5 Implementation

In the previous chapter, I described the graphical user interface that makes caching translucent to the user. In this chapter, I focus on the implementation of that interface. The first section presents an overview of the system's architecture. I continue this chapter by describing a number of implementation details. These details include descriptions of some finite state machines that control the user interface as well as an overview of the modifications required within Venus. I conclude the chapter with an overview of the Translucent Caching API. This API allows Venus to notify the CodaConsole of important events and allows the CodaConsole to make the user's wishes known to Venus.

5.1 Architecture of System

Logically, the functionality of the CodaConsole belongs within Venus (see Chapter 2) because Venus alone has knowledge of the various events needed to drive the system. However, other considerations led me to make this functionality external to Venus. The benefits of translucent caching extend beyond Coda. Moving this functionality external to Venus allows the CodaConsole to be used with any highly available file system implementing its API. A second consideration was purely practical. Venus is a large and complex body of code. Further, at the time of this research, it was under development by a number of individuals. Making my interface as independent of Venus as possible isolated me from these other development efforts. For these reasons, the CodaConsole is located outside the scope of Venus and communicates with Venus via a well-defined API.

Placing the CodaConsole external to Venus imposes additional communication overhead by adding a local RPC2 call [45]. Because daemons invoke many of these calls, the user does not directly observe this additional overhead. Those calls that are invoked in the process of servicing a user request are generally infrequent or long running; thus, the additional overhead is inconsequential. A few calls would impose too great a burden and are thus batched.

The architecture, shown in Figure 5.1, is implemented in three pieces. I refer to the first two pieces, the user interface and the Advice Monitor, jointly as the CodaConsole. They implement the graphical user interface described in the previous chapter. The user interface interacts with the user. The Advice Monitor acts as a liaison between this user interface and Venus. The third piece consists of a set of hooks within Venus that inform the CodaConsole of various events.

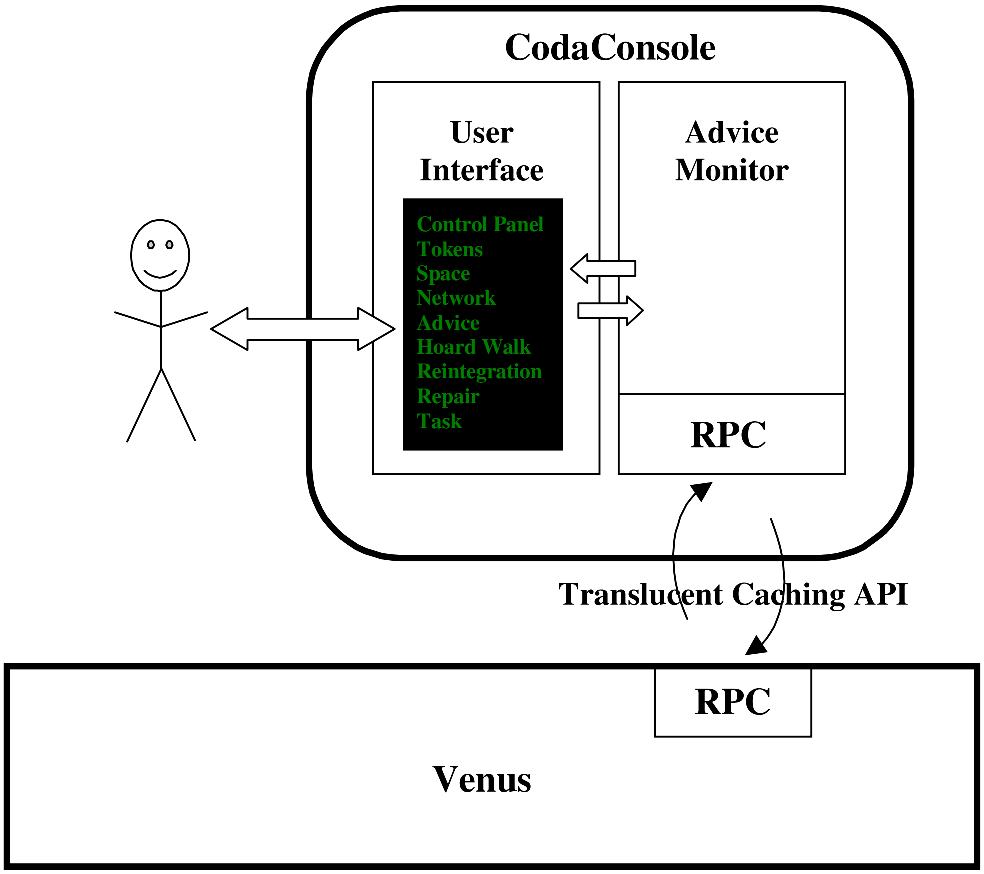

Figure 5.1: Architecture

This diagram shows the architecture of the system. The CodaConsole consists of two parts: the user interface and the Advice Monitor. The user interface implements the graphical interface described in the previous chapter. The Advice Monitor acts as a liaison between Venus and this interface. Venus notifies the Advice Monitor of various events via the RPC interface implementing the Translucent Caching API.

These three pieces communicate using two different mechanisms. The user interface communicates with the Advice Monitor via a pair of unidirectional pipes, as shown in the figure. The Advice Monitor communicates with Venus via a pair of RPC2 connections, also shown in the figure.

5.2 Details of Implementation

This section describes the implementation of each of the three components described above. I begin by describing the implementation of the user interface, focusing on the finite state machines that drive the indicator lights. I then present a brief description of the Advice Monitor. I conclude with a summary of the modifications necessary to implement the Translucent Caching API within Venus.

5.2.1 User Interface

Chapter 1 described the design of the user interface component of the CodaConsole. The user interface, which is implemented in Tcl/Tk [39, 56] using the Tix widget library [29], assumes the user is running a windowing system[14]. It is entirely event driven. As events arrive, the interface notifies the user via the indicator lights. As the user requests information, it displays the appropriate information windows. If it needs data from Venus, it contacts the Advice Monitor to request that data.

The interface presented to the user contains a set of indicator lights. These indicator lights are implemented as small, event-driven, finite state machines. The arrival of an event potentially causes a transition from one state to another. These state changes may then cause the appearance of the indicator lights to change. I will now describe three of these state machines. The remaining ones are similar in spirit, though the details differ.

5.2.1.1 Tokens Indicator

The finite state machine for the *Tokens* indicator has three states: Valid, Invalid, and Expired&Pending. The user is required to have tokens before the advice monitor begins executing so the finite state machine will start in the Valid state. When the interface receives notification that the user's tokens have expired, the machine transitions into the Invalid state. If the interface then receives a notification that an activity, a reintegration for example, is pending tokens, the machine transitions into the Expired&Pending state. The machine transitions to the Valid state from either of the other two states upon receiving notification that the user has obtained valid tokens. These states and the transitions between them are shown in Figure 5.2.

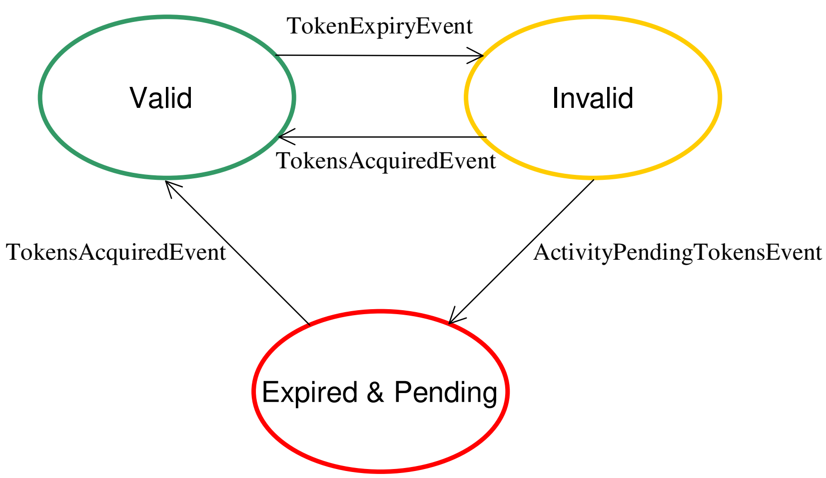

Figure 5.2: State Machine of the Tokens Indicator

[14] Currently, the system works under X windows, though it does not require that windowing system.

5.2.1.2 Hoard Walk Indicator

The finite state machine of Figure 5.3 drives the *Hoard Walk* indicator and contains four states: Inactive, Active, Stalled, and Suspend. The machine initially begins in the Inactive state. If the interface is then notified that a hoard walk has begun, the machine transitions to the Active state. Should Venus request advice, the machine transitions to the Stalled state until the user provides that advice and causes the machine to transition back to the Active state. When Venus announces the completion of the hoard walk, the machine transitions back to the Inactive state. Should the user request that Venus refrain from performing periodic hoard walks, the machine will transition to the Suspend state until periodic hoard walks are once again enabled.

5.2.1.3 Task Indicator

A smaller but slightly more complex pushdown automaton drives the Task indicator. This machine, which is shown in Figure 5.4, contains just two states (All Tasks Available and At Least One Task Unavailable), a counter, and a list of available tasks. At the end of each hoard walk, Venus announces the availability of each hoarded task owned by the user. This availability information is presented to the user in the *Task Information* window.

Suppose that Venus just completed a hoard walk and announced that all hoarded tasks were available. Further suppose that a hoarded file is invalidated (because, for instance, it was updated remotely). Venus would then announce that an object has become unavailable. In this message, it would also specify the task and the size of the object. The interface would now transition to the At Least One Task Unavailable state, set the counter of unavailable tasks to 1, remove this task from the list of available tasks, update the information about this task, and, finally, notify the user of this event. Now suppose that Venus must replace a hoarded file (for instance, due to space limitations). When this event occurs, Venus announces that the object has become unavailable as before. The interface, however, must be more careful in responding to this event. It must first determine whether or not this task was already unavailable. If the task was previously unavailable, then the system simply adjusts the percentage of the task available based upon the new information. If the task was previously available, then the system increments the counter of unavailable tasks, removes this task from the list of available tasks, and adjusts the percentage of the task available. Because the interface is already in the At Least One Task Unavailable state, it does not take a transition.

5.2.2 Advice Monitor

The primary purpose of the Advice Monitor is to act as a liaison between the user interface and Venus. As events arrive from Venus, the Advice Monitor translates the request into an appropriate procedure call within the interface, and then ships that call to

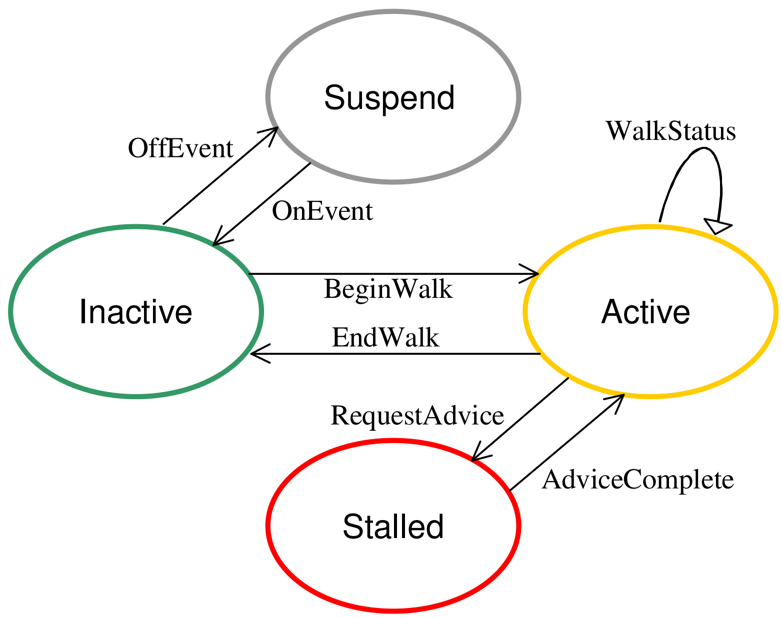

Figure 5.3: State Machine of the Hoard Walk Indicator

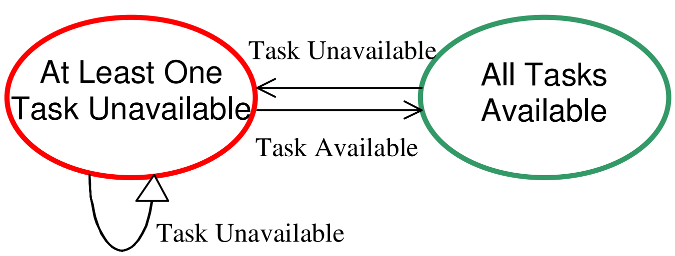

Figure 5.4: State Machine of the Task Indicator

the interface. As requests arrive from the interface, the Advice Monitor translates the request into an appropriate RPC2 call.

The Advice Monitor also has a number of secondary responsibilities. These include moving data files (the results of small questionnaires) into the Coda file system, processing certain usage statistics provided by Venus for the purpose of offering advice (as discussed in Sections 4.6.2 and 6.2), and controlling the Application Specific Resolvers, as described in [28].

The Advice Monitor uses a total of eight light-weight processes [45], or threads, to fulfill its obligations. Each thread and its primary responsibilities are described below:

Main: Performs initialization, loops waiting for incoming RPC requests and incoming messages from the user interface, and also dispatches other threads

CodaConsoleHandler: Handles messages arriving from the user interface, making the appropriate RPC to Venus (if required)

WorkerHandler: Handles RPCs arriving from Venus

ProgramLogHandler: Analyzes incoming program logs

ReplacementLogHandler: Analyzes incoming replacement logs, adding entries to the database

DataHandler: Moves data from temporary locations into permanent storage in Coda file system

EndASREventHandler: Watches for completion of Application-Specific Resolvers (ASRs) and returns the results to Venus

ShutdownHandler: Handles shutdown requests, closing connections and exiting the user interface

Two of these threads warrant further attention and will be discussed in the following chapter. These two threads, the Program Log Handler and the Replacement Log Handler, help to reduce the burden hoarding places on users.

5.2.3 Venus Modifications

Supporting the CodaConsole required modifications to Venus in two areas. First, I added the advice module, which implements the Venus side of the Translucent Caching API. The advice module exposes certain Venus *data* to the user and provides wrappers around the calls of the API. Second, I added hooks throughout Venus that invoke the wrappers within the advice module. These hooks expose certain *events* to the user by notifying the CodaConsole. This section describes both modifications.

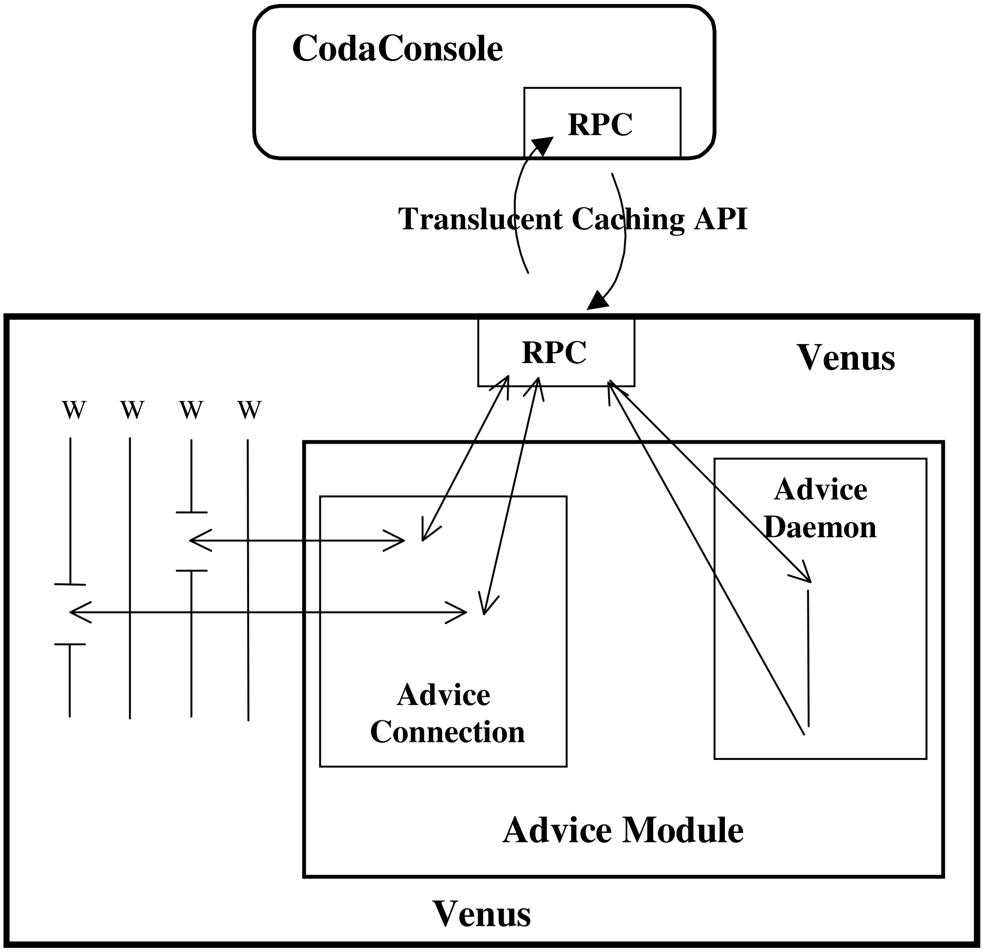

Figure 5.5: Relevant Venus Internals
This diagram shows a detailed view of the Venus component of Figure 5.1. The details include only those aspects of Venus relevant to supporting the CodaConsole.

5.2.3.1 The Advice Module

The advice module contains two pieces: the Advice Connection and the Advice Daemon. These two pieces, which are shown in Figure 5.5, handle the communication to and from the Advice Monitor respectively. The Advice Connection maintains a connection to a user's Advice Monitor and handles requests from Venus intended for that user. The Advice Daemon handles requests arriving from the user interface via the Advice Monitor.

5.2.3.1.1 The Advice Connection

Venus maintains at most one advice connection, the adviceconn class, per user. The connection is made on a per-user basis for security reasons—information flow to the interface includes pathnames of referenced and hoarded files, information that would be inappropriate to release to other users.

Venus maintains at most one connection for each user simply to avoid confusion. It supports multiple simultaneous advice connections by different users with two caveats. First, other aspects of Venus (in particular, disconnected and weakly connected operation) assume a single-user workstation. Second, certain activities are restricted to authorized users, as defined by the user currently logged in on the console or the user specified (at startup) to Venus.

Venus requires that the connection be made from the local host. This decision avoids any security concerns associated with remote control of Venus, improves performance because the RPCs between Venus and the interface are now local, and avoids the possibility of the Advice Monitor becoming disconnected from Venus. Finally, it is consistent with the philosophy that the Advice Monitor is logically a part of Venus.

An advice connection maintains a small amount of state, a subset of which is shown in Figure 5.6 and described below.

User Lock: Controls communication with the interface and, thus, the user

Connection Information: Allows the Advice Daemon to establish a connection from Venus to the Advice Monitor, includes the state, hostname, port, and handle

Process Group ID: Prevents deadlock in the case that a request made on behalf of the user generates a request back to the user

Interest Array: Allows Venus to suppress event notifications of which the CodaConsole is not configured to notify the user, an optimization that potentially removes a local RPC from the critical path of the call.

Logging State: Allows Venus to log each file access and the program that made it, as well as each cache replacement, includes file descriptors, pathnames, and line counts

Methods: Provides a set of methods that are, essentially, wrappers around the API; provides a clean interface to the rest of Venus; and ensures that all locking and counting occur as expected

The methods provided by the advice connection wrap around half of the Translucent Caching API, the half that is serviced by the Advice Monitor. The calls defined by this API, which are summarized in Figure 5.7 below and described in detail in Appendix A, expose certain events to the Advice Monitor, and thus the user.

The Translucent Caching API, which includes all of the calls shown in Figure 5.7 as well as those shown later in Figure 5.8, is key to the CodaConsole's independence from Venus, and by extension from Coda. Any distributed system that offers high availability should be able to take advantage of the CodaConsole simply by supporting this API.[15]

[15] Of course, not all distributed file systems offering high availability will necessarily support all of the conceptual ideas of Coda. Any conceptual differences would obviously need to be handled separately.

```
class adviceconn {
    // State related to controlling access to the user
    struct Lock userLock;

    // State related to the connection to the Advice Monitor (A.M.)
    AdviceState state;             // of this connection
    char hostname[MAXHOSTNAMELEN]; // on which A.M. is running
    unsigned short port;           // on which A.M. is listening
    RPC2_Handle handle;            // to active RPC connection

    int pgid;                      // of the Advice Monitor

    // State related to events in which the user is interested
    int InterestArray[MAXEVENTS];

    // State related to logging various events
    FILE *programLog, *replacementLog;
    char programLogName[MAXPATHLEN], replaceLogName[MAXPATHLEN];
    int numProgLines, numReplaceLines;

    .
    .
    .

  public:
    // Methods
    void TokensAcquired(int);
    void TokensExpired();
    void ServerAccessible(char *);
    void ServerInaccessible(char *);

    .
    .
    .
};
```

Figure 5.6: State Maintained by the `adviceconn` Class

This figure shows a subset of the state maintained by the `adviceconn` class. This state includes a lock that controls access to the user, a group of variables that maintain information regarding the current connection to this user's Advice Monitor, an array that records the user's interest in each event, and a group of variables that support logging various events. The complete definition of this class is available in the Coda sources in the file `coda-src/venus/adviceconn.h`.

Remote Procedure Call	Description
LostConnection	Announces that the Advice Monitor has lost its connection
TokensAcquired	Announces that the user has acquired tokens
TokensExpired	Announces that the user's tokens have expired
ServerAccessible	Announces that a server is accessible
ServerInaccessible	Announces that a server is no longer accessible
ServerConnectionWeak	Announces that the connection to a server is weak
ServerConnectionStrong	Announces that the connection to a server is strong
NetworkQualityEstimate	Announces the current server bandwidth estimates
Reconnection	Requests that a reconnection questionnaire be presented to the user
ReadDisconnectedCacheMiss	Requests advice regarding a read disconnected cache miss
WeaklyConnectedCacheMiss	Requests advice regarding a weakly connected cache miss
DisconnectedCacheMiss	Requests that a disconnected cache miss questionnaire be presented to the user
HoardWalkAdviceRequest	Requests advice regarding a hoard walk
HoardWalkBegin	Announces that a hoard walk has begun
HoardWalkStatus	Announces the progress of a hoard walk
HoardWalkEnd	Announces that a hoard walk has ended
HoardWalkPeriodicOn	Announces that periodic hoard walks have been enabled
HoardWalkPeriodicOff	Announces that periodic hoard walks have been disabled
ObjectInConflict	Announces that a file system object is in conflict
ObjectConsistent	Announces that a file system object has been repaired
ReintegrationPendingTokens	Announces that a reintegration is pending, waiting for the user to obtain authentication tokens
ReintegrationEnabled	Announces that a reintegration is ready to proceed
ReintegrationActive	Announces that a reintegration is in progress
ReintegrationCompleted	Announces that a reintegration has completed
TaskAvailability	Announces the availability of hoarded tasks
TaskUnavailability	Announces the unavailability of an element of a hoarded task
ProgramAccessLogAvailable	Announces the availability of a program access log
ReplacementLogAvailable	Announces the availability of a replacement log
InvokeASR	Invokes application-specific resolution

Figure 5.7: Calls of the API Serviced by the Advice Monitor

Remote Procedure Call	Description
NewAdviceService	Announces the existence of a new Advice Monitor
RegisterInterest	Records user's interest in various events
GetCacheStatistics	Retrieves current space information
OutputUsageStatistics	Outputs disconnected usage statistics
SetParameters	Modifies value of internal variables
ResultOfASR	Provides result of application-specific resolver
ImminentDeath	Announces imminent death of Advice Monitor

Figure 5.8: Calls of the API Serviced by Venus

Another interesting issue to consider is what would be required to move Venus into a kernel module, as was done with production versions of AFS. Currently, Venus uses RPC2 to support the Translucent Caching API, but this choice is not critical to its success. Should Coda ever be moved down into the kernel to improve performance, this RPC2 interface can easily be replaced by system calls, leaving the CodaConsole to run in user space where it clearly belongs.

5.2.3.1.2 The Advice Daemon

The implementation of the Advice Daemon is straightforward. The main routine of this Venus process loops waiting for a request to arrive and then handles that request. The daemon is responsible for implementing the seven calls of the Translucent Caching API serviced by Venus. These calls are summarized in Figure 5.8 and described in detail in Appendix A. They expose certain data to the Advice Monitor.

5.2.3.2 Venus Hooks

Venus informs the user of events that occur during normal operation. These events include token expiration, hoard walks, cache misses, and task availability. For each event, I added a relatively straightforward hook. The hook first identifies which user (or users) to notify. It then makes the appropriate call to that user's Advice Connection, including all required information. Each hook generally requires only tens of lines of code to implement. The challenge in adding this functionality lay largely in tracking down where in the code to add the appropriate calls to the API. Occasionally, collecting the necessary information required a minimal amount of additional work.

Hooks were added to the routines shown in Figure 5.9 below. This table serves two purposes. The first is to document the location of these hooks for future Venus developers. The second, and more important one, is to show that these hooks correspond roughly one-to-one to the API calls of Figure 5.7. Only a few API calls require more than one hook and none required more than three hooks. Generally, all of the hooks appear in the obvious location (to someone familiar with Venus internals).

Event	Class::Method
LostConnection	adviceconn::NewConnection
TokensAcquired	userent::SetTokens
TokenExpiry	userent::Invalidate
ServerAccessible	srvent::ServerUp
ServerInaccessible	srvent::ServerError, srvent::Connect
ServerConnectionStrong	srvent::InitBandwidth, srvent::GetBandwidth
ServerConnectionWeak	srvent::InitBandwidth, srvent::GetBandwidth
NetworkQualityEstimate	srvent::InitBandwidth, srvent::GetBandwidth
Reconnection	volent::TakeTransition
ReadDisconnectedCacheMiss	fsdb::Get
WeaklyConnectedCacheMiss	fsdb::Get
DisconnectedCacheMiss	fsdb::Get
HoardWalkAdviceRequest	hdb::Walk
HoardWalkBegin	hdb::Walk
HoardWalkStatus	hdb::StatusWalk, hdb::DataWalk
HoardWalkEnd	hdb::Walk
HoardWalkPeriodicOn	hdb::Enable
HoardWalkPeriodicOff	hdb::Disable
ObjectInConflict	fsdb::Get
ObjectConsistent	lrdb::CheckLocalSubtree, vproc::do_ioctl
ReintegrationPendingTokens	volent::TakeTransition
ReintegrationEnabled	volent::TakeTransition
ReintegrationActive	volent::Reintegrate
ReintegrationCompleted	volent::Reintegrate
TaskAvailability	hdb::PostWalkStatus
TaskUnavailability	fsobj::Kill
ProgramAccessLogAvailable	::VTALRM, fsobj::Get
ReplacementLogAvailable	::VTALRM, fsobj::ReclaimFsos, fsobj::ReclaimBlocks
InvokeASR	fsdb::Get

Figure 5.9: Location of Hooks Supporting Translucent Caching

This table shows the location of hooks that support translucent caching. For each call in the Translucent Caching API, I show each method of Venus that calls it. Note the nearly one-to-one mapping between these hooks and the routines provided by the API (refer to Figure 5.7).

5.3 Use of Translucent Caching API

In this section, I will make the API more concrete by exploring three scenarios. The first one illustrates the initial sequence of messages that startup the CodaConsole. The second one illustrates a basic event notification, in this case a notification that the user's tokens have expired. The third one illustrates the conversation that occurs between Venus and the CodaConsole during a hoard walk.

5.3.1.1 Initialization

Venus begins running during the Unix startup sequence (e.g., out of /etc/rc). Some time later, the user logs in, starts X, and eventually starts the CodaConsole. Before Venus and the CodaConsole, two independent processes, can cooperate to make caching translucent to the user, communication between them must be established. Further, because the CodaConsole also consists of two cooperating processes, communication must be initialized for them as well. As described previously and illustrated in Figure 5.1, communication between Venus and the CodaConsole occurs via a pair of RPC connections while communication between the two components of the CodaConsole occur over a pair of pipes. In this section, I describe how this initialization, which is illustrated in Figure 5.10, occurs.

Because Venus starts long before the CodaConsole, it is the CodaConsole's responsibility to initiate contact with Venus. The main thread of the Advice Monitor initiates contact with Venus during its initialization phase. It issues the NewAdviceServiceAvailable RPC2 call, shown in Figure 5.8, to Venus. Parameters to this call include connection information, version information, and the process group ID of the Advice Monitor. As described previously, the process group ID is used to prevent deadlock (see Section 5.2.3.1.1). The version information is used to detect version skew between the Advice Monitor and Venus, and thus ensure that they are using the same API. The connection information includes the (ephemeral) port on which the Advice Monitor will be listening for the return call from Venus.

Upon receiving this call, Venus initializes the adviceconn for the user based on the incoming parameters. Venus also sets the state of this connection to indicate that it is waiting. Venus then returns from the RPC and signals the initialUserSync variable. At this point, we have unidirectional communication between Venus and the Advice Monitor. The Advice Monitor can issue RPCs to Venus, but Venus cannot yet issue RPCs to the Advice Monitor.

Two more processes must still complete before our initialization is finished. The user interface must be started and Venus must return the connection to complete the bidirectional communication between it and the Advice Monitor. Although either of these things could occur next, I will assume that the interface is started first and that Venus' return connection occurs second.

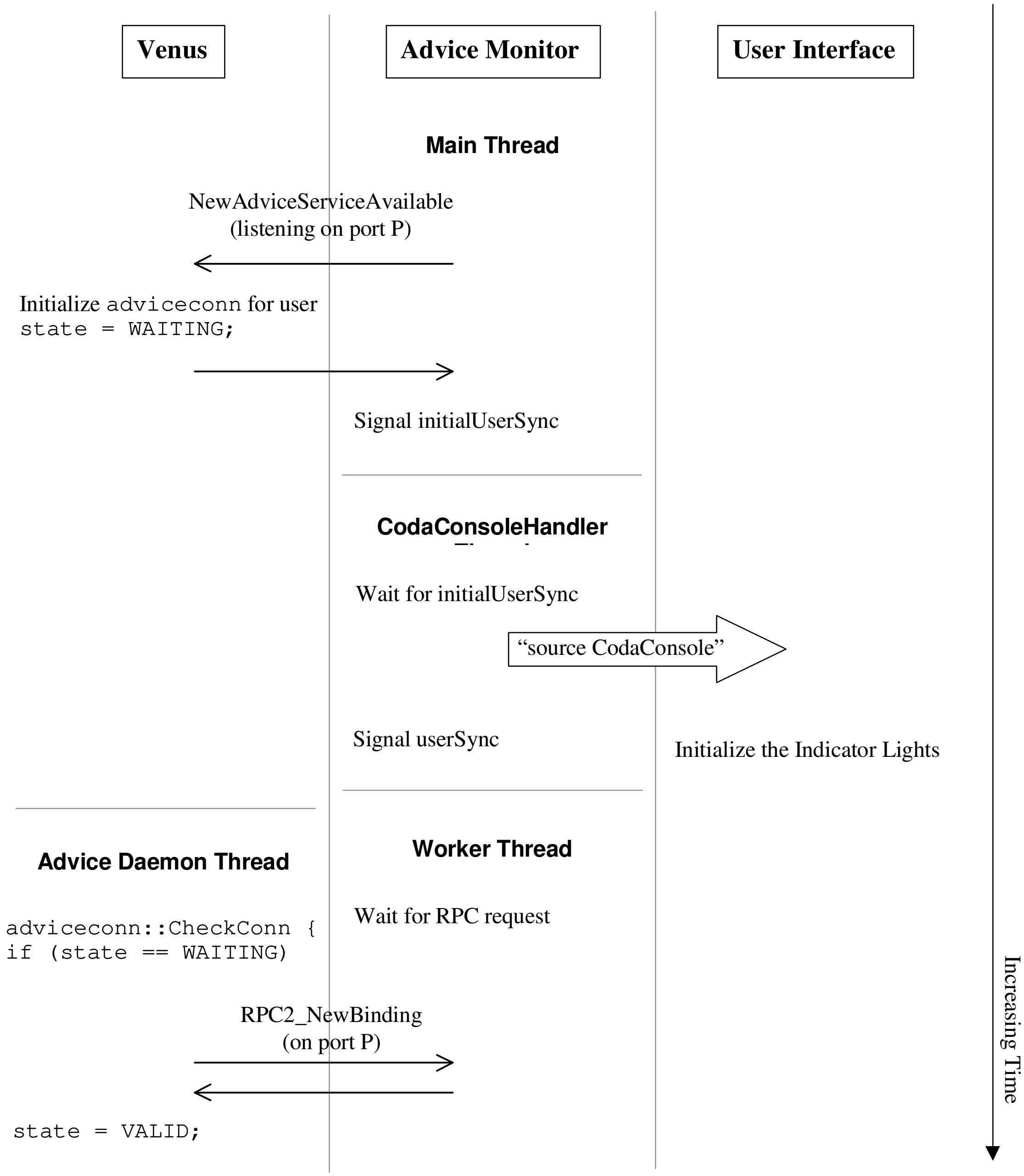

Figure 5.10: Initialization

This figure illustrates the initialization process. Each process is shown in its own column. Interprocess communication is shown as arrows crossing the columns. The thin arrows represent remote procedure calls. The thick arrow represents a pipe.

Signaling the initialUserSync variable allows the CodaConsoleHandler thread to wake up. This thread is responsible for starting the user interface and, later, for processing input received from its pipe[16]. After waiting for this signal, the CodaConsoleHandler issues the command that starts the user interface. It then enters a loop that waits for input from the user interface and processes that input. The user interface is now active.

The final step in the initialization procedure is to establish the connection from Venus to the Advice Monitor. Eventually, the Advice Daemon (a lightweight thread within Venus) discovers that an adviceconn is in the waiting state. It will then issue a RPC2_NewBinding request to the Advice Monitor on the port specified in the original NewAdviceServiceAvailable message. Once this call returns, the state of the connection is set to Valid and Venus can now issue RPCs to the Advice Monitor.

The initialization of communication between all three processes is now complete. Venus can issue event notifications to the interface. The interface can issue requests for information. As important cache-related events occur, the system will notify the user. Caching is now translucent.

5.3.1.2 Basic Event Notification

The majority of calls from Venus to the CodaConsole interface via the Advice Monitor are basic event notifications. In this section, I will walk through the messages involved in performing such a notification. I will use token expiry as my example. Figure 5.11 shows an illustration of this notification.

During token expiry, Venus calls the TokenExpiry RPC2 wrapper. This wrapper checks that the user is interested in receiving notification of this event. If so, it locks the userLock stored in the user's adviceconn structure. It then issues the TokenExpiry RPC. This call is sent to the Advice Monitor. The Advice Monitor translates this call and outputs a message, "TokenExpiryEvent", to the CodaConsole interface via the appropriate pipe. After flushing the pipe, it returns from the RPC2 call. The wrapper routine then releases its lock, checks for errors, and returns to Venus.

Upon receiving the "TokenExpiryEvent" from its input pipe, the user interface first looks up the current configuration of this event. Based upon this configuration, it then notifies the user that this event has occurred. By default, this notification will turn the *Tokens* indicator light yellow.

---

[16] The pair of pipes, which allow the Advice Monitor and the CodaConsole to communicate, were initialized very early in the startup process. Before the user interface process was forked, two pipes were created. The Advice Monitor maintained file descriptors for one end (the read side of one and the write side of the other) of each pipe and mapped standard input and output for the user interface to the other ends of these pipes. One pipe allows the Advice Monitor to send messages to the user interface. The other allows the user interface to send messages to the Advice Monitor.

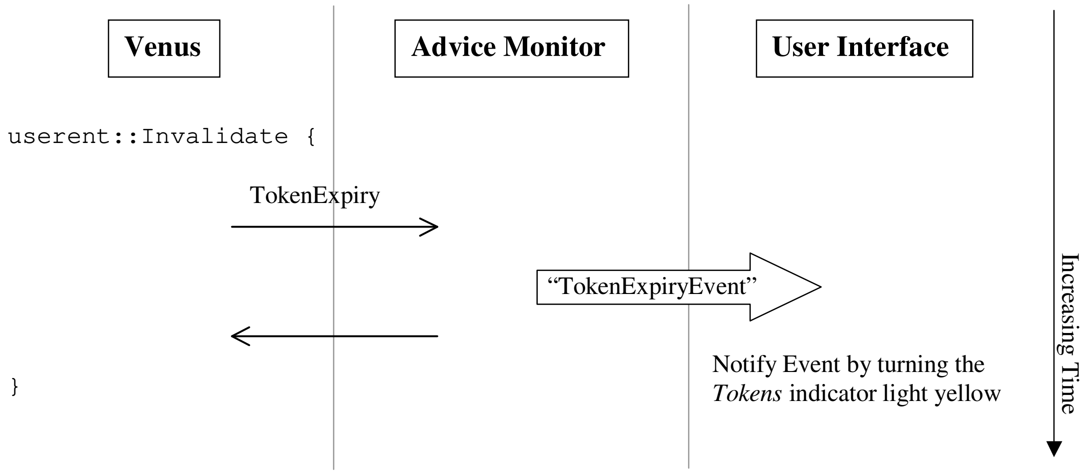

Figure 5.11: Notification of Token Expiration

This figure illustrates a basic event notification. The event shown is the token expiry event. Each process is shown in its own column. Interprocess communication is shown as arrows crossing the columns. The thin arrows represent remote procedure calls. The thick arrow represents a pipe. Return from the RPC occurs immediately after flushing the pipe.

5.3.1.3 Traffic During a Hoard Walk

One of the more complicated interactions occurs as Venus performs a Hoard Walk (see Section 2.2.2). In this section, I describe the many different messages that go back and forth between Venus and the CodaConsole. This conversation is illustrated in Figure 5.12. For the purposes of this example, I assume that Venus requests advice from the user.

When Venus first begins a hoard walk, it notifies the CodaConsole. Venus then performs a status walk, during which it equilibrates the cache fetching all necessary status blocks. After completing the status walk, it informs the CodaConsole as to its progress. Then, it generates a file containing the information for which it would like advice from the user and initiates the request for advice. To request advice, it triggers an RPC to the Advice Monitor, the HoardWalkAdviceRequest call. In this call, it includes the name of the input file it previously generated and the name of the file in which it expects to find the user's advice. When the Advice Monitor receives this RPC, it forwards the request to the CodaConsole and then waits on the hwSync variable.

The user interface notifies the user of the request as specified by the configuration for this event (see Section 4.2.1). It then waits for the user to provide advice. Once the user has done so, it sends a message to the Advice Monitor informing it that advice is now available. This causes the CodaConsoleHandler thread to signal the hwSync variable.

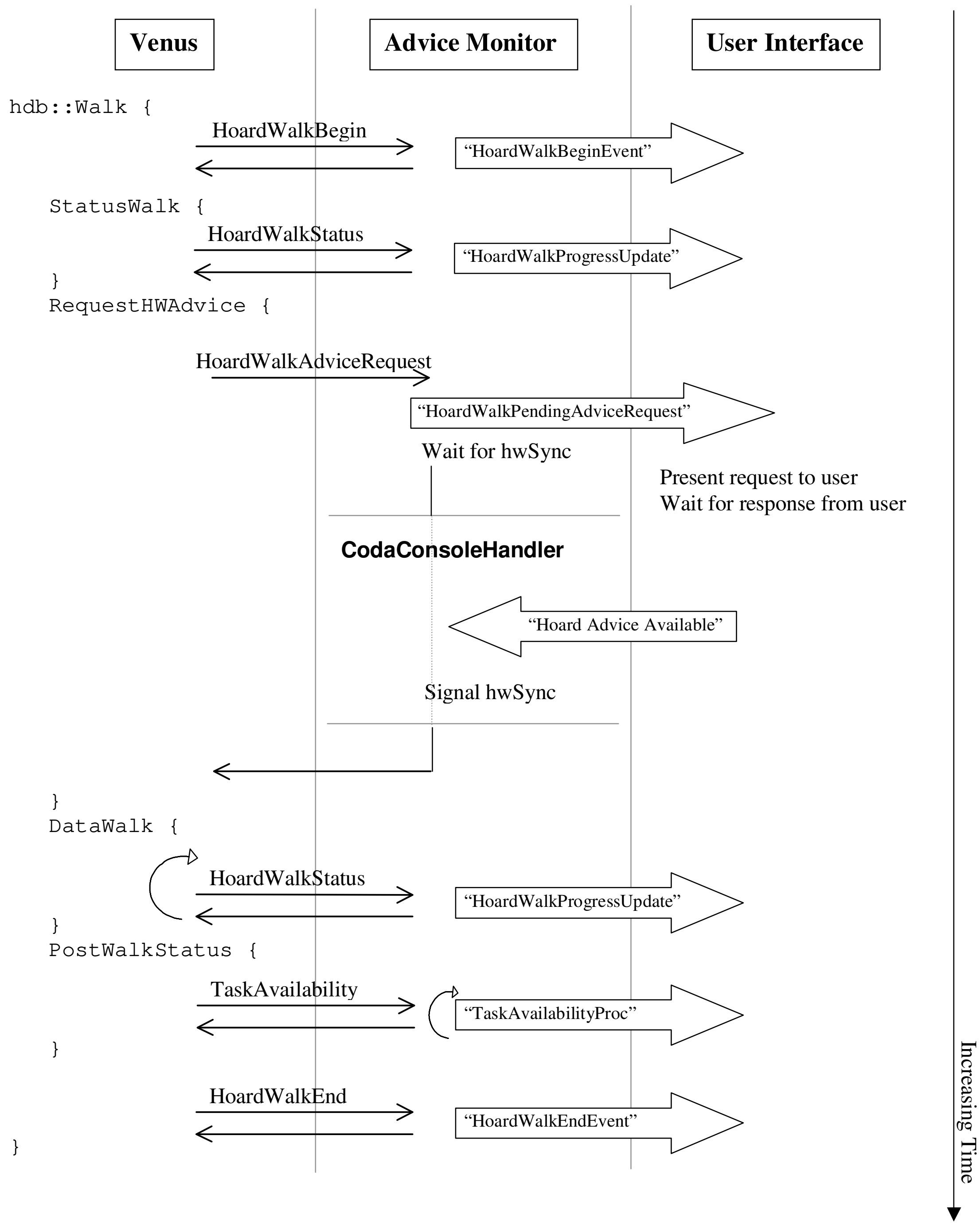

Figure 5.12: Traffic During a Hoard Walk
This figure illustrates traffic that occurs during the process of a hoard walk. Each process is shown in its own column. Interprocess communication is shown as arrows crossing the columns. The thin arrows represent remote procedure calls. The thick arrows represent pipes.

The HoardWalkAdviceRequest RPC then returns to Venus with the resulting advice. Venus is finally able to proceed with the data walk during which it fetches any data that it needs. During this data walk, Venus informs the CodaConsole of its progress using the HoardWalkStatus RPC. This call, another basic event notification, drives the progress meter shown in the *Hoard Walk Information* window (see Figure 4.12).

Upon completing the data walk, Venus has two additional responsibilities. First, it determines the availability of all hoarded tasks. It sends this information to the CodaConsole in the TaskAvailability RPC. Using this information, the Advice Monitor issues messages, one per task, to inform the user interface of the availability of each task. Venus then notifies the CodaConsole that it has completed the hoard walk.

This completes the process of a hoard walk. As you can see, many different messages can be triggered during a single hoard walk. The total number of messages possible is bounded at 104. Four notify the CodaConsole of the beginning of the walk, of the advice request, of the task availability, and of the end of the walk. The 100 additional messages represent the numerous progress updates. One hundred represents an upper bound for this number because progress messages are only sent when the percentage of the hoard walk completed increases by at least 1.[17]

5.4 Summary

This chapter has described the implementation of the CodaConsole. This implementation included the user interface itself, the Advice Monitor, and a number of modifications to Venus. The implementation uses the Translucent Caching API to support communication between Venus and the CodaConsole.

At the beginning of this chapter, I discussed my decision to place the functionality of the user interface external to Venus as much as possible. It isolated me from concurrent Venus development activities, thus allowing a more stable development environment for this research! More importantly, however, it made the interface independent of Venus, and therefore of Coda. Other file systems offering high availability could easily adopt the CodaConsole simply by implementing the API. Overall, I was able to keep Venus modifications to a minimum. I added just 2500 lines of code, making Venus approximately 3% larger than it would otherwise be. Had I implemented the functionality completely within Venus I would have added another 12,500 lines, increasing the size of Venus by more than 15%. Hindsight suggests that this was a wise decision.

[17] The original design informed the user interface of the current progress after the completion of each individual fetch. However, because the system may fetch numerous small files, the number of RPC messages quickly became unreasonable. The revised design limits the total number of messages sent to equal the maximum number the user can distinguish; any messages beyond that represent wasted effort.