the previous section, we saw how the server processes the various attribute types. Once the server finishes processing these attribute submissions, it invokes the rules that are applicable for the current session (section 2.10.2, 3.8.2). One goal of this operation is to set *OutgoingPage* (a variable available to the rules) to indicate to the server what should be sent to the user in response to the last submission. *OutgoingPage* can be set to the file name of a form template (sought in the server directory) or to the file name of an image (sought first in the session and then in the user directory).

When the rules return, the server performs the following operations:

1. The session status is checked. If the session has been canceled (i.e., closed) by the rules and it is a slave session, the server de-allocates the session and unlinks, i.e., reverts to the master session. If the slave session has been committed by the rules, its attributes are written to the database (file attributes in the user directory) and its uploaded images (if any) are moved to the user directory. Then the server runs the rules again - this time for the master session, to determine the value of *OutgoingPage* - and re-executes step 1.

2. The server determines whether *OutgoingPage* points to a form template or to an image file. This is decided based on the first three characters of the file name. Image files have names starting with "img"; this prefix should never be used to name a form template file. If the file name stored in

*OutgoingPage* indicates a form template, i.e., it doesn't start with "img", the server checks if it terminates with a suffix, i.e., contains a period. If not, the standard suffix ".htpl" is appended to the file name. The server opens the file, reads the form, and sends it to the user executing it along the way. If the file contains an image, the server sends the image preceded by the appropriate "content-type" header. This resulting document is sent to the driver script (over the same socket on which the submission was received) which in turn relays the document to the user.

3. The socket connecting the server to the driver script is closed.

4. The server examines the session status. If the session has been canceled, the server de-allocates its object. If the session has been committed, its attributes, including uploaded images, are written to the user directory. Note that now the canceled session cannot be a slave session, because closed slave sessions are processed in step 1.

This completes one submission cycle. Note that if the session rules determine that a slave session should be closed, the new outgoing form will be generated by the rules of the master session. These rules may in turn decide that the master session should be closed as well, and if the session itself is a slave session of another master session, the rules of the new session will determine the value of *OutgoingPage*. But if the closed session has no master, it will present the last outgoing page to the user before being finally closed.

If a session being closed has active slave session, all those slaves are recursively closed as well.

If a session being closed has an attribute named *discard_on_close*, and the value of this attribute is not an empty string, the user record is removed from the database. This simple trick is used to discard users created accidentally (or for fun), whose initial sessions haven't been completed in a sensible way. Note that for any new session, the initial value of *discard_on_close* is yes. If the session rules decide that the record has reached a shape that will make it an acceptable entry in the database, they will unset the attribute. Otherwise, when the session is forcibly closed in an incomplete state, the user record will be erased.

At the end of a submission cycle, the value of *OutgoingPage* is stored in a temporary session attribute. These are used in those circumstances when the server has to re-send the form that was last presented to the user. However, if *OutgoingPage* points to an image rather than a form template, its value is not saved in the temporary attribute.

## 3.6 Attribute protection implementation

In our previous chapter, we mentioned that a session template is maintained for each user class (section 3.2.2, 2.10.1). This session template is responsible for defining the accessibility for users with respect to whether the user can access his/her own class attributes or attributes belonging to different classes. The first line of the session template usually is { OWN "" }. This declaration tells the server that the owner of the attributes can access or view the attributes that are listed in the session template. If the owner of this class can also link and probably view or access the information belonging to the other class, say class2, then we would also see a declaration like { class 2 "" } and likewise for class3 and so on.

Note that the above declaration has only initialized which user classes are able to link or access the attributes. We have yet to give a specification of attribute type for each of the attributes. Now in the same session template, a Tcl list is defined for each attribute consisting of five fields. The second field in this list is dedicated for specification of the user attribute type. We have already explained the different specifications that are allowed and their meanings in Chapter 2. However in this section we explain the implementation of our proposed technique.

The application server implements the attribute protection mechanism by having two list "pyes" and "pno". The "pyes" list store all the session attributes that are accessible under the current session and the "pno" list store all the session attributes that are not accessible. These lists are constructed when the server tries to build a session and are set by the ***setAccess*** procedure in the application server. When the server is in the process of building a user session, it reads the corresponding session template and constructs an empty list TAccess(class1, OWN) if there is a declaration of { OWN ""} in class1's session template, TAccess (Class2, Class1) if there is a declaration of { class2 "" } in the class1's session template and so on. The ***setAccess*** procedure then uses these TAccess lists to learn whether it is an access to its own attributes or an access from a different class. This is important for the ***setAccess*** procedure because attribute rights may change depending on whether it is accessed by its owner or from another defined class. The setAccess procedure then reads in each of attribute from the session template and its second field that specifies the attribute type and accordingly puts it into "pyes" or "pno" list depending on whether accessible or not. Therefore, whenever the server wants to determine whether the attribute is accessible under the current session, it has to check the pyes or pno list. These two lists are active for as long as the session is active.

## 3.7 Form Templates

As mentioned in our earlier section, a form template is nothing but a straightforward Tcl program that gets executed by the server to produce plain HTML document. In other words, the Tcl program consists of a sequence of procedure calls with specified arguments that are used to generate various components of an HTML document. All these procedures that generate the HTML content are in the *server* script.

In this section we start the discussion from the instance when the application server has already determined the page template that must be used to construct the outgoing HTML page. This is usually determined by the rules, which stores the page template to be used by the application server in the variable *OutgoingPage*. Therefore, having known the name of the template file, the application server tries to open it, reads the content of the files, and then executes the program. The server uses the procedures that generate HTML code and outputs the HTML contents directly to the driver program. However if the server cannot find the page template in the server directory, or it cannot open/read it, it initiates an appropriate session failure message to the user.

## 3.8 Session description

A complete description of the application-dependent module comes in three parts as discussed in chapter 2: Form templates, Session templates, and Session rules. We already discussed about form templates and continue our discussion with respect to session templates and session rules.

## 3.8.1 Session template

In the previous chapter we studied the role of a session template and its use in providing the following information.

- Initial values of session attributes.
- Patterns for assessing the validity of submitted values of those session attributes for which this is possible and makes sense.
- Accessibility rights for session attributes.
- Linkage rules for the session.
- Session timeouts

Usually each attribute in the session template will be of the following format.

{*attribute-name, access-right, default-value, verify-pattern, error-message*}

The above specification for an attribute is a Tcl list consisting of five fields: the attribute name, class type, its default value in the database, a regular expression indicating the attribute verification pattern, and an error message to be presented to the user in case the attribute value does not follow the regular expression pattern.

If an attribute has a verification pattern associated with it, this pattern is matched to the value of the attribute before the session rules are executed. If the value is incorrect, it is nonetheless stored in the session attribute, but the specified error message is presented to the user. This way we are able to present accurate warning messages to the user. The second field of the attribute list is a single character letter describing the access rights for different session users (section 2.8.1).

Note that a session attribute need only be specified in the template if at least one of the following happens to be the case:

- Its default value is different from the empty string
- A verification pattern must be specified for the attribute
- The attribute should be made accessible

Otherwise, the attribute will be added to the session when submitted for the first time. When referenced before then, the attribute's value will be an empty string. Also note that if the attribute does not appear in the session template, the server assumes that the attribute is not accessible to the current session.

### 3.8.1.1 Timeouts and accessibility rules

Recall that the names of a session attribute whose value is to be stored in the database cannot start with an upper case letter (section 2.5). Since only permanent attributes can be specified in the session template, entries starting with capitalized keywords are used for non-attribute specifications. Two such specifications are used in the session template: one for initializing the access classes and the other for specifying the timeouts.

The format for specifying the accessibility is as follows:

{*class* ""}

where, *class* is a user class identifier (or **OWN**). The specification refers to all session attributes whose names match the specified pattern.

If the *class* is not **OWN**, it must be a user class identifier (e.g., CLASS1, CLASS2). Such a specification initializes the accessibility of attributes of the indicated session class, when it becomes a slave of the present master class.

If the keyword **OWN** appears as the first item, the specification refers to the current session class, when carried out as a master session, i.e., by its OWN user.

Note that if the users of a given (master) session class are to be able to link to (slave) sessions of another class, the template of the master class must specify the initialization of the linkage to the slave session. For example, if the users of CLASS1 (master) can link and access attributes of CLASS2 (slave), the session template of CLASS1 should have the following specification: {CLASS2 ""}.

The second non-attribute specification in the session template is the timeouts. The format of this type of specification is as follows:

{TIMEOUT "xx" "yy"}

where, *xx* represents the time in minutes of session inactivity when carried out as a master session, and *yy* represents the timeout interval in minutes, if it is the slave session linked to by another user. If only one timeout value is specified, both timeouts are assumed to be the same. However, if no **TIMEOUT** specification is given, the session timeout (for both cases) is determined by the installation parameter **SessionTimeout**. In either case, after the timeout, the server will forcibly close the session.

## 3.8.2 Session rules

In the previous chapter we studied that session rules can be viewed as a collection of procedures executed in the prescribed order. The primary purpose of the set of rules is to code the functional behavior of the application and determine the next form to be presented to the user. In addition, they also keep track of the session state and decide when the session should be committed and/or terminated. In the next sub-section we study how rules are programmed and used by the application server

### 3.8.2.1 Rule execution environment

A rule is declared (in the global rules file) by invoking the following function:

rule *body*

where, *body* is the list of Tcl statements comprising the rule.

The rules files associated with each class of user (rules template) will only have just a list of rules (from the global file) to be applied in those sessions. The rules are called by using the following statement in the rules template:

**apply** *rule_name*

The order of the apply operations determines the order in which the indicated rules will be executed after every form submission within a session.

A rule has no arguments but it can access certain variables and execute certain functions. The variables directly accessible to every rule include the Session attributes as they appear after the last form submission. This includes the temporary attributes and also the special temporary attribute *Display*. It is recommended to use **sessionAttribute** function for reading the attribute values. To set an attribute within a rule, a rule can directly reference the Session handle.

For example,

```
        setattr $Session Selfile_mlt ""
```

sets the value of (temporary) attribute *Selfile_mlt* to an empty string.

At the end of execution of all the rules, it determines the next page to be presented to the user and accordingly assigns the page name to the variable *OutgoingPage*.

Note that although rules resemble (argument-less) functions, they in fact execute within the same local context. This means, for example, that when one rule sets the value of a (local) variable, this value will be available to the next rule (unless, of course, the variable is unset in the meantime). Therefore, there is no need (and no sense) to use global variables to pass state information among the rules. Also, it doesn't make sense to use global variables to store information across different submissions (i.e., web transactions) of the same session. This will not work because global variables may be overwritten by other sessions. Any information that must be saved across submission cycles must be stored in (temporary) session attributes.

If a rule executes **break** outside the scope of a construct for which break is meaningful, it breaks the execution of the rule chain. A rule can do this when it has determined the value of *OutgoingPage* and there is nothing more that the subsequent rules could possibly do.

## 3.9 Summary

In this chapter we first presented the organization of our tool and showed how the different files and components are organized within our package. We then explained how our application server processes form submission and special links/linkage requests. A brief description of the method for computing the session id was also provided. Finally we presented our implementation of attribute protection mechanism, execution of form templates and session descriptions (session and rules template).

# Chapter 4

# Blood Database System – A Sample Application

## 4.1 Overview

In this chapter we present a sample application to demonstrate the effectiveness of our tool. The application represents a complex database system incorporated by the American Red Cross to perform auditing on patient transfusion data [6]. At first, we briefly describe the operation and requirements of the system. We then provide the implementation of the system and demonstrate how our generic server facilitates rapid application development and successfully achieves the various promises.

The chapter is organized as follows. In section 4.2 we briefly outline the various functional and design expectations of the system. This section also explains the operation of the entire system. Then section 4.3 introduces the system users and also explains their functionality. Section 4.4 specifies the linkage between the different system users and finally section 4.5 describes the attribute protection as required when accessing information between these various classes of users. Section 4.6 presents an outline of our implementation of the application and the chapter concludes with section 4.7 where we give a brief summary.

## 4.2 System requirements

A Blood database system is a database management tool used to capture and analyze patient's transfusion records [6]. The American Red Cross hospital customers use this system as a tool for conducting transfusion audits. The blood database system stores patient transfusion data on the following blood products: red blood cells, whole blood, platelets, fresh frozen plasma, and cryoprecipitate.

The critical functional objectives of the system are to:

- Store patient transfusion records
- Check patient medical records against preset transfusion audit criteria
- Tally the number of transfusions associated with each indication
- Produce reports detailing the records that do not automatically meet the transfusion audit criteria
- Provide web enabling to perform the above operations via the Internet

In addition to the above functional requirements, the system also requires the following design attributes:

- Dynamic page generation
- High level specification of user rights
- Security
- Ease of Maintenance
- Extensibility
- Rapid application development
- Low bandwidth requirement

The above design attributes are common and important to many applications, both from the perspective of the client as well as the developer. We therefore demonstrate through this application how our server accomplishes the above objectives. We also prove the generic nature of our web server in an attempt to clearly differentiate between the application-dependent and application-independent components of our package.

However, before going into any such details, it is important for us to first discuss and explain the operation of the application and understand the various functions. The following sections of this chapter introduce the system users, attribute accessibility, and the linkage requirements between various system users.

### 4.2.1 Understanding the system

As mentioned earlier, the Blood Database system is a tool that stores patient transfusion data and helps audit these transfusion records. When the patient is admitted to the hospital for a transfusion, the designated doctor (also called physician) fills in the general patient information such as patient name, blood group, date of birth, etc. This doctor then decides the blood product required for the patient and fills in the blood product order form. The blood product could be one of the following: red blood cells, whole blood, platelets, fresh frozen plasma, or cryoprecipitate.

Note that until now, the physician has still not entered the transfusion event in the patient record. This transfusion event is defined as the process of transfusing the blood product into the patient's blood stream. If the physician decides to transfuse the blood product, the doctor enters the transfusion event data on the corresponding transfusion form. For example, if the physician decides to transfuse red blood cells, he enters the transfusion event data on a red blood cell transfusion form.

Once the physician enters the corresponding transfusion event into the database, the system has to automatically perform auditing on this transfusion event. This auditing is performed by a set of standard procedures that are defined for each type of blood product. These audit procedures try to match the patient transfusion data with some pre-set audit criteria defined for each blood product. In the following sub-sections we will show each of these auditing procedures for the corresponding blood products.

Based on this auditing result, the transfusion event may then be flagged for further review depending on whether the patient transfusion data matched the audit criteria or not. All the transfusion events that are flagged for further auditing are then sent to the clinical services department for further review. The clinical services department is responsible for analyzing the transfusion data and produces a report that can be viewed by the physician.

In addition to the above auditing function of the system, the application also must provide searches on patient records based on different conditions and parameter levels. It also requires easy specification of different level of confidentiality of attributes depending on the class of user that is trying to access the attribute. In the next couple of sections, we address each of these requirements in more detail.

### 4.2.2 Auditing red blood cells

The figure 4-1 presents the algorithm that is followed for auditing each red-blood cell transfusion record.

In the procedure the pre-set flags *hemoglobin flag level*, *hematocrit flag level*, and *symptom* vary depending on the *Indication* of the patient. If the *Indication* is Surgical Blood loss, Trauma Blood loss, Exchange Transfusion, or Approved Protocol, the *hemoglobin flag level* is assigned "0", the *hematocrit flag level* is assigned "0" and the flag *symptom* is assigned "not required". However, if the *Indication* of the patient is Infection, Collagen Vasc Disease, Post Surgery, Burn, Vitamin, or Other, the *hemoglobin flag level* is assigned "70", the *hematocrit flag level* is assigned "21" and the flag *symptom* is assigned "required". For *Indication* of Malignancy, the *hemoglobin flag level* is assigned "80", the *hematocrit flag level* is assigned "24" and the flag *symptom* is assigned "required". Lastly, for Indication Chronically Transfused, the *hemoglobin flag level* is assigned "90", the *hematocrit flag level* is assigned "27" and the flag *symptom* is assigned "required".

```
FOR EACH red blood cell transfusion record
 Read the indication for transfusion
  IF hemoglobin and hematocrit is not checked for indication
          Do not flag for auditing
 OTHERWISE
  Assign hemoglobin and hematocrit flag level using preset audit criteria
    Read the hemoglobin and hematocrit level of current transfusion
    IF hemoglobin <= hemoglobin flag level OR
        IF hematocrit <= hematocrit flag level
           Do not flag for auditing
    OTHERWISE
        IF indication is valid, and symptom not required
           Do not flag for auditing
        ELSEIF indication is valid, symptom required
           Do not flag for auditing
        ELSE
           Flag for auditing
        END
   END
 END
ENDFOR
```

**Figure 4-1: Auditing Red Blood Cells**

### 4.2.3 Auditing whole blood cells

The figure 4-2 presents the algorithm that is followed for auditing each whole-blood cell transfusion record.

```
FOR EACH whole blood cell transfusion record
 Read the indication for transfusion
  IF indication is "OTHER"
            Flag for auditing
  ELSEIF indication is listed and not "OTHER"
          Do not flag for auditing
  ELSE
          Flag for auditing
  END
ENDFOR
```

**Figure 4-2 Auditing Whole Blood Cells**

This auditing procedure does not require any pre-set flag levels and is performed solely on the indication of the patient undergoing the whole blood transfusions.

### 4.2.4 Auditing platelets

The figure 4-3 presents the algorithm that is followed for auditing each platelet transfusion record.

In the procedure the pre-set flags *pre-platelet*, and *flag pre-plateletCount* vary based on the *Indication* of the patient. If the *Indication* of the patient suggests Thrombocytopenia, the flag *pre-platelet* is assigned "checked" and the flag *flag pre-plateletCount* is assigned a value of "20". However, if the *Indication* of the patient suggests Microvascular Bleedi or Major Surgery Prep, the flag *pre-*

*platelet* is assigned "checked" and the flag *flag pre-plateletCount* is assigned a value of "50". For other Indications of Thrombocytopathy, Approved Protocol, and other, the flag *pre-platelet* is assigned "not checked" and there exists no value for the flag *pre-plateletCount*.

```
FOR EACH platelet transfusion record
 Read the indication for transfusion
   IF indication is "OTHER"
           Flag for auditing
   OTHERWISE
   IF indication is valid and pre-platelet count is checked
          IF pre-plateletCount <= flag pre-plateletCount
            Do not flag for auditing
          ELSE
            Flag for auditing
          END
  ELSEIF indication is valid and pre-platelet count is not checked
      Do not flag for auditing
   ELSE
      Flag for auditing
   END
END
ENDFOR
```

**Figure 4-3: Auditing Platelets**

### 4.2.5 Auditing fresh frozen plasma

The figure 4-4 presents the algorithm that is followed for auditing each plasma transfusion record.

As with other audit procedures, the pre-set audit flags *flag pre-INRt*, and *pre-Inr* vary depending on the *Indication* of the patient. If the Indication of the patient undergoing plasma transfusion suggests Coagulopathy, Hepatic Failure or Warfarin Reversal, the audit flag *pre-Inr* is marked "checked" and the other flag *flag pre-INRt* is assigned a value of "1.5". For all other possible indications of Massive Bleeding, Antithrombin III, Heparin Cofactor, Protein C, Protein S, Immunodeficiency, Approved Protocol or Other, the flag *pre-Inr* is marked "not checked" and requires no value for the other flag *flag pre-INRt*.

```
FOR EACH fresh frozen plasma transfusion record
 Read the indication for transfusion
   IF indication is "OTHER"
          Flag for auditing
   OTHERWISE
   IF indication is valid and pre-INR is checked
          IF pre-INR <= flag pre-INRt
            Do not flag for auditing
          ELSE
            Flag for auditing
          END
  ELSEIF indication is valid and pre-Inr is not checked
       Do not flag for auditing
   ELSE
       Flag for auditing
    END
END
ENDFOR
```

**Figure 4-4: Auditing Fresh Frozen Plasma**

## 4.2.6 Auditing cryoprecipitate

The figure 4-5 presents the algorithm that is followed for auditing each cryoprecipitate transfusion record.

This procedure requires two pre-set audit flags: *PTT*, and *Fibrinogen* and these flags vary based on the *Indication* of the patient. If the *Indication* is Hemophilia, the flag *PTT* is assigned "checked", and no value is assigned to flag *Fibrinogen*. However for the Indication Hypofibrinogenemia, the flag *PTT* is assigned "not checked", and a value of "1" is assigned to flag level *Fibrinogen*. For all other indications, no value is assigned to these flags.

```
FOR EACH cryoprecipitate transfusion record
 Read the indication for transfusion
   IF indication is "OTHER"
           Flag for auditing
   OTHERWISE
   IF indication is valid and PTT is checked
          IF PTT entered
            Do not flag for auditing
          ELSE
            Flag for auditing
          END
  ELSEIF indication is valid and PTT and Fibrinogen not checked
       Do not flag for auditing
   ELSE IF indication is valid and PTT and Fibrinogen checked
          IF Fibrinogen  < 1.0 g/L
            Do not flag for auditing
          ELSE
            Flag for auditing
          ENDIF
   ELSE
       Flag for auditing
   ENDIF
 ENDIF
ENDFOR
```

**Figure 4-5: Auditing Cryoprecipitate**

## 4.3 System Users

The Blood Database System caters to three user classes in addition to a **SUPERVISOR** class. These user classes are:

**PATIENT**

These are the various patients of our system. Although actual patients never connect to the database at any moment for whatever reasons, their class is always linked to from the other two classes. This user class is required to have all the patient attributes in one file. The server creates a user ID for the patient when initiated by the DATAENTRY or the SUPERVISOR class user.

**DATAENTRY**

These are the various users who add patients into the system and enter their personal information, order blood products, and enter transfusion events into the database. Every member of this class has unlimited access to all patient files except for the comments of the clinical services. The user of this class can browse the clinical service comments, but cannot modify them.

**CSERVICES**

Members belonging to this class can browse through each patient's files (in the same way as the DATAENTRY class), but in contrast to a DATAENTRY class, he/she is not allowed to modify the file, except for adding/editing personal comments. Further, this class of user only has access to the patient files whose records have been flagged for auditing.

PATIENT-class users are always created and added to the database by the DATAENTRY-class.

**SUPERVISOR**

This is the user who is responsible for over-all administration of the users.

DATAENTRY and CSERVICES users are introduced into the database only by the supervisor. He is also responsible for adding/deleting users in the system.

## 4.4 Information linkage

In the previous section we introduced three user classes, PATIENT, DATAENTRY, and CSERVICES, and also studied their functional responsibilities. It is imperative that in such applications involving various user classes, each of these user classes has their own attributes as well as requirement for accessing the attributes of the different class. It is therefore important for us now to understand how these attributes are linked between each of these classes. In other words we need a specification that a certain user class can link and access information belonging to a different class.

Accordingly in the system, the PATIENT-class of user has no link either to its own information or the attributes belonging to the users of the other two classes. They represent a dummy user-class only to have all the patient attributes together. The DATAENTRY-class can link to the PATIENT-class of users and access the patient information. Similarly, the CSERVICES-class of users can also link to PATIENT-class of information. The DATAENTRY-class and the CSERVICES-class can also link to their own attributes respectively. However, there is no linkage requirement or specification between the DATAENTRY-class of attributes and CSERVICES-class of attributes.

## 4.5 Attribute accessibility

The Blood database system requires various levels of accessibility specification to preserve the confidentiality of different user-class attributes.

The DATAENTRY-class of users are doctors that are responsible for creation of PATIENT-class users. Accordingly, the DATAENTRY-class of user has full-unrestricted access to almost all PATIENT-class data. This class of user also has full access to its own attributes. Obviously, the PATIENT-class of user has no access to his/her own data, since it makes no sense in this application. The CSERVICES-class of users are the ones from the clinical services department. Therefore based on the operational requirements discussed in the previous section, this class of users requires two specifications in addition to the fact that this class of users can access their own data. One is that the CSERVICES-class of user has access to only those PATIENT-class users whose records are flagged for further auditing and second, this class of user has only access to browse the PATIENT-class information without the right to modify them. Also, the CSERVICES-class of user has exclusive rights to write and modify the CSERVICES comments for each flagged patient transfusion, which can only be browsed and not modified by the DATAENTRY-class of user.

To summarize the above specification requirements, the PATIENT-class of user has no access at all to its own information. Further, the DATAENTRY-class of user has full access to all PATIENT-class of user, unlike the CSERVICES-class of user. However, there are also situations when the DATAENTRY-class of user does not have full access over the patient attribute.

## 4.6 Implementation

In this section we explain our implementation to the proposed application using our server program, as described in the previous chapters.

A BLOOD database transaction is usually a series of related actions, e.g., filling out a sequence of forms, and viewing a sequence of forms. Therefore the BLOOD system implements sessions understood as sequences of related web transactions.

A session is executed within the context of a given user. To initiate a session, the user must log on to the system. The session will be terminated when the user logs off. As users have the natural tendency to abandon web sessions, a BLOOD session can time out. If there is no session-related activity for a prescribed timeout interval, BLOOD will close the session automatically. Note that dangling (i.e., abandoned but not terminated) sessions pose an obvious security risk as some malicious users can easily tamper it.

In the philosophy of BLOOD, a session must be committed, i.e., properly completed, to be considered valid and actually result in an update of the database record. An abandoned (or intentionally uncommitted) session is void: the database record is completely unaffected by the session. This way, we make sure that the database record is always consistent and its status is clearly determined from the viewpoint of the user. If there has been no committal (which is always explicit, though usually automatic at logout), the stored database record will look exactly as it did before the session.

In principle, different classes of users operate in different environments, in the sense that they see different forms during their sessions. For example, a DATAENTRY-class user is allowed to browse through the files of all patients and also view and modify their entire contents. To this end, he/she is offered simple (yet powerful) means to search through the files, pick files for processing, keep track of their audit status, and so on.

Most of the forms presented by BLOOD to the users are generated dynamically, based on form templates [2.9]. Essentially, the forms viewed by a CSERVICES-class user are almost the same as the ones seen by a DATAENTRY-class user. The only difference is that the information presented in editable text areas to a DATAENTRY-class user is hard-coded into the page shown to a CSERVICES-class user. Also there are some components within the same page template that would make it not appear to a particular class of user. For example, although the DATAENTRY-class user and CSERVICES-class user share the same page template, the page sent to a CSERVICES-class user should be void of the button to create and add new patient record, in contrast to the DATAENTRY-class user. Therefore, the templates are the same, but the interpretation and execution of the information has been made different and dynamic.

The last few paragraphs try to explain our server program in the context of the application at hand. This is done to familiarize and introduce more clarity to the reader with our generic server program. However the implementation details of the application, i.e., session templates, rules template, and forms template are presented separately in Appendix B.

## 4.7 Summary

In this chapter we first studied BLOOD database application, and it's functional and design requirements. We also studied the need for accessibility rights and demonstrated how our generic application server could be used to suit the BLOOD database application.

# Chapter 5

# Conclusions

There is a great demand for web application servers that offer specialized database services and more importantly there is demand for application servers that are very generic and can adapt easily to a wide family of web database applications. Through this thesis report, we first studied the various issues facing the current web developers in an attempt to build an application server that is easy to maintain, scalable, portable across various platforms, requires low bandwidth and facilitate rapid application development, in addition to providing some specialized database services. We then illustrated how scripting languages like Tcl would help achieve our desired goals. We have built a generic web application server based on Tcl with specialized features like dynamic page/content generation, easy specification of user accessibility rights, image uploading, and database querying. This application server can be used to rapidly build new web applications that require these specialized features. All it requires is to just code the application specific part, i.e., writing the session templates, rules templates and the page templates. We have also shown an example database application to illustrate building the application specific part and gluing it to our server module.

We have thus demonstrated the feasibility and effectiveness of a new tool to rapidly build web applications, with the objective of limiting the development cost for experienced programmers, and also making maintenance, extending the application an easy task for the web developer.

# Bibliography

[1] Scriptics Corporation, "Tcl/Tk 8.0 software tool kit," URL: http://www.scriptics.com

[2] Brent B. Welch, *Practical Programming in Tcl and Tk,* Prentice Hall PTR, 2nd edition, 1997.

[3] Mark Brown and Jerry Honeycutt, *Using HTML 4,* QUE, 4th edition, 1998.

[4] Pawel Gburzynski, *The SICLE Control Package,* Reference Manual, version 1.0, Department of Computing Science, University of Alberta, 1998.

[5] Don Libes, "Writing CGI scripts in Tcl," In *Proceedings of the fourth Annual Tcl/Tk Workshop*, pp. 189-201, July 1996.

[6] American Red Cross, *User Requirements Document for Transfusion Audit System,* version 2.0.0, August 1994.

[7] Jim Davidson, "Tcl in AOL Digital City: The Architecture of a Multithreaded High-Performance Web Site," America Online Inc., In *Proceedings of the seventh Annual Tcl/Tk Workshop*, pp. 132-141, February 2000.

[8] John K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer Magazine*, Vol. 31, No. 3, pp. 23-30, March 1998.

[9] Scriptics Corporation, "System Programming or Scripting?," URL: http://www.scriptics.com/scripting/choose.html

[10] Mark Harrison and Michael McLennan, *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk,* Addison-Wesley, 1998.

[11] Adam Sah, Kevin Brown, and Eric Brewer, "Programming the Internet from the Server-Side with Tcl and Audience1," In *Proceedings of the USENIX Fourth Annual Tcl/Tk Workshop*, pp. 183-188, July 1996.

[12] Alex Shah, and Tony Darugar, "Creating High Performance Web Applications using Tcl, Display Templates, XML, and Database Content," Binary Evolutions Inc., In *Proceedings of the USENIX Sixth Annual Tcl/Tk Workshop*, pp. 121-126, September 1998.

[13] Chris Scollo, Sascha Schumann, and Jason McKnight, *Professional Php Programming,* Wrox Press Inc., 1st edition, 1999.

[14] Andrew Feborchek, and David Rensin, *Developing Internet Server Applications with Microsoft Active Server Pages,* IDG Books Worldwide, 1st edition, 1997.

[15] Proposed Federal Information Processing Data Encryption Standard, Federal Register (40FR12134), March 17, 1975

# Appendix A

# Reference Manual

## A.1 Constructing Page templates

In this section we explain the different procedures that are responsible for constructing a page template. A brief description of their syntax, usage and operation of these functions are provided. We have also provided some small examples of these procedures wherever we found useful.

## initialise_page

### Overview

This function is used to construct the header of each HTML output page.

### Syntax

initialise_page "tex"

### Description

**initialise_page** *tex* function outputs the HTML code with the *tex* occurring as a banner. This function also assigns the background color of the HTML page. This is usually the first statement of every page template.

## initialise_form

### Overview

Initializes the driver program that should be invoked upon each form submission.

### Syntax

initialise_form prog att enc

### Description

The function requires three arguments as its input: the name of the driver program to invoke upon form submission, name of the tag, and an optional encoding style. The third argument is used only to distinguish between standard encoding and multipart encoding. If not present, the function assumes standard encoding.

If the function is called only with only two arguments, it outputs the following two HTML lines which are:

<form method = “post” action = “driver.cgi”>
<input type=“hidden” name=“tag” *value= “session-id”*>

The first line indicates that when the form is submitted, the web server should invoke **driver.cgi**. When this happens, the form information will appear on the standard input of the driver script. The second line defines an invisible attribute of the form named **tag**. Every form submitted to the server must have this attribute. When the form template is turned into an actual HTML form, the server assigns a value to the tag, *session-id*, which is a unique identifier, assigned to the session (section 3.4). The value of the tag is returned to the server (as a part of the submission information) when the form is submitted.

This way the server keeps track of multiple sessions. The above form initializing function assumes standard encoding type (by default), which is **application/x-www-form-urlencoded**.

To specify a different encoding style in case of image upload, a third argument "multipart" is required by the function. Therefore, when the function sees a third argument as "multipart", it outputs the following two tags.

```
<form method="post" action="driver.cgi" enctype="multipart/form-data">
<input type="hidden" name="tag" value="session-id">
```

In the first line **enctype** is used to specify multipart encoding. This is the only possible encoding type for an image submission. However, it is recommended to avoid multipart encoding for those forms that do not need it, as it is less efficient than the default application encoding.

## check_warning

### Overview

Displays warning messages if present

### Syntax

check_warning

### Description

The **check_warning** procedure is a Tcl procedure that checks whether the rules have come across any warning messages and if so, displays them. Otherwise the server simply neglects the above call. Every page template should have this call somewhere close to the top. The server checks these warning messages by means of **Inserts** array with the insert code **WA**. The insert code **WA** is used by the server to display warning messages in the outgoing HTML form.

## addtitle

### Overview

Adds a heading message

### Syntax

addtitle tit

### Description

**addtitle** procedure is used to provide headings in HTML. It provides an outline of the text that forms the body of each of the HTML document. The procedure provides the second level of heading and therefore inserts the HTML tags <h2> and </h2> before and after the message element respectively. One should note that the above level has its own appearance in the reader's viewer, and the programmer has no control over what that appearance is. This is part of the HTML philosophy.

## start_of_table

### Overview

Declares the start of table

### Syntax

start_of_table

### Description

The **start_of_table** is a Tcl procedure that outputs the HTML tag <table> required initializing a table. The procedure call does not require any arguments. The <table> element provides a container for the table's data and layout.

## end_of_table

### Overview

Declares the end of table

### Syntax

end_of_table

### Description

The **end_of_table** procedure unlike start_of_table outputs the HTML tag </table> required indicating the end of table. The procedure call does not require any arguments. The </table> element indicates an end of container for the table's data and layout. Each call of start_of_table should have a matching call for end_of_table.

## getinputtext

### Overview

Gathers a simple line of text

### Syntax

getinputtext typ att siz

### Description

**getinputtext** is a Tcl procedure that is used to gather a line of text. It takes three arguments as its input: input type, the attribute name and attribute size.

The input type can be of two types: *text* or *password*. The difference between these two input types is that *password* displays typed characters as bullets instead of characters actually typed as in case of default text type.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want a text field whose attribute name is *user_name* and size 20 characters. The appropriate call for the procedure as would appear on the page template is:

```
getinputtext text user_name 20
```

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<input type="text" name="user_name" size="20">
```

Now, this is not the end of the procedure operation. The server checks whether the attribute name *user_name* is accessible under the current session privileges. If so, it converts the above HTML tag into the following by appending the value field with the appropriate value of the attribute at the end.

```
<input type = "text" name = "user_name" size = "20"  value = “”>
```

This modified tag is then passed as an output of the procedure call to the driver.

If the attribute is not modifiable based on the session privileges, the above tag is replaced by a plain text that represents the value of the attribute and sent as an output. Note that if there is no value defined for the attribute, it just sends an empty text as an output.

The other usage of the procedure call is during password entry. An example usage of this type is as shown below:

```
getinputtext password pass_word 10
```

The only difference in the above call is that the above specification generates an HTML code in which the user-typed characters are displayed as bullets instead of the characters actually typed.

## initialise_radio_button

### Overview

Aligns the text with right justification

### Syntax

initialise_radio_button tit

### Description

**initialise_radio_button** procedure is used to align a table element with right justification. When the server finds the call “initialise_radio_button Username” in a page template it will output the following HTML code to the driver.

```
<td align = \"right\"> Username:</td>
```

## radio_option

### Overview

Generates radio button

### Syntax

```
radio_option att ival nam
```

### Description

**radio_option** is a Tcl procedure that is used to generate radio buttons. These radio buttons allows only one of a related set to be chosen. It takes three arguments as its input: the attribute name, option value and the text to be displayed to the user.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want to generate a radio field whose attribute name is *user_sex*, value is *male* and has to appear as *Male* to the user. The appropriate call for the procedure as would appear on the page template is:

```
radio_option user_sex male Male
```

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<input type="radio" name="user_sex" value="male"> Male
```

Now, this is not the end of the procedure operation. The server checks whether the actual value of the attribute *user_sex* is indeed the value of this option, which in our example is “male”. The server also determines whether the attribute name *user_sex* is accessible under the current session privileges. If yes, and also if the option value matches the actual value, it converts the above HTML tag into the following by appending CHECKED at the end.

<input type="radio" name="user_sex" value="male" **CHECKED**> Male

This modified tag is then passed as an output of the procedure call to the driver.

Otherwise, if the attribute is accessible under the current session privileges, but its option value is not the actual value of the attribute, it simply outputs the same tag as before.

However, if the attribute is not modifiable based on the current session privileges, and the option value matches the actual value of the attribute, the above tag is replaced by a letter “X” which indicates that the option value is checked. Note that if the option value does not match the actual value, the server just sends a “-“ as an output.

## check_option

### Overview

Generates a check button

### Syntax

check_option att ival tit

### Description

**check_option** is a Tcl procedure that is used to generate check boxes. It is almost similar to radio button and is used when there exists multiple selections for a given attribute and the choice is yes or no. It takes three arguments as its input: the attribute name, option value and the text to be displayed to the user.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want to generate a check box whose attribute name is *user_preferences*, value is *hockey* and has to appear as *Hockey* to the user. The appropriate call for the procedure as would appear on the page template is:

```
check_option user_preferences hockey Hockey
```

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<input type = "checkbox" name = "user_preferences" value = "hockey">
Hockey
```

Now the processing is slightly different from the radio_option only because the check boxes can take multiple selections. Therefore the server first determines if it is a multi-valued attribute or not. Note that this check only involves verifying the suffix of the attribute for "_mlt" (section 2.5). If yes, the server checks the option value with the values in the list. If the option value is present in the list, we are not yet done since we have not checked whether the attribute is accessible under the current session privileges or not.

If the attribute is accessible and the value is present in the list, the server outputs the following:

<input type = "checkbox" name = "user_preferences" value = "hockey" **CHECKED**> Hockey

This modified tag is then passed as an output of the procedure call to the driver.

Otherwise, if the attribute is accessible under the current session privileges, but its option value is not in the list, it simply outputs the same tag as before.

However, if the attribute is not modifiable based on the current session privileges, and the option value matches the value in the list, the above tag is replaced by a letter "X" which indicates that the check box is checked. Note that if the option value does not match the value in the list, the server just sends a "-" as an output.

## generate_select_option

### Overview

Generates a scrolling list for selection

### Syntax

generate_select_option name oplist

### Description

**generate_select_option** is a Tcl procedure that is used to generate a scrolling list. It takes two arguments as its input: the attribute name, and a list describing the different list elements. Each list member is a <value, name> pair where value is the actual value of the attribute and name is the representation of the value that is displayed to the user.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want to generate a selection list whose attribute name is *user_preferences*, and the different values of selection choices are hockey, soccer and football.

The appropriate call for the procedure as would appear on the page template is:

generate_select_option user_preferences "hockey Hockey soccer Soccer football Football"

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<select name = "user_preferences">
        <option value = “hockey”> Hockey
        <option value = “soccer”> Soccer
        <option value = “football”> Football
</select>
```

As shown above, the procedure processes the list in sequential order starting from left and moving towards right assuming that the list in <value,name> pair.

Now, this is not the end of the procedure operation. The server now determines the actual value of the attribute and checks against each of the option values. The server also checks whether the attribute name *user_preferences* is accessible under the current session privileges. If yes, and also if one of the option values matches the actual attribute value, it converts the appropriate option value tag into the following by appending SELECTED at the end.

In the above case, if the actual value selected were Soccer, the server would generate the following:

```
<select name = "user_preferences">
        <option value = “hockey”> Hockey
        <option value = “soccer” SELECTED> Soccer
        <option value = “football”> Football
</select>
```

However, if the attribute is not modifiable based on the current session privileges, and one of the option value matches the actual value of the attribute, the entire selection is replaced as a simple text representing the selected value. If none of the option value matches the actual value, the server just sends a “Unknown“ as an output.

## outputtextarea

### Overview

Generates a field allowing for multiple lines of input.

### Syntax

outputtextarea att col row

### Description

**outputtextarea** is a Tcl procedure that is used to generate text fields that allows one to enter multiple lines of information. The procedure accepts three arguments as its input: attribute name, number of rows, and the width of the field in characters. The number of rows and the width of the field are 4 and 40 by default respectively.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want to generate a text area whose attribute name is *user_comments*, and the number of rows are 6 and the character width is 60.

The appropriate call for the procedure as would appear on the page template is:

```
outputtextarea user_comments 6 60
```

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<textarea name = "usercomments" cols = "6" rows=”60">
</textarea>
```

Now, this is not the end of the procedure operation. The server checks whether the attribute name *user_comments* is accessible under the current session privileges. If yes, the procedure outputs the above field to the driver.

However, if the attribute is not modifiable based on the current session privileges, the procedure modifies the entire field by a single-row/single-column table containing a direct textual representation of the attribute's value.

## link_anchor

### Overview

Generates a special HTML link

### Syntax

link_anchor  tex par

### Description

**link_anchor** is a Tcl procedure that is used to generate a special HTML link whose URL is processed by the driver script (section 2.4.1). It accepts two arguments as its input: text and parameter. The text is the link test (to be highlighted) and parameter is the link parameter to be passed as an argument to the driver script.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want to generate a link to a page personal.htpl from a page selection.htpl with the link text as "Personal".

The appropriate call for the procedure as would appear on the page template selection.htpl is:

```
          link_anchor "Personal" "Sel_genpersonal"
```

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<ahref="driver.cgi?$Tag&Sel_genpersonal&$EventCount">Personal</a>
```

The above HTML tag shows two additional global parameters that may be of interest to the reader: *Tag* and *EventCount*. The *Tag* represents the unique session identifier assigned by the server and *EventCount* represents the serial number of the link. As explained earlier, the serial number is added by the server to make sure that subsequent editions of the same link (e.g., in different versions of the form) appear different, so that client and proxy caching is effectively disabled (section 3.3.1).

## link_anchor_target

### Overview

Generates a special HTML link with target specification

### Syntax

link_anchor_target  tex par tar

### Description

**link_anchor_target** is a Tcl procedure that is very similar to the link_anchor procedure except that this procedure has a third argument called *target*. It is used to generate a special HTML link whose URL is processed by the driver script and the linked information is presented in the target window. It accepts three arguments as its input: text, parameter and target. The *text* is the link test (to be highlighted), *parameter* is the link parameter to be passed as an argument to the driver script, and the *target* identifies the target browser window to display the information requested by the special link. If not specified, the linked information is presented in the current window as in the case of link_anchor procedure.

To understand the usage and execution of the above procedure, let us take a simple example.

Suppose you want to generate a link to a page *personal.htpl* from a page *selection.htpl* with the link text as "*Personal*" and you want the linked information in a different browser window named "*Personal Information*".

The appropriate call for the procedure as would appear on the page template *selection.htpl* is:

```
link_anchor "Personal" "Sel_genpersonal" "Personal Information"
```

When the server processes the above procedure call to generate the appropriate HTML code, it generates the following:

```
<ahref="driver.cgi?$Tag&Sel_genpersonal&$EventCount" target="Personal
Information">Personal</a>
```

The above HTML tag is very similar to the HTML code that is generated as a result of call to link_anchor procedure. The only difference is the addition of the target field, which specifies the name of the target window.

## insert_file

### Overview

Inserts a specified file at a specified location

### Syntax

insert_file fname

### Description

**insert_file** is a Tcl procedure that is used to insert a specified file at an appropriate position. The position of insertion is the point where the actual call of the procedure is placed in the page template.

This procedure is useful in cases when a certain information has to be presented in all the outgoing HTML pages. When the server receives such a request, it reads the content of the file, executes it and outputs the HTML code to the driver.

## insert_attribute

### Overview

Inserts a attribute value

### Syntax

insert_attribute att

### Description

**insert_attribute** is a Tcl procedure that is used to insert the value of the attribute at a desired position. The position is determined by the actual call of the procedure. It accepts only one argument as its input: attribute name.

When the server sees such a call, it retrieves the value of the attribute from the database and outputs the value as a plain text to the driver.

## plain_output

### Overview

Outputs the text as it is

### Syntax

plain_output tex

### Description

**plain_output** is a Tcl procedure that is used to output its input contents as it is. It accepts only one argument, which represents the actual contents to be passed to the driver. This procedure can be used to output any HTML contents that cannot be generated by our previous procedures. Examples of some of the usage of this procedure are as follows:

```
plain_output “<td>”
plain_output “<p> This is a test program”
```

## check_filetype

### Overview

Sets the flag for image upload widget

### Syntax

check_filetype

### Description

**check_filetype** is a simple Tcl procedure that is used to set a flag **Canupload**, used by the server for image upload operations. This procedure is usually called from the page template after there is an image upload field (i.e., input type file). The server uses this flag to determine whether the user has the right to delete an image that appears on the outgoing page. This right is only granted if the page actually allows the user to upload images.

A simple illustration of this procedure is as follows.

Suppose you want to upload an image *simp_tsh_img* from the form. The following sequence of statements would be required.

```
plain_output“<inputtype=\”file\”name= \”simp_tsh_img\”>
check_filetype
```

## A.2 Standard database functions

In this section, we present some of the standard functions for performing various database-related operations such as adding a user, retrieving an attribute from the database, searching the database, etc. Note that these functions execute as part of a function called by the server within the processing cycle of a form submission and in principle can use all functions of the server. However, it makes better sense if, as a matter of policy, the rules restrict themselves to the subset of those functions discussed in this section.

## addUser

### Overview

Adds a new user to the database

### Syntax

addUser username password uclass enc

### Description

**adduser** is a Tcl function that adds a new user to the database. It accepts four arguments as its input: username, password, userclass and an optional encoding field.

The *username* represents the name of the new user to be added. The *password* is the password for the new user. This password can be in plain text or it can be already encrypted, depending on the value of the last argument "enc". *uclass* presents the class of the new user. The user's record in the database will be initialized based on the session template associated with the specified class. *enc* is an optional argument. If it is 0 (by default), the argument *password* is assumed to be in plain text. It will be then encrypted (and salted) before it is stored in the authenticity file. If *enc* is 1, the password is assumed to be already encrypted.

If the function succeeds, it returns an empty string. Otherwise if the operation failed for whatever reason like if a user with the name already exists in the database, etc., it returns an error message diagnosing such a problem.

## changePassword

### Overview

Changes the password of an existing user

### Syntax

changePassword username password enc

### Description

**changePassword** is a Tcl function that is used to change the password of an existing user. It accepts three arguments as its input: username, password and encoding bit.

The function changes the password for user *username* to a new password *password*. The specified password is assumed to be either in plain-text (if enc is 0 or absent), or encrypted (if enc is 1). If the function succeeds, it returns an empty string; otherwise, the returned string is an error message diagnosing the problem.

## deleteUser

### Overview

Deletes a specified user from the database

### Syntax

deleteUser username

### Description

**deleteUser** is a Tcl function that is used to delete a specified user from the database. The function accepts one input as its argument: username. The function returns 1 if the operation is successful, else 0. The operation is assumed to be successful even if the specified user didn't exist in the database.

## existsUser

### Overview

Checks whether the user exists

### Syntax

existsUser username

### Description

**existsUser** is a Tcl function that checks whether the specified user exists in the database. It accepts one input as an argument: username. The function returns 1 if the specified user exists in the database, else 0.

## validUserName

### Overview

Checks whether the username specified is syntactically correct

### Syntax

validUserName username

### Description

**validusername** is a Tcl function that determines whether the username specified is syntactically correct or not. It accepts one input as its argument: username. The function returns 1 if the specified user name is formally valid and 0 otherwise. A valid user name is no more than 32 characters long, starts with a letter (lower or upper case), and contains letters and/or digits only.

## userAttributes

### Overview

Retrieves the specified attributes of a given user from the database

### Syntax

userAttributes user attlist

### Description

**userAttributes** is a Tcl function that can be used to retrieve a set of attributes that belong to a particular user. The function takes two arguments: username and a list containing the attribute names (attlist).

Given the list of attribute names (attlist), the function returns the list of their values (in the same order) extracted from the database record of the specified user. The value of a nonexistent attribute is assumed to be an empty string.

## inaccessible

### Overview

Tells whether the attribute is accessible or not

### Syntax

inaccessible attr

### Description

**inaccessible** is a Tcl function that determines whether the given attribute is accessible or not under the current session privileges. It takes one argument as its input: attribute name.

This function returns 1 if the session attribute specified in the argument is inaccessible, and 0 otherwise.

## sessionAttribute

### Overview

Returns the value of the attribute

### Syntax

sessionAttribute aname

### Description

**sessionAttribute** is a Tcl function that is used to return the value of the attribute from the database. It takes one input as its argument: attribute name.

Note that the function returns an empty string for a non-existent attribute.

## searchUsers

### Overview

Searches user records for certain matching conditions

### Syntax

searchUsers { class params limit { ulist } }

### Description

**searchUsers** is a Tcl function that allows to perform searches in the database. This function accepts four arguments as its input: user class, list of parameters, limit on the number of searched records, and an optional list of user names to search for.

The first argument gives the class of users to which the search should be limited. As the search criteria are based on record attributes, it generally makes no sense to perform a single search among users of different classes. The second argument is a list containing search parameters. Each search parameter is a triplet containing three members. This will be explained in depth shortly. The third argument (limit) specifies the maximum number of user names that can be returned by the function. As soon as that many users have been found that fulfills the search criteria, the search will be terminated.

The fourth optional argument is a list of user names to search for. If present, the search is limited to those users alone. Otherwise, all users are examined. The list of user names whose database records fulfill the search criteria described in *params* is returned as the function value.

The search criteria *params* as discussed are described by a list of three-element lists <attribute, range, relation>. One triplet can be viewed as a single criterion.

A user record is selected by the search operation if and only if it meets all search criteria.

The first element of a criterion identifies the database attribute whose value is to be examined. The second element represents a range for that value. The last item tells how the attribute value should be related to the range. The possible relations are:

**equu:** Equality
The criterion is fulfilled if the attribute value exactly matches the second element of the triplet.

**gtr:** Greater
The attribute value must be a number that is greater than the number specified as range.

**les:** Lesser
The attribute value must be a number that is less than the number specified as range.

**sub:** Substring of
The criterion is met if and only if the attribute is defined and range is a substring of its current value.

**rcp:** Pattern matching
In this case, attribute may be a list of attribute names. The criterion is met if the concatenated values of all the attributes (separated by a single space) match range viewed as a regular expression. The case of letters is ignored for this matching, as well as for other regular expression matching performed by

*searchUsers*. If one or more of the specified attributes is undefined, the criterion fails. Note that the list of the attributes to match may consist of a single attribute.

**rcn:** Does not match
The same as **rcp** except that the concatenated attribute values must not match the regular expression for the criterion to be met. As before, the criterion fails if one or more of the specified attributes are undefined.

**rmp:** Pattern matching
Similar to **rcp**, except that each of the multiple attributes is matched individually against the regular expression. The criterion is met if at least one of them matches the regular expression, even if some attributes are undefined. Note that rcp and rmp are equivalent, if the first argument of the criterion identifies a single attribute.

**rmn:** Does not match
The reverse of **rmp**. All defined attributes from the list must not match the regular expression for the criterion to be met.

To illustrate the above search mechanism, consider the following example.

```
set crit {
  {salary 4000 gtr}
  {{cuv_experience cuv_statement}
    "computer.*networks|implemented"
      rcp}}
set ul [searchUsers APPLICANT $crit 10000]
```

The above search criteria is an example that can be used to search all APPLICANT class users whose salary is greater than 4000 and whose "experience" and "statement" field from their CV contains either "computer" followed by "networks" or the word "implemented".

The above search also tells the *searchUser* procedure to search for the first 10000 applicants only. The return result is put in the variable *ul*.

Note that by using the result returned by one invocation of searchUser as the last (optional) argument of another one, it is possible to perform complex (refined) searches.

## A.3 Miscellaneous User functions

This section explains the various utility functions that are provided by the server to be included and used by the session rules.

## addWarning

### Overview

Add a warning message

### Syntax

addWarning mess

### Description

**addWarning** is a Tcl function that is used to display a warning message. The function takes one argument as its input: messsage.

This function adds a new warning message to the **WA** insert. This insert starts empty at the beginning of every submission. If a rule (or the server) detects a problem with the form submission (e.g., an incorrect or inconsistent value of a submitted attribute), it can add a warning message to the **WA** insert by executing *addWarning*.

Every form template should include the special sequence *check_warning*, as previously discussed preferably close to the top. When the form is presented to the user, this sequence will be replaced by the current contents of the WA insert if present. Also, by checking whether Inserts(WA) is nonempty, a rule can tell whether any problems with the last submission have been diagnosed so far.

## sessError

### Overview

Displays any fatal errors

### Syntax

sessError [-fatal] mess

### Description

**sessError** is a Tcl procedure to present any fatal errors during a session. It accepts one input as its argument: message.

This function can be called when the rules detect a serious problem with the session, i.e., something that shouldn't happen in the normal course of action. Note that an erroneous submission (i.e., an incorrect attribute value) does not qualify as a serious problem.

If the switch [-fatal] is present, the server aborts the current session. Otherwise, the user has a chance to continue by hitting the "Back" button of the browser and trying to resubmit the last form. In both cases, the user is presented a special error page (file *error.nfrm* in the server directory) displaying *mess* as the error message and explaining the server's action.

Note that *error.nfrm* is a non-form template. The only difference in processing such a template, as compared to a regular form template (suffix .htpl), is that a non-form is not remembered by the server as the last form presented to the user.

## linkAnchor

### Overview

Creates a linkage anchor

### Syntax

linkAnchor user text

### Description

**linkAnchor** is a Tcl procedure that is used to create a linkage anchor. Its purpose is to create a special HTML link (typically to be included in the outgoing form, e.g., through an insert) representing a linkage request. It takes two inputs as its arguments: user name and text.

The first argument is the user name for the slave session and the second argument is the highlighted text of the link. The server transforms the above procedure call into the following HTML code.

"<ahref=\"driver.cgi?${Tag}&**LINK{user}**&$EventCount\">${text}</a>"
where, driver.cgi is the driver program that processes the linkage request, Tag is the session identifier, and EventCount is the serial number of the link.

## setLinkage

### Overview

Links to a slave session for the specified user

### Syntax

setLinkage user

### Description

**setlinkage** is a Tcl procedure that is used to create a slave session for the specified user. The function links the current session to it. The value returned is either 1 (if the function has succeeded), or 0 (if it has failed). If setLinkage fails, it invokes sessError on its own indicating that the session cannot satisfy the linkage request. Thus, the caller doesn't have to worry about diagnosing the failure.

## runRules

### Overview

Runs the rules defined for a given session

### Syntax

runRules

### Description

**runRules** is a Tcl function that (re)executes the session rules. The only place where it makes sense to call it after a successful invocation of setLinkage, so that the rules for the newly linked slave session can determine the first OutgoingPage to be presented to the user.

## tidyText

### Overview

Formats a string

### Syntax

tidyText text

### Description

**tidyText** function takes as argument a text string (typically the value of a textarea field) and formats it by prepending "<p>" in front of every line and adding "<br>\n" at the end. The result, which is suitable for inclusion in an HTML document, is returned as the function value.

## isImageFile

### Overview

Checks whether the attribute is a name of an image file

### Syntax

isImageFile att

### Description

**isImageFile** is a Tcl function that determines whether its input argument is a name of an image file. Obviously, the legitimate input to this function is a file name. The function checks an "img" field in the attribute and if present returns 1, else 0.

## isLocalImageFile

### Overview

Checks whether the file name is a local image file name

### Syntax

isLocalImageFile  att

### Description

**isLocalImageFile** is a Tcl function that determines if the file name is a local image file name, i.e., one belonging to the data base, as opposed to being a generic image. If so, it returns a 1, else 0.

## isIDeleteAttribute

### Overview

Determines if the attribute represents an image deletion request

### Syntax

isIDeleteAttribute att root

### Description

**isIDeleteAttribute** is a Tcl function that checks whether the attribute represents an image deletion request. The function accepts two input arguments: attribute and image attribute name.

The function checks whether it is a deletion request by pattern matching the attribute with field "imd". If the attribute represents an image deletion request (means that the field "imd" is present in the attribute), it returns a 1. It also returns the image attribute name in root.

## isMultivalued

### Overview

Determines whether the attribute is multi-valued

### Syntax

isMultivalued { att }

### Description

**isMultivalued** is a Tcl function that determines whether the input argument is multi-valued or not. The function checks for this by pattern matching the attribute with the field "mlt". If the attribute has a "mlt" field, it returns a 1, else 0.

## A.4 Standard Server procedures

In this section we explain the various important server procedures that were written for constructing a session. The main reason for this introduction is to illustrate the user about how our server goes about building and then constructing a session.

## fetchUser

### Overview

Reads user attributes to the session

### Syntax

fetchUser { username uclass { linker "" } }

### Description

This procedure is used to move the user data to the session.

The procedure first checks whether the user directory is already present in the SESSIONS directory. Note that if it is present, it means that some body else is working on it. Therefore the procedure generates a fatal error indicating that the user file requested is currently locked. Otherwise, the procedure opens a file *server,* and writes the host name and the port number into this file. It also creates a temporary directory for images. It then makes a call to build the session and once the session is over, it removes the user directory from the SESSIONS directory.

## buildSession

### Overview

Builds a session

### Syntax

buildSession { username class linker }

### Description

This procedure is used to build a session.

The server in order to build a session has to first preprocess the rules file. It therefore tries the open the *rules* file in the TEMPLATE directory, executes and preprocesses it. The procedure then opens the session rules file for the corresponding user class and preprocesses that too. It then creates a session tag, which uniquely identifies the session. The procedure then creates a session process and fetches the session attributes from the database.

## getSession

### Overview

Fetches user attributes from the database

### Syntax

getSession { user stype linker }

### Description

This procedure is used by the server to fetch the user attributes from the database. It reads the session template for the corresponding class of user and fetches the default attribute values, the session time out, and the linkage declarations. The procedure then sets the access for each of these attributes and reads the database into session.

### readSessionTemplate

#### Overview

Reads the session template

#### Syntax

readSessionTemplate { stype }

#### Description

This procedure reads the session template into memory for the specified user class. It then splits the file contents into four arrays: attribute defaults, attribute patterns, error messages, and access type. The attribute default represents the default value of the attribute. The attribute patterns convey the syntactic representation of the attribute value. The error message represents the message that the user will see if the entered value does not match the specified pattern. Finally the access type is one of five letters A, B, C, D or E depending on the access specification as discussed above.

## setAccess

### Overview

Sets the accessibility rights

### Syntax

setAccess{ sess pist uast}

### Description

This procedure is used to set the access rights for a given session. The procedure reads the fourth field of the corresponding session template and then accordingly puts the attribute into one of the two lists: pyes or pno. It puts all the accessible attributes into the list pyes and the entire inaccessible attributes into pno.

## processPage

### Overview

Executes the page template

### Syntax

processPage { line }

### Description

This procedure is responsible for executing the page template. Once the server and rules determine the outgoing page, the procedure opens the page template, executes it and outputs the HTML to the driver.

## A.5 Status Attributes

The server stores various status attributes that form a special category of session attributes in an array called *Status*. They represent certain internal properties of the session that are neither read from form fields nor stored in the database. To make them distinct and separate from other attributes, they are kept in an array called Status. This array is viewed as a single (albeit compound) attribute of the session object.

Below we list all status attributes and explain their meaning.

**Status(User)**

This attribute contains the user name of the session.

**Status(SType)**

This attribute gives the class identifier of the current session, e.g., Class1, Class2 or Class3

**Status(IPAddr)**

This attribute gives the IP address of the current session client. If Obsessive was set to 1, this attribute can never change, as the session must be carried out entirely from a single (original) host.

**Status(Tag)**

This attribute contains the complete session tag, including the parameters of the server host. For brevity, this attribute can be referenced as **sessionAttribute** *tag*, although formally there is no database session attribute named tag.

**Status(STag)**

This attribute holds the 8-character session identifier, which is equal to the session tag (Status(Tag)) with the host parameters stripped off.

**Status(Timeout)**

This attribute stores the session timeout in minutes. This is equal to the maximum idle time of the session, after which it should be automatically closed and terminated.

**Status(Timer)**

If this attribute exists, it points (via a SICLE handle) to the SICLE process implementing a timer for the session. This timer is used to close the session automatically, if there has been no session-related activity for a prescribed amount of time.

**Status(Link)**

This attribute is defined for a slave session that has been created and linked to from another session. It points (via a SICLE handle) to the SICLE object representing the master session.

**Status(NLinks)**

This attribute counts the number of slave sessions of this current session. Note that it is possible (and quite natural) to be linked to several slave sessions at the same time, possibly, but not necessarily, in different windows. A counter is needed to avoid deallocating the session on a timeout, for as long as it has non-terminated slave sessions. The inactivity timer is disabled for such a session until Status(NLinks) goes to zero.

Note that no matter how sessions are linked, the ones that have no slave sessions will time out if they are inactive. Once all slave sessions of a given master session have timed out, the inactivity timer of the master session is resumed. This way, any abandoned session will eventually time out.

**Status(BSType)**

This attribute gives the class name of the master session linked to this session. For example, if a Class2 user links to a Class1 session, the Status(BSType) attribute of the Class1 session will be Class2. For a session that has been created on its own, and has no master session, Status(BSType) is equal to Status(SType) (see above). This attribute is used for protection.

**Status(AccN)**

This attribute is a list of attributes describing those session attributes that are inaccessible (i.e., excluded from access) by the current user.

**Status(AccY)**

This attribute is a list of attributes describing those session attributes that are explicitly made accessible to the current user.

**Status(LastPage)**

This attribute stores the file name of the last form template that has been presented to the user as part of this session. This filename is relative to the server directory, i.e., it contains no directory prefix.

For every slave session to which the current session has linked, there exists one status attribute of the following form:

**Status(Links,stag)**

Where, *stag* is the 8-character session identifier of the slave session. Note that a session may have multiple slave sessions at the same time, but there can be no more than one master session for each slave.

## A.6 Insert Codes

The session rules may often dynamically generate some strings that would require being included in the outgoing page. Examples of these are the warning messages that a session rule may want to generate as a result of exception conditions or in some cases also error messages. The server therefore allows these messages to be included in the outgoing page by means of a special Inserts array. The server provides three standard insert codes (xx): **EM**, **EC** and **WA**. An Insert is always called or referenced by these two-letter insert codes (ex. Inserts(xx)).

The first two insert codes, **EM** and **EC**, are inserted into the standard session error page (*error.nfrm*) to describe the error. The third insert code, **WA**, is used to display warnings in all outgoing forms. Every form template should therefore include this insert code (*check_warnings*) somewhere close to the top of the page template. If there are no messages in the list, the insert call will be ignored.

## A.7 Example of a page template

An example of a page template is as follows;

```
initialise_page "Transfusion Audit System (NAME)"
initialise_form driver.cgi tag
check_warning
addtitle "Patient Information"

start_of_table

plain_output "<tr> <td align = \"right\"> Surname: </td>"
plain_output <td>
getinputtext  text pat_surname 15
plain_output "</td> </tr>"
plain_output "<tr> <td align = \"right\"> First name: </td>"
plain_output <td>
getinputtext text pat_firstname 15
plain_output </td></tr>\n
plain_output "<tr> <td align = \"right\"> Middle name: </td>"
plain_output <td>
getinputtext text pat_middlename 1
plain_output </td></tr>\n

plain_output "<tr>"
initialise_radio_button Sex
plain_output "<td>"
radio_option pat_sex male Male
radio_option pat_sex female Female
plain_output "</td> </tr>"


plain_output "<tr> <td align = \"right\"> Patient DOB: </td>"
plain_output <td>
getinputtext text pat_birthdate 10
plain_output </td></tr>\n
plain_output "<tr>"
plain_output "<td align=\"right\"><b>Patient ABO/RH:</b></td>"
plain_output "<td>"
generate_select_option "pat_aborh" "OPOS OPOS\
APOS APOS BPOS BPOS ABPOS ABPOS ONEG ONEG\
ANEG ANEG BNEG BNEG ABNEG ABNEG"
```

```
plain_output "</td>"
plain_output "</tr>"
end_of_table

plain_output "<hr>"

plain_output "</body>"
plain_output "</html>"
```

The above template represents a sample page template that the server holds before being presented to the user. When the server determines that this is the page to be presented to the user, the server opens the respective page template, and executes it. The server processes each of these procedure calls in sequence and outputs the corresponding HTML code directly to the driver output.

The above example outputs the HTML page with the banner as “Transfusion Audit System (NAME)”, and a heading of “Patient Information”. The template also includes calls to various other procedures that generates text-input fields, radio buttons, select-options and other HTML tags.

# Appendix B

# Implementation of BLOOD application

## B.1 Session template

In chapter 2 we saw the purpose of having session templates and also saw that each class of user in the system must have a corresponding session template (section 3.8.1). In this section we present a fragment of the session template of each of the user classes of the BLOOD database system and explain each of them in detail.

A fragment of the patient session template as present in the server directory "/TEMPLATE/PATIENT/template" is as shown in figure B-1.

```
{TIMEOUT "90"}
{discard_on_close "" "yes"}
{rbc_audit "A" "" "" "" }
{wbc_audit "A""" "" "" }
{ffp_audit "A""" "" "" }
{plat_audit "A" "" "" "" }
{cryo_audit "A" "" "" "" }
   . . .
{pat_firstname "A" "" "^[A-Za-z-]+$" "First name appears to be invalid"}
{pat_birthdate "A" "yy/mm/dd" "^(19)?[5678][0-9]/(0?[1-9]|1[0-2])/(0?[1-
 9]|[12][0-9]|3[01])$" "Birth date appears to be invalid"}
   . . .
```

**Figure B-1: PATIENT-class Session Template**

The above template starts with the TIMEOUT specification indicating that the session should timeout after "90" minutes of inactivity (section 3.8.1.1). Please note that in the above session template we do not have any specification regarding the class of users. This is because in our application, they represent only a dummy class. However, the session template has several attributes.

Each attribute is a Tcl list consisting of two, three or five fields. The first field represents the attribute name, second field presents the attribute class to which the attribute belongs to, third field specifies the attribute default value, fourth field specifies a valid verification pattern for attribute value, and the fifth field specifies the error message to display to the user in case the attribute value doesn't match the valid pattern. The attribute class may be assigned to any one in the following set {A, B, C, D, E} and is assigned based on the access requirements (section 2.8). Here in the above template all the attributes are specified as belonging to A-class. This means that the PATIENT-class user cannot access the attribute at all, but however, the DATAENTRY-class has full access to the attribute and CSERVICES-class has read-only access. If there is no specification regarding the attribute class, the server assumes that the user has no access over the attribute.

A fragment of the DATAENTRY-class session template as present in the server directory "/TEMPLATE/DATAENTRY/template" is shown in figure B-2.

The first two lines of the template convey that a DATAENTRY-class user can access his/her own attributes, and also link to the PATIENT-class user. Note that these two lines are only for initialization purposes and by no means give full access to all the attributes. The access rights are further specified at the attribute level and can be different for different attributes. The third line specifies the timeout interval of 90 minutes for session timeout. The rest of the attributes follow the same specification as that of the PATIENT-class.

```
{OWN ""}
{PATIENT ""}
{TIMEOUT "90"}

{sel_selfile "B" "" "" ""}
{sel_status "B" "any" "" ""}
{sel_flagged "B" "no" "" "" }
{sel_name "B" "" "" ""}
{sel_bloodproduct "B" "any" "" ""}
      . . .
{sel_hemo "B" "" "" ""}
{sel_hemomode "B" "" "" ""}
{sel_platcount "B" "" "" ""}
{sel_platmode "B" "" "" ""}
      . . .
```

**Figure B-2: DATAENTRY- class Session Template**

A fragment of the CSERVICES session template as present in the server directory "/TEMPLATE/CSERVICES/template" is shown in figure B-3. This template almost resembles the DATAENTRY session template and therefore requires no further explanations.

```
{OWN ""}
{PATIENT ""}
{TIMEOUT "90"}

{sel_selfile "B" "" "" ""}
{sel_status "B" "any" "" ""}
{sel_flagged "B" "yes" "" ""}
{sel_name "B" "" "" ""}
{sel_bloodproduct "B" "any" "" ""}
{sel_indication "B" "any" "" ""}
      . . .
{sel_limit "B" "10" "^[1-9][0-9]*$" "Search limit must be a
 decimalnumber>0"}
{sel_hemo "B" "" "" ""}
{sel_hemomode "B" "" "" ""}
      . . .
```

**Figure B-3: CSERVICES-class Session Template**

## B.2 Session rules

The session rules describe the functionality and the behavior of the application. As described in chapter 2, the primary purpose of the set of rules is to determine the next form to be presented to the user (determined by the variable *OutgoingPage*) (section 3.8.2). Each class of user has a rules template file, **rules**, that includes all the rules that are applicable to the session of that user class (section 3.2.2). These rules are all defined in this template by means of **apply** operation. The order of apply operations determines the order in which the indicated rules will be executed after every form submission within the session. The methods of these rules are all defined in global **rules** file, which contains the description of all the rules that are applicable to all the classes, defined in the application.

Now, in the context of our application on hand, the rules template for PATIENT-class session as present in the server directory "TEMPLATES/PATIENT/rules" is shown in figure B-4.

```
apply gen_selection
apply gen_anchor
apply gen_verify
apply gen_person
apply gen_bloodorder
apply gen_rbc
apply gen_wbc
apply gen_plasma
apply gen_platelet
apply gen_cryo
apply gen_firstpage
```

**Figure B-4: PATIENT-class Rules Template**

Similarly, the rules template for DATAENTRY-class session as present in the server directory "TEMPLATES/DATAENTRY/rules" is shown in figure B-5.

```
apply com_fixsel
apply com_mail
apply com_logout
apply com_newapp
apply com_popup
apply com_delete
apply com_erase
apply com_linkage
apply com_search
apply com_reports
apply com_firstpage
```

**Figure B-5: DATAENTRY-class Rules Template**

The rules template for CSERVICES-class session as present in the server directory "TEMPLATES/CSERVICES/rules" is shown in figure B-6.

```
apply com_mail
apply com_logout
apply com_popup
apply com_delete
apply com_erase
apply com_linkage
apply com_search
apply com_reports
apply com_firstpage
```

**Figure B-6: CSERVICES-class Rules Template**

We may observe that the rules template for DATENTRY session as well as CSERVICES session differs only with respect to rule "com_newapp". This is because we have already mentioned that only DATAENTRY-class of users can add new patients. As discussed before, the method for each of these rules will be present in the global **rules** file present in the server directory "TEMPLATE/". In the next sub-section we deal with this global rules file and their ability to meet the functional requirements of the application.

## B.3 Execution of rules

As described in chapter 4, the BLOOD database system's primary responsibility was to audit each transfusion records against some pre-set audit criteria (section 4.2). Now in this section we explain how these audit procedures are implemented by the rules. We also explain how the rules process the special linkages and other user selections.

In the previous section (B.2) regarding the patient's rules template file we saw the following: "apply gen_rbc", "apply gen_wbc", "apply gen_plasma", "apply gen_platelet", and "apply gen_cryo". These rules are used to verify the semantics of the corresponding blood product order form and also perform the auditing. Now, let's have a look at the rule in figure B-7 whose method is present in the global rules file in the server directory "/TEMPLATE".

```
rule gen_rbc {
 if [info exists Changed(ord_rbc_trmdid)] {
  set error 0
  set nn [sessionAttribute ord_rbc_trmdid]
  set hn [sessionAttribute pat_rbc_hemoglobin]
  set sn [sessionAttribute pat_rbc_symptoms]
  set dn [sessionAttribute pat_rbc_trdate]
  set ln [sessionAttribute pat_rbc_leukodepleted]
  if { $nn != ""} {
    if { $hn == ""} {
    addWarning "You have to specify the Haemoglobin level"
    set error 1
    } elseif { $hn == ""} {
    addWarning "You have to specify the Symptom"
    set error 1
    } elseif { $dn == "yy/mm/dd"} {
    addWarning "You have to specify the Date of Transfusion"
    set error 1
    } elseif { $ln == "yes"} {
    set pre [sessionAttribute pat_rbc_prestorage]
    set bb [sessionAttribute pat_rbc_bloodbank]
    set bs [sessionAttribute pat_rbc_bedside]
```

```
   set res [sessionAttribute pat_rbc_reasonforuse]
     if { $pre == "" && $bb == "" && $bs == "" && $res == ""} {
     addWarning "You have to specify the leukodepleted reason"
     set error 1
     }
   }
}

 if $error {
 set OutgoingPage "genrbc"
 break
 } else {
 set indicate [sessionAttribute pat_rbc_indication]
if { $indicate == "SURGICAL" || $indicate == "TRAUMA" || \
 $indicate == "EXCHANGE" || $indicate == "APPROVED"} {
 set hgblevel 0
 set hctlevel 0
 set symptom no
 } elseif { $indicate == "INFECTION" || $indicate == "POST" \
 || $indicate == "COLLAGEN" || $indicate == "BURN" ||\
  $indicate == "VITAMIN" || $indicate == "OTHER"} {
 set hgblevel 70
 set hctlevel 21
 set symptom yes
} elseif { $indicate == "MALIGNANCY"} {
 set hgblevel 80
 set hctlevel 24
 set symptom yes
 } elseif {$indicate == "CHRONICALLY" } {
 set hgblevel 90
 set hctlevel 27
 set symptom yes
 }
if{[sessionAttribute pat_rbc_hemoglobin] == ""\
  && [sessionAttribute  pat_rbc_hematocrit] == "" } {
       set flag "-"
       setattr $Session rbc_audit $flag
} else {
 set hemo [sessionAttribute pat_rbc_hemoglobin]
 set hema [sessionAttribute pat_rbc_hematocrit]

   if { $hemo <= $hgblevel || $hema <= $hctlevel } {
     set flag "-"
     setattr $Session rbc_audit $flag
   } elseif { $symptom == "no" } {
    set flag "-"
```

```
     setattr $Session rbc_audit $flag
  } elseif { $symptom ==  "yes" && \
    [sessionAttribute pat_rbc_symptoms] != ""} {
   set flag "-"
   setattr $Session rbc_audit $flag
  } else {
   set flag "X"
   setattr $Session rbc_audit $flag
  }
 }
}
#End of Error else
gen_submit genrbc
break
 }
}
```

**Figure B-7: Rule for auditing RBC**

The above rule gets executed when the session attribute *ord_rbc_trmdid* has changed. Note that this does not mean that value of the attribute has in fact changed, but only that it has arrived in the last submission. The rule then checks for form completeness. This is an attempt to check whether the user has indeed entered all the required information that is needed to perform a RBC transfusion. If the rule detects that the user did not fill some of the required attributes, it prepares a warning message, sets *OutgoingPage* to the same page template **genrbc** and re-submits the form with the warning. This is done by function **gen_submit**, which is presented in figure B-8.

However, if the form is void of any errors and is complete, the rule performs auditing based on the procedure discussed in section 4.2.2, and accordingly sets the audit flag *rbc_audit* to either X or -. "X" indicates that the record is flagged for further review and "-" indicates that the record does not require further auditing.

The other rules gen_wbc, gen_plasma, gen_platelet and gen_cryo are also coded in the similar fashions.

```
proc gen_submit { from } {
#
# Page distributor
#
 useown OutgoingPage
 upvar Changed Changed
 if ![info exists Changed(Submit)] { return }
 set val [sessionAttribute Submit]
 switch $val {
  Personal     { set OutgoingPage "genperson"             }
  Order        { set OutgoingPage "genorder"              }
  RBC          { set OutgoingPage "genrbc"                }
  WBC          { set OutgoingPage "genwbc"                }
  CRYO         { set OutgoingPage "gencryo"               }
  FFP          { set OutgoingPage "genplasma"             }
  PLAT         { set OutgoingPage "genplatelets"          }
  Logout       { gen_close                                }
  Close        { gen_close                                }
  Next         { set OutgoingPage [gen_nextpage $from]    }
  Previous     { set OutgoingPage [gen_prevpage $from]    }
  Upload       { set OutgoingPage $from                   }
  default      { set OutgoingPage "genselect"             }
 }
 gen_checkpage
}
```
**Figure B-8: Procedure for Page Distribution**

Let us now talk a little bit about the **gen_submit** procedure. Although session rules execute in some predefined environment with immediate access to some standard variables, functions defined in the rules file are just regular Tcl functions. Thus they have to make sure by their own means that they perceive the objects they want to access. All such functions execute within the environment of the SICLE object representing the current session context. More specifically, this object is a SICLE process that has been created to take care of the current submission. Variables like *OutgoingPage*, *Commit*, *Cancel*, *Inserts*, and also *Session* containing the SICLE handle to the actual session object, are attributes of the context process. They can be made visible to a function invoked by a session rule by the *useown* operation of SICLE. Based on the value of the Submit button, **gen_submit** selects the proper outgoing form. The functions, **gen_nextpage** and **gen_prevpage** define the next and previous form in the session, to implement the functionality of the "Next" and "Previous" buttons as shown in figure B-9 and B-10 respectively. The functions, **gen_nextpage** and **gen_prevpage** are presented as follows. Each of these procedures takes in one argument that represents a page name and the procedure returns either the next or the previous page respectively. An important point to be observed in the procedure **gen_nextpage** is with the input parameter *genorder*. In this state, the procedure determines the blood product that is ordered and appropriately returns the corresponding transfusion form as the next page. This implements the idea of generating the appropriate transfusion form based on the blood product that is ordered.

It may also be worthwhile to have a look at function **gen_close** as shown in figure B-11, which terminates the session.

The function "gen_close" invokes **gen_complete** (another auxiliary function defined in the rules file) to determine whether the present configuration of session attributes comprises a sensible application. In other words, the function checks whether the record is complete enough in all respects to be termed as a proper patient record. If not, *OutgoingPage* is set to the form that will warn the user that the application is incomplete and will be discarded, if the user closes or abandons it in this state. If **gen_complete** returns 0, it also builds an insert, *Insert (MI)* that lists the missing items in the form. Of course, template genincom.htpl includes this insert.

```
proc gen_nextpage { from } {
 switch $from {
  genperson   { return "genselect"  }
  genorder    {
           if { [sessionAttribute ord_bloodproduct] == "rbc" } {
            return "genrbc"
               } elseif { [sessionAttribute ord_bloodproduct] == "wbc" } {
           return "genwbc"
               } elseif { [sessionAttribute ord_bloodproduct] == "platelets" } {
           return "genplatelet"
              } elseif { [sessionAttribute ord_bloodproduct] == "ffp" } {
           return "genplasma"
             } elseif { [sessionAttribute ord_bloodproduct] == "cryo" } {
          return "gencryo"
             }
        }
  genrbc      { return "genselect"  }
  genwbc      { return "genselect"  }
  genplatelet { return "genselect"  }
  genplasma   { return "genselect"  }
  gencryo     { return "genselect"  }
  default     { return "genselect"  }
 }
}
```

**Figure B-9: Procedure to generate next page**

```
proc gen_prevpage { from } {
 switch $from {
  genperson   { return "genperson"  }
  genorder    { return "genperson"  }
  genrbc      { return "genorder"   }
  genwbc      { return "genorder"   }
  genplatelet { return "genorder"   }
  genplasma   { return "genorder"   }
  gencryo     { return "genorder"   }
  default     { return "genselect"  }
 }
}
```

**Figure B-10: Procedure to generate previous page**

```
proc gen_close { } {
   useown Session Commit Cancel Inserts OutgoingPage
   if [gen_complete] {
    use $Session discard_on_close
    if [info exists discard_on_close] {
      unset discard_on_close
    }
    set Commit 1
    set Cancel 1
    set OutgoingPage "genclose"
    set Inserts(CL) "Your record in the database has been updated."
   } else {
    set OutgoingPage "genincom"
   }
  }
```

**Figure B-11: Procedure to terminate a session**

If the form appears complete, the function unsets *discard_on_close*, to notify the server that it shouldn't remove the database record when the session is closed. To accomplish this, the function needs a direct access to the attribute (SICLE operation use on the session handle). Then **gen_close** sets both *Commit* and *Cancel* to 1 and presents the closing form to the user. That form includes an insert labeled **CL** intended to display the final message.

Form **genincom.htpl** is not an attribute submission form. It has two graphic buttons that send special links to the server. Specifically this form template has two anchor sequences. One is the cancel button that allows filling the forms again, and the other is the logout button, which abandons the session. These anchor sequences expand into special link requests. When any of the two buttons is clicked, the resulting submission will consist of the *Display* attribute whose value is either *Sel_genforce* or *Sel_genselect*. By convention, some rules interpret such values as requests to display the form indicated by the part that follows "Sel_". Some values, however, are special. In particular, *genselect* is the name of the banner form of the PATIENT session. This form will be presented when the user hits the "Cancel" link, i.e., decides to continue filling the forms. But *genforce* is not a form name. This link is interpreted as a request to force the logout operation.

To see how this is implemented, let us have a look at the somewhat lengthy rule that takes care of such submissions (and a bit more) as shown in figure B-12.

```
rule gen_selection {
 set display [sessionAttribute Display]
 if { $display != "" } {
  set drqst [string range $display 0 2]
  if { $drqst == "Sel" } {
   set OutgoingPage [string range $display 4 end]
   switch $OutgoingPage {
    "genlogout" { gen_close }
    "genabort" { gen_abort }
    "genforce" {
     set Commit 0
     set Cancel 1
     set OutgoingPage "genclose"
     set Inserts(CL)\
          "Your file has been discarded from the database."
    }
    default {
     if { $OutgoingPage == "previouspage" } {
      set OutgoingPage\
          [gen_prevpage [sessionAttribute Status(LastPage)]]
     } elseif { $OutgoingPage == "nextpage" } {
      set OutgoingPage\
          [gen_nextpage [sessionAttribute Status(LastPage)]]
     }
     if { $OutgoingPage != "genperson" &&\
               [sessionAttribute pat_surname] == "" } {
      set OutgoingPage "genname"
      set Inserts(WA) "<font size=+2 color=\"ef0000\">You
               have to fill this form first!</font>"
     }
    }
   }
```

```
    break
   } elseif { $drqst == "Txt" } {
    if ![regexp -- {Txt_(.*):(.*)} $display junk att hdr] {
     set att [string range $display 4 end]
     set hdr ""
    }
    set display ""
    set OutgoingPage "gentxt.nfrm"
    if [inaccessible $att] {
     addWarning "You have no access to this attribute"
    } else {
     set Inserts(TI) "[sessionAttribute pat_firstname]\
                      [sessionAttribute pat_surname]"
     if { $hdr != "" } {
      append Inserts(TI) " ($hdr)"
     }
     set Inserts(TX) [tidyText [sessionAttribute $att]]
    }
    break
   }
  }
}
```

**Figure B-12: Form Submission processing**

The above rule is only executed if the temporary attribute *Display* has a value. Thus, its role is to process special link requests (at least some of them) (section 2.4.1). If the first three characters of the submitted value are "*Sel*", the rule strips the "Sel_" prefix, and stores the remaining portion of the value in *OutgoingPage*. The standard interpretation of this value is as a form identifier. One application of such links is to implement direct navigation among forms without submitting their contents.

Next, the rule determines if we have one of the special cases. One of them is *genlogout* triggered by the “Save+Logout” button on the banner form or the “Logout” link on any other form. Such a request is processed by **gen_close**, in the same way as a logout request triggered by the “Close” (submission) button.

Another special case is **genabort** triggered by the "Ignore Changes + Logout" button (the body of **gen_abort** is quite simple). The last special case, **genforce**, occurs when the user hits the "Confirm" button on the *genincom* form. In this case, the rule unconditionally terminates the session.

The default part of the switch statement is entered if the value of *Display* starts with "Sel_", but it isn't one of the three special cases discussed above. Then it represents a form to be sent back to the user. As a matter of fact, we still have to consider two special cases of such a submission. In response to *previouspage* triggered by pressing the "Previous" button on a form, the rule calls **gen_prevpage**, to find out which page is considered previous to the current one. Similarly, when the user presses the "Next" button (whose value is *Sel_nextpage*), the rule calls **gen_nextpage** to find the next page to the current one. Otherwise, the current value of *OutgoingPage* is assumed to point directly to the form to be presented next. If this form is not the patient personal data form (*genperson*) and the patient's name hasn't been entered yet, the rule forces the form to be *genperson*. Also, a warning message is inserted into the form. This way, the rule makes sure that the physician provides the patient's name before any other data.

Another type of special links handled by **gen_selection** are those whose values start with "*Txt_*". By convention, the rules assume that such a link represents a request to show the contents of a selected text-area in a separate window of the browser. Note that most text-areas that appear in form templates are accompanied by clickable buttons that can be used to display the text in a separate window. Such a button only appears if the corresponding text-area is nonempty. For example, have a look at the following fragment of *gencom.htpl*:

link_anchor_target “img src="tbull.jpg" alt=”[+]”” “Txt_tra_tsh:Comments” “Comments”

The above procedure generates a special link with value “Txt_tra_tsh:Comments”. Note that the last (optional) argument of the anchor (Comment) is used as the identifier of the target window to display the information requested by the special link.

When processing such a link, **gen_selection** strips the “Txt_” prefix and splits the remaining part into two components, as separated by the colon. The first component identifies the session attribute containing the text to be displayed (the value of the text-area field). The second component is used as the title. The rules set *OutgoingPage* to **gentxt.nfrm**. By its suffix, this page is a non-form. It includes two inserts: **WA** and **TI**. The first is the standard warning insert filled by **addWarning** if the text-area attribute is inaccessible in the current session, the other is built by the rule by combining the specified title with the contents of the textarea preprocessed by **tidyText**.

## B.4 Form template

A form template looks like a straightforward Tcl program with a sequence of procedure calls that generate different HTML components (section 2.9). The BLOOD application maintains several of these page templates that represent different forms that are presented to the application user. In this section we present some of these page templates and show how the server uses these page templates to dynamically generate the HTML code.

Let us consider the following fragment of the page template *genselect.htpl* as shown in figure B-13.

```
...
if { [sessionAttribute Status(Link)] != ""} {
 if { [sessionAttribute pat_surname] == ""} {
  plain_output "<center>"
  plain_output "<font size=+1 color=\"ef0000\">\n"
  plain_output "Welcome to the Blood Database system!"
  plain_output </font>
  plain_output </center>
  plain_output "\n <br><br> \n"

  plain_output "<p>\
You must start filling these forms from the personal form.\
Later, you may add and/or modify information in any order. \n"
  plain_output "<hr> \n"
 } else {
  plain_output "\n <p>"
  plain_output "<b><font size=+1>Audit Status: <i>"
  if { [sessionAttribute ord_bloodproduct] == "rbc" } {
  plain_output "RBC : "
  insert_attribute rbc_audit
  }
 if { [sessionAttribute ord_bloodproduct] == "wbc" } {
   plain_output "WBC : "
   insert_attribute wbc_audit
  }
  if { [sessionAttribute ord_bloodproduct] == "ffp" } {
   plain_output "FFP : "
   insert_attribute ffp_audit
  }
 if { [sessionAttribute ord_bloodproduct] == "platelet" } {
   plain_output "PLATELET : "
   insert_attribute plat_audit
  }
 if { [sessionAttribute ord_bloodproduct] == "cryo" } {
   plain_output "CRYO : "
   insert_attribute cryo_audit
  }
  plain_output "</i></font></b> \n"
 }
}
plain_output "<h3>"
plain_output "<font size=+1>"
plain_output "<b>Please select one of the following sections:"
plain_output "</b></font></h3> \n"

plain_output "<center> \n"
plain_output "<table border>"

plain_output <tr>
plain_output <td>
link_anchor "<img src=\"rbut.gif\" alt=\"Personal\">" "Sel_genperson"
plain_output </td>
plain_output <td>
link_anchor "<font size=+1>Patient's Personal Information (name,
etc.)</font>" "Sel_genperson"
plain_output </td>
plain_output "</tr>"

if { [sessionAttribute Status(SType)] != "CSERVICES"} {
plain_output <tr>
plain_output <td>
link_anchor "<img src=\"rbut.gif\" alt=\"Order\">" "Sel_genorder"
plain_output </td>
plain_output <td>
link_anchor "<font size=+1>Order Blood Product</font>" "Sel_genorder'
plain_output </td>
plain_output "</tr>"
}
...
```

**Figure B-13: Page template**

The above template represents the banner page of the PATIENT session. From the above template, we show how page templates can be coded to dynamically generate the HTML page. We see that the template checks for the session attribute *Status(Link)* which is defined for a slave session that has been linked to from another session. If this attribute is not empty, as desired, it means that the session has been linked. We then check whether the attribute *pat_surname* is empty. This check is done to ensure whether it is the first time access to the record. If so, the page generates the welcome message as directed. Otherwise, the server skips the message and executes the else condition, which shows the audit status of each the blood product.

Further down the code, we also see how we hide the option of ordering blood product to the CSERVICES-class of user. The session attribute *Status(SType)* gives the class name of the master session linked to another session. For example, if the page *genselect.htpl* (formally the banner page of PATIENT-class session) is linked from CSERVICES class, the attribute *Status(Stype)* will hold CSERVICES. Thus the page templates can include various run-time checking of attributes to dynamically present different contents to different classes of users.

We now list all the page templates for each class of users and their use in the application.

| | |
|---|---|
| genselect.htpl | : Banner page of the PATIENT session |
| genorder.htpl | : Is used to enter the preliminary order information |
| genrbc.htpl | : Is used to enter RBC transfusion events |
| genwbc.htpl | : Is used to enter Whole Blood Unit transfusion events |
| genplatelets.htpl | : Is used to enter platelet transfusion events |
| genplasma.htpl | : Is used to enter fresh frozen plasma transfusion events |

| | |
|---|---|
| gencryo.htpl | : Is used to enter Cryoprecipitate transfusion events |
| combanner.htpl | : Banner page of DATAENTRY and CSERVICES class |
| comreports.htpl | : Search page for DATAENTRY and CSERVICES class |
| comadduser.htpl | : Is used to add new patients by the DATAENTRY class |
| gencom.htpl | : Comments page for CSERVICES class |
| genincom.htpl | : Used to present incomplete information to the user |
| genclose.htpl | : Closing page of PATIENT session |
| comclose.htpl | : Closing page of DATAENTRY and CSERVICES session |
| supbanner.htpl | : Banner page for SUPERVISOR session |
| supadduser.htpl | : Used to add system users by the SUPERVISOR class |
| supdeluser.htpl | : Used to delete users by the SUPERVISOR class |
| supchpassw.htpl | : Used to change users' password by SUPERVISOR class |
| supconfirm.htpl | : Closing page of SUPERVISOR session |

In addition to the above page templates, the server also maintains the following form inserts.

| | |
|---|---|
| gentrailer.ins | : Is included at the bottom of each page template that generates navigational buttons without form submissions. |
| gensbuttons.ins | : Is included at the bottom of each page template that generates navigational buttons with form submissions. |

# Appendix C

# Installing and Running the application

## C.1 Installing the application

The entire application package comes as a gzipped tar file that unpacks into directory *src*. This directory contains two subdirectories, SICLE and the BLOOD.

The user who will own the database should perform the installation procedure. This should be neither root nor the owner of the web server (typically nobody).

The following steps need to be performed:

1. Make sure that Tcl (version 8.0 or later) is installed on your system (on all machines on which the server and the CGI driver script will be executed). Also, make sure that there is a link to the Tcl executable (tclsh) from "/usr/bin/tclsh".

2. Make sure that SICLE is installed on all machines on which the BLOOD server will be running. To install SICLE, move to directory SICLE and make sure that the **Makefile** specifies the correct path to the Tcl library (the main installation directory of Tcl). Then become root and execute:

```
~/> make install
```

Note that you must repeat this on all machines on which the server is going to run, unless the Tcl directory is shared by all those machines.

When you are done with this step, become again the user who is going to own the database.

3. Move to directory BLOOD and edit the Configuration file. Specifically, do the following (using the existing definitions for illustration):

- Set **Servers** to describe the list of BLOOD servers. This variable is a Tcl list whose entries are three-element lists. Each triplet describes one server by specifying: the Internet name of the host on which the server will be running, the port (socket) number via which the server will be visible, and the pattern used to determine the IP addresses of the clients that will be directed to that server. This pattern is a straightforward regular expression, as understood by Tcl.
- Set **DPrefix** to the full path of the server directory. This path must be globally valid for all copies of the server, regardless on what machines they run, as well as for the CGI driver script.
- Set **WebDir** to the full path of the web directory. This directory determines the URL address of the banner (login) page of the BLOOD database.
- Set **ExecPath** to the execution path for the standard programs on the server machines, specifically mkdir, ls, kill, and echo. The value of *ExecPath* will be used to set the environment variable PATH in the server for execution of those programs.
- Set **ExecHome** to the home directory of the database owner. The value of *ExecHome* will be used to set the environment variable HOME for execution of external programs.

- Set **SPassword** to the DES-encrypted and salted password of the Supervisor user. A simple way to create such a password is to perform the following (interactive) Tcl session (which will also confirm that SICLE has been installed correctly):

```
~/> tclsh
% package require siclef 1

1.0
% crypt plainpassword
..........
% exit
```

The argument of crypt should be the plaintext password of the supervisor user. Tcl will respond with an encrypted (and salted) version of this password, which then should be used as the value of *SPassword.*

- Set the constants **GeneralUser** to PATIENT class, **CLASS2** to DATAENTRY class and **CLASS3** to CSERVICES class. These represent the three different classes in the blood database system and are passed to the server program as three parameters.
- The remaining three constants, **SessionTimeout**, **MailerTimeout** and **Obsessive** can be left intact. The *SessionTimeout* specifies the default timeout for those sessions whose timeout is not specified in their templates. There are no such sessions in the present application of BLOOD. *MailerTimeout* specifies the reply timeout for the mail server used by the blood database server for sending mail. Obsessive tells whether session identifiers should include the IP addresses of the connecting clients. The default value (0) means "no". If you reset it to 1, a user who gets

disconnected from his/her provider, and then re-connects with a different IP address, will not be able to continue the interrupted session.

Execute **./Install** in the BLOOD directory.

```
~/BLOOD> ./Install
```

This script should put everything in the right place. Note that because both directories, i.e., *DPrefix* and *WebDir* must be visible in the same way by all copies of the server, you do not have to repeat this step on the other machines.

This completes the installation procedure.

## C.2 Running the application

To start the application server on a given machine, you should execute **`startup_blood`** in the server directory of the installed package. This startup script (which is actually a wrapper for BLOOD) is run as SETUID and therefore can be executed by any user. However, when the server directory is set up, its permissions are set in such a way that only the database owner is allowed to access it. Of course, root is also allowed to read and execute programs in the server directory. Therefore, you may put a call to **startup_blood** into the system startup script.

The startup operation is actually performed by the **blood** script. This script determines the name of the host on which it has been invoked, examines the full list of servers and starts the instances of the servers for the current hosts. This is simply accomplished by calling server specifying the port number as the argument. The server opens the indicated port and listens on it for connections from the driver script. Note that the server is invoked from its private instance of a simple script that does it in an infinite loop. This way, if the server dies for whatever reason, it will be immediately restarted.