UNIVERSITY OF CALIFORNIA SAN DIEGO

An Introduction to Rich Services/Erlang

A Thesis submitted in partial satisfaction of the requirements for the degree of Master of Science

in

Computer Science

by

Tyler Elias Netherland

2009

The Thesis of Tyler Elias Netherland is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2009

Table of Contents

List of Figures

List of Tables

Acknowledgements

My complete appreciation and gratitude go out to Dr. Ingolf Krueger for giving me the opportunity to participate in research under his supervision. In addition I need to thank Massimiliano Menarini and Filippo Seracini for their advice throughout my endeavor to produce this thesis. I would also like to mention the support of my committee members, Bill Howden and Ryan Kastner. I would like to thank the United States Navy for giving me the opportunity to attend UCSD and earn a master's degree. I think that it is also important for me to publically recognize God for the blessings that I have received through the opportunities and people I have experienced and met while studying at UCSD.

ABSTRACT OF THE THESIS

An Introduction to Rich Services/Erlang

by

Tyler Elias Netherland

Master of Science in Computer Science

University of California, San Diego 2009

Professor Ingolf H. Krueger, Chair

Rich Services addresses the challenges of building and integrating distributed software systems. These include the need for managing multiple stake holder concerns, for integrating distributed software systems as a single entity into higher level distributed software systems behind interfaces, for adapting to changing requirements and for providing scalability. There is a need for software libraries that support the development of software systems designed using the Rich Services architecture.

Rich Services/Erlang is the first software library that supports the creation of distributed software systems designed using the Rich Services architecture. Erlang is a functional programming language which explicitly supports the development of distributed, concurrent software. The library leverages the features of Erlang and the Rich Services architecture to

empower developers to focus on the design and application logic of their software systems, rather than the implementation complexity of the integration and messaging system.

We begin with an introduction of the challenges encountered in the creation of distributed software systems and with a discussion of the need for the Rich Services architecture. We continue with an overview of Erlang and then the introduction to Rich Services/Erlang. The following chapter includes a description of the process used to develop Rich Services software systems using the library. The final chapter about Rich Services/Erlang discusses the run-time view of systems implemented with it. The next few chapters present our enterprise integration patterns and chat system case studies. The thesis finishes with our evaluation and our conclusion.

Chapter I Introduction

The Rich Services architecture supports the creation of concurrent, distributed software systems and addresses the various challenges facing systems-of-systems integration projects [2,4,5,6]. At the outset of the research presented in this thesis there was not a library that specifically supported the development of Rich Services software systems.

One can make a simple statement about the main contribution of the research presented in this paper: the author created the first library that supports implementations of software systems designed using the Rich Services architecture. The library is implemented using the functional programming language Erlang, chosen for its support of concurrent, distributed software systems.

Furthermore our research introduces a few modifications to the Rich Services architecture (such as support for multiple Routers per Messenger). In addition to introducing Rich Services/Erlang, this paper uses two case studies to discuss the flexibility and feasibility of systems designed and created using Rich Services/Erlang. Finally an evaluation of Rich Services/Erlang, Rich Services and Erlang is provided for the reader.

In sum this paper presents the contributions made by the author to the Rich Services architecture. Chapter II introduces the Rich Services

Architecture while Chapter III introduces Erlang. Chapter IV introduces core features of Rich Services/Erlang. Chapter V discusses how designers take a system’s requirements to design a system and then how to implement that system using Rich Services/Erlang. Chapter VI reveals the run-time implementation details of Rich Services implemented using Rich Services/Erlang. Chapter VII presents a case study that discusses the use of various enterprise integration patterns with the Rich Services/Erlang library while Chapter VIII introduces a chat system case study built using the library. Chapter IX provides an evaluation for the topics discussed in this paper. Finally Chapter X leaves the reader with a conclusion for this paper and a brief discussion of future work.

Chapter II An Introduction to Rich Services

Systems-of-systems integration poses many engineering challenges. Solutions must provide interfaces for distributed subsystems, address the need for scalability, accommodate the concerns of multiple stakeholders, allow for the integration of legacy systems and easily adapt to changing requirements over time [5,6]. Service oriented architectures (SOA) try to address at least some of these challenges.

Web services (and SOAs in general) enable systems-of-systems integration but do not address some challenges adequately. Specifically, cross cutting concerns (policy management, governance, and authentication) frustrate the integration task, forcing solutions that are not true to the light weight nature of web services [2,4,5,6]. They are dealt with using fragmented standards that make it difficult to re-integrate the resulting services [5].

Rich Services is a service oriented architecture that addresses the concerns present in horizontal and vertical service integration and that provides a direct mapping between a system's logical architecture and its implementation [5,6]. Horizontal integration refers to the management of services in the same logical deployment level. Vertical integration refers to the hierarchical decomposition of one service into a set of sub-services.

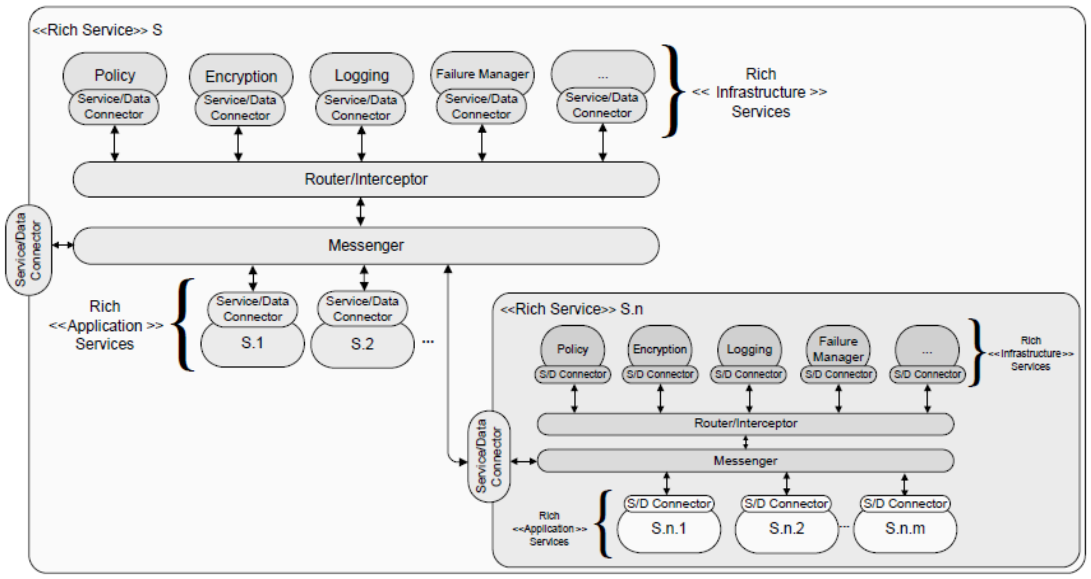


Figure 1: **Rich Service Architecture Diagram**

Figure 1 shows the Rich Services architecture. The main components of the architecture are the Messenger/Router, Service Data Connector (SDC) and Rich Service [2]. A Rich Service exports an interface through a SDC [6]. Rich Application Services (RAS) connect to the Messenger, and Rich Infrastructure Services (RIS) connect to the router. The Messenger/Router provide a core message-based communication infrastructure to connect a set of Rich Application Services.

A Rich Service can be simple, implementing a feature or functionality, or it can be composite, implementing a feature through a subset of internal services. These are exported through a SDC interface. Any composite Rich Service could have its own SDC to support its integration into a higher level Rich Service. Therefore vertical integration is supported by designing the architecture in such a way that it is replicated across hierarchical layers [2].

The communication infrastructure is derived from an enterprise service bus (ESB) messaging strategy. ESBs have three advantages. They benefit from the strengths of message-oriented middle ware, from the flexibility of plug-in architectures and from the convenience of data adaptors/connectors for integrating data sources, applications and services [2]. Furthermore they support the use of multiple routing strategies (such as dynamic or static routing).

The Messenger layer transmits messages between endpoints using asynchronous message passing and provides the decoupling of services [2,6]. The Router intercepts messages in the Messenger to facilitate dynamic policy injection [2]. It is responsible for routing messages to the appropriate destination(s) [6].

SDCs connect Rich Services to the communication infrastructure and encapsulate their internal structures [2,6]. The communication infrastructure only requires knowledge of a Rich Service's SDC. Rich Services therefore can hide the architecture pattern behind the SDC to support vertical integration. Designers could attach an SDC to any Rich Service [2].

The Rich Service is the main entity of the architecture [2,6]. Rich Application Services provide core application services and communicate directly with the messenger. Rich Infrastructure Services provide infrastructure services and communicate directly with the router [6].

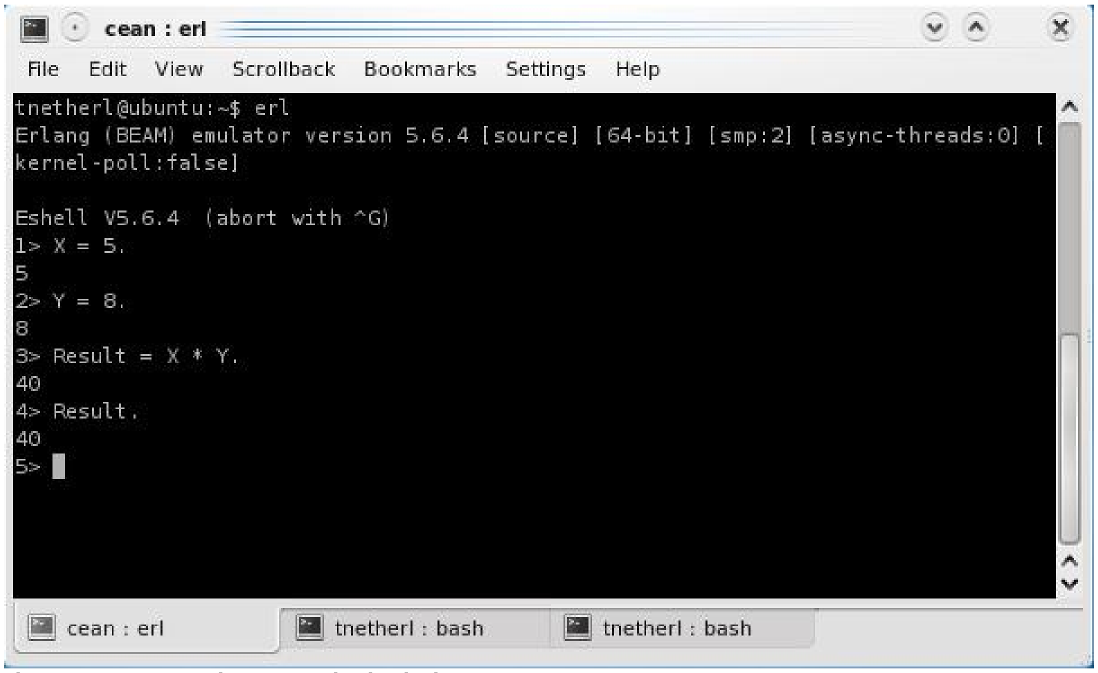
Chapter III An Introduction to Erlang

The purpose of the research presented in this paper was to use and evaluate Erlang as a potential implementation language for a Rich Services infrastructure. The result is the Rich Services/Erlang library. This chapter focuses on key Erlang concepts to provide the reader with a basic understanding of Erlang.

Erlang is a functional programming language that supports the creation of concurrent, distributed software. According to Joe Armstrong, Erlang is useful for the following reasons [8]. It supports the creation of concurrent, distributed software more efficiently than other languages, provides fault-tolerance primitives, allows dynamic code loading and boasts successful deployment in commercial grade software systems. In addition, Erlang programs require less code than the same programs written in other languages.

Erlang Terminal Window

Programmers can execute Erlang functions by typing commands into an Erlang terminal [1]. Figure 2 is an image of a terminal window. The top line shows how to open the shell in linux using the erl command. The assignment statements demonstrate how commands are entered in the shell.



```
tnetherl@ubuntu:~$ erl
Erlang (BEAM) emulator version 5.6.4 [source] [64-bit] [smp:2] [async-threads:0] [
kernel-poll:false]

Eshell V5.6.4  (abort with ^G)
1> X = 5.
5
2> Y = 8.
8
3> Result = X * Y.
40
4> Result.
40
5>
```

Figure 2: **Erlang Terminal Window**

Expressions

Erlang expressions include many different concepts including terms, variables and patterns [9].

Terms

Erlang terms describe any Erlang data type [10]. These include atoms, process identifiers (Pid), tuples, lists, strings and fun expressions.

Atom

Erlang atoms represent “non-numerical constant values” [1]. The value of an atom is itself. Table 1 lists example atoms. They are similar to global constants in C.

Table 1: Example Erlang Atoms

Example Erlang Atoms
exampleatom
example_atom
example@atom
exampleAtom
‘ExampleAtom’

Process Identifier

Processes are assigned process identifiers (Pid). They are used to uniquely identify each process (especially for use with inter-process communication) [11].

Tuple

A tuple is a “compound data type with a fixed number of terms” [12]. Tuples start with an open curly brace ({) and end with a close curly brace (}). They have the form: *{term*$_1$*, …, term*$_x$*}*.

Table 2: Example Erlang Tuples

Example Erlang Tuples
{one}
{two}
{three}
{a,b}
{c,d}

List

A list is a “compound data type with a variable number of terms” [13]. Lists have the form: [term$_1$, …, term$_x$].

Table 3: Example Erlang Lists

Example Erlang Lists
[one]
[two]
[three]
[a,b]
[c,d]

String

Similar to other languages, Erlang strings can be written with double quotes; they are actually stored as lists of ASCII values [14].

Table 4: Example Erlang Strings

Example Strings
“hello world”
“Erlang is awesome”
“You should learn Erlang too!”

Funs

Fun expressions are use to declare functions, except that the functions are not named [16]. Fun expressions can be saved and referenced using variables. The following line shows an example:

FunExpression = fun(param$_1$,…,param$_x$) -> true end.

Variables

Erlang is different from many programming languages in that variables are single assignment, and the '=' operator tries to match the patterns of the two operands [1]. Only on the first use of '=' does a variable receive a value; it becomes permanently bound to that value. Only in patterns can variable expressions be unbound (have no value). Table 5 shows various Erlang variable statements and their results. Notice that the names of variables always start with a capital letter.

Table 5: Example Erlang Variable Assignments and Results

Variable Assignments and Results		
Erlang Statement	**Comparison**	**Result**
X=5.	Unbound=5	True
X=6.	5=6	error
Y=8.	Unbound = 8	True
Z = 8.	Unbound = 8	True
Y=Z	8=8	True
X=Y.	5=8	False

Patterns

Patterns are used in multiple structures in Erlang (if statements, case statements, fun expressions). To describe what patterns are and how they work this section focuses on their use with Erlang functions. Pattern matching is a key feature used by Rich Services/Erlang; it drives router selection, policy selection and SDC message to interface binding [1].

Table 6: Example Erlang Function Definitions

Example Erlang Function Definitions

op({addition, Operand1, Operand2}) -> Operand1 + Operand2;
 	
op({subtraction, Operand1, Operand2}) -> Operand1 - Operand2.
 	
op(multiplication, Operand1, Operand2) -> Operand1 * Operand2;
 	
op(division, Operand1, Operand2) -> Operand1 / Operand2.

Table 6 lists a set of functions that could be defined and exported by some Erlang module. Notice that the first two functions have only one parameter (a tuple with three terms) while the second two functions have three parameters. The first two functions match a three term tuple where the first term is the atom *addition* or *subtraction*. The second two functions match any three inputs where the first input is the atom *multiplication* or *division*.

The *Operand1* and *Operand2* parameters in the function definitions in Table 6 are unbound variables; they bind to any matching value of the input parameter when the function is called. A successful function call binds the input to the parameter patterns.

Table 7: Example Erlang Function Calls and Results

Example Erlang Function Calls	
Function Call	**Result**
op({addition, 1,2})	3
op({subtraction,4,2})	2
op(multiplication, 5, 6)	30
op(division,10,2)	5
op(x)	Undefined
op({subtraction})	Undefined
op(d,5,6)	Undefined

Modules

Modules contain functions and attributes, each followed by a period. Attributes are settings of the module (such as the name) and begin with the single dash (-) character [18]. Figure 3 shows an example Erlang module. The *module* attribute should be the same as the file name and specifies the reference used to call functions defined in the module.

```
-module(my_first_module).

-export([english/0,spanish/0]).

english() -> “hello world!”

spanish() -> “hola el mundo!”
```

Figure 3: Example Erlang Module

Table 8: Example Erlang Module Function Calls

Example Module Function Calls and Results	
Function Call	**Return value**
my_first_module:english()	“hello world!”
my_first_module:spanish()	“hola el mundo!”

Processes

Erlang processes are “small self-contained virtual machines that can evaluate Erlang functions.” All code is executed within an Erlang process. Their use is extremely efficient relative to operating system processes. Because they are Erlang run-time process and not operating systems processes, using them simply requires knowledge of a few simple primitives [1].

Inter Process Communication

Processes communicate with each other using the send and receive primitives [1]. Each process has a mailbox in which messages are placed. Processes extract messages using pattern matching in the receive structure. For example, the *receive {test_message} -> received end* code waits for the message *{test_message}* to enter a process’s mailbox and extracts it from the mailbox.

Nodes

An Erlang node is defined as an “executing Erlang runtime system.” Nodes use names of the form *name@host* for distributed communication [15]. Processes run inside Erlang nodes. Processes are able to communicate with process on the same node, and processes are also able to communicate over a network with processes on other nodes.

Process Links

Process links allow processes to monitor each other's aliveness. In other words, if two processes are linked when either dies, the Erlang run-time system will send an exit signal to the remaining process [1]. This mechanism helps implement fault-tolerant systems.

By default when a process receives an exit signal from a process it is linked to, it will crash. But the process can avoid crashing by trapping exit signals. Exit signals will then be placed in the process's mailbox.

Process Monitors

Process links are two-way. Processes must become system processes to trap exit signals and avoid crashing. A monitored process may not care if the monitoring process crashes. A process can place a monitor on another process. If the monitored process dies then the monitoring process will be sent an exit signal; it must trap exits to process the signal or it will crash. If the monitoring process dies the monitored process will not receive an exit signal and will not crash.

Node Monitors

Processes can place monitors on nodes. If a node dies then the process receives a node down message that contains the name of the node.

Behaviors

An Erlang behavior is an “application framework that is parameterized by a callback module.” Erlang provides the framework and the programmer provides a callback module. The callback module consists of custom code and implements the functional pieces of the behavior [1]. The next few sections discuss the application, supervisor, gen_server and fsm behaviors.

1. Application

Erlang applications help to control complex applications. Mostly the behavior provides a central point for inserting and modifying configuration settings and interfaces for starting and stopping the application [1]. Every application must define a configuration file (*application_name.app*) as well as a callback module that defines the *start/1* and *stop/1* functions. Programmers put the code to start the application and to stop the application inside these functions. The framework also requires that application files be saved in a specified directory structure [22]. Applications are started and stopped on nodes using the application module. Callings *application:start(application_name)* and *application:stop(application_name)* will causes the *start/1* and *stop/1* functions to execute.

2. Supervisor

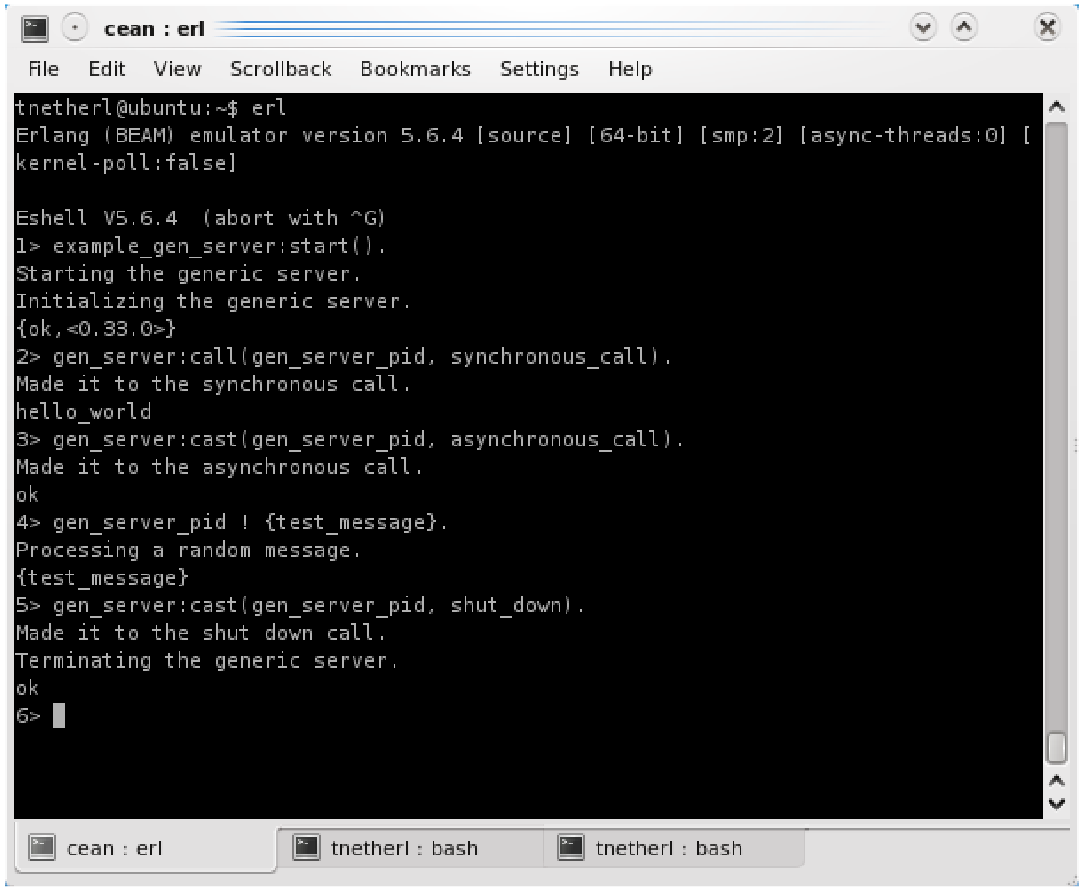
The supervisor behavior supports the creation of supervision trees. The concept is simple. Processes are placed in a logical tree hierarchy where parent processes monitor their children processes. The parent process keeps a specification for each of its children that its uses to restart them when they crash. It also follows a specified restart strategy when one or more child processes crash [1]. For instance, it can restart a single crashed process or terminate and restart all processes when a single process crashes. The callback module defines the *init/1* function, which contains all initialization logic for the supervisor, including the definition of a set of child specifications.

3. Generic Server

The generic server behavior helps to implement the server piece of a client-server architecture [19]. The callback module must define an *init/1* function that starts the server process, conducts any appropriate initialization tasks and sets the server state. It may also define the handle_call/3 and handle_cast/2 functions for providing synchronous and asynchronous communication with the server. The function *handle_info/2* allows the server to process any other messages. Figure 4 contains an example generic server, and Figure 5 demonstrates calls to the sever using an Erlang terminal.

```
-module(example_gen_server).
-export([start/0,init/1,handle_call/3,handle_cast/2,handle_info/2]).

-behavior(gen_server).

start() ->
        io:fwrite(user, “Starting the generic server.~n”,[]),
        gen_server:init({local, process_name}, example_gen_server,[],[]).

init(_Args) ->
        io:fwrite(user, “Initializing the generic server.~n”,[]),
        {ok, []}.

handle_call(synchronous_call, _From, State) ->
        io:fwrite(user, “Made it to the synchronous call.~n”,[]),
        {reply,hello_world,State}.

handle_cast(asynchronous_call, State) ->
        io:fwrite(user, “Made it the asynchronous call.~n”,[]),
        {noreply,State};

handle_cast(shut_down, State) ->
        io:fwrite(user, “Made it to the shut down call.~n”,[]),
        {stop,normal,State}.

handle_info(AnyMessage,State) ->
        io:fwrite(user, “Processing a random message.~n”,[]),
        {noreply, State}.

terminate(Reason,State) ->
        io:fwrite(user, “Terminating the generic server.~n”,[]),
                ok.
```

Figure 4: **Example Erlang Generic Server Module**



```
tnetherl@ubuntu:~$ erl
Erlang (BEAM) emulator version 5.6.4 [source] [64-bit] [smp:2] [async-threads:0] [
kernel-poll:false]

Eshell V5.6.4  (abort with ^G)
1> example_gen_server:start().
Starting the generic server.
Initializing the generic server.
{ok,<0.33.0>}
2> gen_server:call(gen_server_pid, synchronous_call).
Made it to the synchronous call.
hello_world
3> gen_server:cast(gen_server_pid, asynchronous_call).
Made it to the asynchronous call.
ok
4> gen_server_pid ! {test_message}.
Processing a random message.
{test_message}
5> gen_server:cast(gen_server_pid, shut_down).
Made it to the shut down call.
Terminating the generic server.
ok
6> 
```

Figure 5: Example Generic Server Calls

4. FSM

Finite state machines consist of a set of relations where each is defined as a state-event pair that causes the execution of an action(s) and transition into a new state [21]. In Erlang a finite state machine is represented with an Erlang module that uses the gen_fsm behavior.

```
StateName(Event, StateData) ->
            // code for actions goes here
            {next_state, NewStateName, NewStateData}.
```

Figure 6: Erlang Finite State Machine State Definition Structure

```
-module(example_fsm).

-behavior(gen_fsm).

-export([init/1, door_shut/2, door_open/2, terminate/3]).

init ([]) -> {next_state, door_shut, [no_key]}.

door_shut ({insert_key, Key}, [no_key]) -> {next_state, door_shut, [Key]}.

door_shut ({insert_key,Key}, [OtherKey]) -> {next_state, door_shut, [OtherKey]}.

door_shut (remove_key, [no_key]) -> {next_state, door_shut, [no_key]}.

door_shut (remove_key, [Key]) -> {next_state, door_shut, []}.

door_shut (open, [no_key]) -> {next_state, door_shut, [no_key]}.

door_shut (open, [correct_key]) -> {next_state, door_open, [correct_key]}.

door_shut (open, [wrong_key]) -> {next_state, door_shut, [wrong_key]}.

door_open ({insert_key, Key}, [no_key]) -> {next_state, door_open, [Key]}.

door_open ({insert_key,Key}, [OtherKey]) -> {next_state, door_open, [OtherKey]}.

door_open (remove_key, [no_key]) -> {next_state, door_open, [no_key]}.

door_open (remove_key, [Key]) -> {next_state, door_open, [no_key]}.

door_open (shut, [no_key]) -> {next_state, door_shut, [no_key]}.

door_open (shut, [correct_key]) -> {next_state, door_open, [correct_key]}.

door_open (shut, [wrong_key]) -> {next_state, door_open, [wrong_key]}.
```

Figure 7: Example Erlang Finite State Machine Module

Figure 6 shows the syntax for a state. The function names separate the individual states. The behavior passes state data through the second parameter of the function definition. For each state there can be any number of events; the first parameter defines each event with an Erlang pattern. After the actions are executed the finite state machine transitions to a new state or terminates. The *terminate/2* function contains termination logic.

Ports

Erlang's connects to the outside world through ports; these are primitive mechanisms by which Erlang communicates with external entities. An Erlang process can open a port and use it to send and receive lists of bytes. External programs in operating system processes are also able to read bytes from and send bytes to the port [20].

Unfortunately, ports support only primitive byte passing. Interpreting the bytes is a painful process. But there are a number of libraries that provide a higher level view to the programmer. These include the Java JInterface, Erlang IDL compiler and Erl Interface (for c) libraries [1].

Dynamic Code Loading

Erlang supports dynamic code loading. Processes execute Erlang functions. During run-time one can freeze the execution of a process and change the function it is executing. After unfreezing the process it will begin executing the new code.

Chapter IV An Introduction to Rich Services/Erlang

Our main contribution is the creation of Rich Services/Erlang. It is the first library to support the creation of Rich Services software systems. It is a set of Erlang modules that allow programmers to create Rich Services systems in Erlang. This chapter discusses the more critical features implemented by the library.

IV. A Miscellaneous Features

Rich Service Roles

The term *Rich Service* describes two different roles: *Rich Application Service* and *Rich Infrastructure Service*. In addition this paper identifies the *Rich Framework Service* for describing services that do not have a SDC. These reside in the highest level of a Rich Service's hierarchy.

The Message Format

Erlang messages are tuples. Programmers have direct access to each field in the tuple. Rich Services/Erlang message tuples consists of the Type, Tag, Dest, Src, RouterSelection, PolicySelection and Content fields.

The *Type* field is used to assist the messenger in interpreting the format of the *Content* field. Currently the atom *erlang* is the only Type value that the messenger recognizes. This message type does not force a specific format on the *Content* field.

The *Tag* field is used to mark individual messages with a unique identifier. The *Tag* field is most useful for capturing sequences of related messages in the SDC. This field does not require any specific format but user defined tags should be unique.

The *Dest* field stores message destinations. A message's destination is defined using a list of lists. Individual Rich Service destinations are represented using a list, and the *Dest* field is a list of one or more Rich Service destinations; therefore the *Dest* is a list of lists.

The *Src* field saves the source of a message. Sources can include the address of every SDC the message passes through; the field is saved as a list of RAS addresses.

The *RouterSelection* field supports the router selection process in the Messenger. The format of the field is determined by the designer of the messenger handling the message.

The *PolicySelection* field can be used to help select Rich Infrastructure Services in the router. The format of this field is set by the designer of the router that is handling the message.

Finally, the *Content* field is where the content of the message is placed. It is a completely customizable field that matches with SDC interfaces.

Message Addressing

Message destinations are saved in the *Dest* field of a Rich Services/Erlang message. The Messenger recognizes the following four addresses:

1. [{ras, RasName}]
2. [{ras, RasName, SdcName}]
3. [{sdc}]
4. [{sdc, SdcName}]

The 2nd and 4th tuples are placed in the *Dest* field by Service Data Connectors when they forward a message to the communication infrastructure. If the message is sent externally the 2nd tuple is used, otherwise the the 4th tuple is used. The 1st and 4th tuples are available to programmers for sending messages without specifying a specific SDC.

The Messenger knows that tuples 1 and 2 refer to a RAS and that tuples 3 and 4 refer to Service Data Connectors of the Messenger's Rich Service. The various address formats tell the messenger which algorithm to use for sending the message.

Hierarchical Addressing

The Rich Services architecture supports hierarchical decomposition of services. Some information may be necessary to specify a path through a series of Rich Services. The addressing mechanism supports multi-level addressing to provide this information.

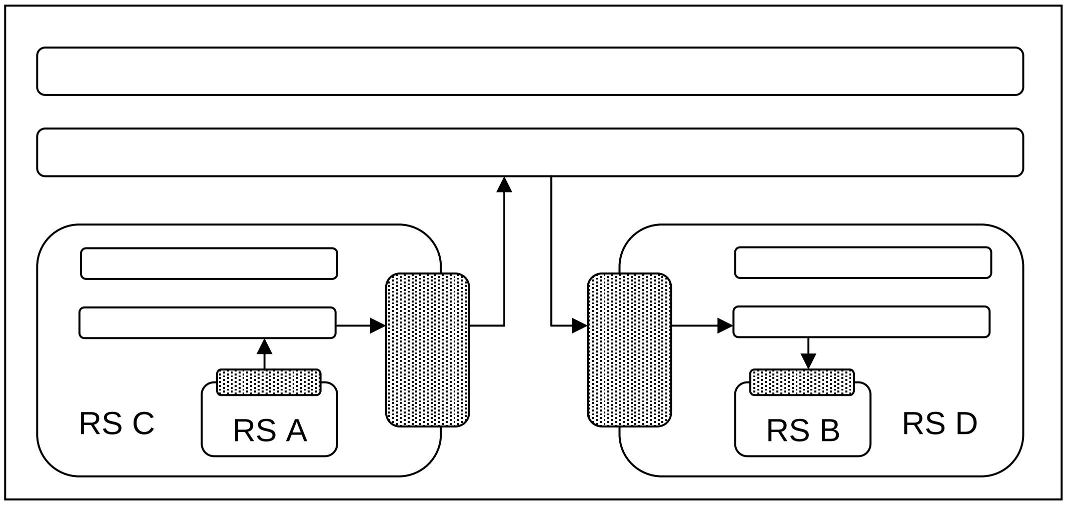


Figure 8: Example Two-Level Rich Service

The *Dest* field is a list of lists. It can address multiple destinations and each destination is a list of Rich Service addresses. Consider the message path from RAS A to RAS B in Figure 8. The message will pass through the Service Data Connectors of three Rich Application Services before reaching the SDC at its destination RAS. Moments before sending the message to the Messenger the SDC of RS A will set the source to:

```
[[
    {ras, RS_A, SDC_A}
]]
```

The next SDC will contain custom implementation logic for handling the message. If it forwards the original message without changes, the message source field would then be set to:

```
[[
    {ras, RAS_C, SDC_C},
    {ras, RAS_A, SDC_A}
]]
```

Upon accepting and processing the message the third SDC forwards it again. After sending the message the source would then be:

```
    [[
        {ras, RAS_D, SDC_D},
        {ras, RAS_C, SDC_C},
        {ras, RAS_A, SDC_A}
    ]]
```

Above is the value of the message's source that the SDC of RAS B would have. Any reply sent to the original source would use this address to exactly recreate the path through the hierarchy. The return path cannot always be reached though. The Service Data Connectors in the path must have logic (usually the SDC conversations wait for a response) to handle any replies.

Fault Tolerance

Rich Services/Erlang uses multiple, concurrent processes to implement every Rich Service. Unforeseen circumstances as well as programmer error present increases risk for the unsuccessful operation of any Rich Service. The integrity of every Rich Service is upheld using the supervisor behavior, process links and node monitors. The only event that should completely crash a Rich Service is the crash of the node it is running on.

Multiple Service Data Connectors

Rich Services/Erlang does not place a limit on the number of Service Data Connectors of a Rich Service. Multiple Service Data Connectors allow communication with multiple messengers or multiple connections to a single Messenger. The only constraint is that each SDC communicate with only one messeger.

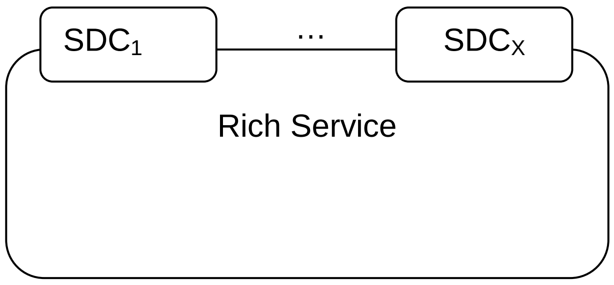


Figure 9: Rich Service with Multiple Service Data Connectors

Open Dynamic Sdc

Rich Service designers may want to leverage the existence of external Rich Services. But the designers of these external services cannot possibly open every SDC connection. These designers are able to provide SDC definitions that clients of the Rich Service use to open new SDC connections as needed. The Rich Service uses unique identifiers for each definition referred to as a *Sdc Selector*. Clients simply call into the Rich Service, providing the *SdcSelector* and connection information. The Rich Service starts a SDC and connects it to the specified Messenger.

Multiple Routers

Rich Service designers can define multiple routers, each with its own set of policies and policy selection rules. To accommodate multiple routers the Messenger must include Router selection rules. An interesting design choice allows for sending messages to multiple Routers. In this case a copy is sent to each selected Router.

Loading Rich Application Services

Complex Rich Services may statically define a set of internal Rich Application Services. These are loaded by the Rich Service at start up. They may also be reloaded during run-time.

IV. B Service Data Connectors

Interface Implementations

Service Data Connectors export both an external and internal interface. Later we describe the run-time implementation details of an SDC. Basically, an interface consists of a set of finite state machines and for each of those a set of initiation events (messages). The SDC matches incoming messages to one or more finite state machines using the pre-defined events, starts each selected finite state machine and sends them the message (as the first event).

Interface Definitions

```
[ {external, fsm_ex, [fun({msg, one}) -> match end], closed, {limit, null}},
  {internal,  fsm_in,  [fun({msg, two}) -> no_match end], closed, {limit, null}}]
```

Figure 10: Example SDC Interface Definition

Interfaces are defined using a list of tuples; the list includes one tuple per finite state machine. Figure 10 gives an example interface definition. The first field states to which side the SDC should bind the finite state machine. The second field specifies the module name of the finite state machine. The third field is a list of Erlang fun expressions that specify what messages should and should not activate the finite state machine. The meaning of the fourth field is a little more complicated. *Closed* interfaces start a new finite state machine for every matching initiation message. *Open* interfaces start one finite state machine that collects every message. The final field limits the number of finite state machines that can run concurrently.

Processing Received Messages

Every message received by a SDC either initiates a new conversation or belongs to an existing conversation (or is dropped). The external and internal SDC interface both process messages using the same algorithm, partly given in Equation 1.

n = number of finite state machines

tp = number of fun guards for finite state machine t

FunGuard_t = list of fun guards for state machine t

$$\sum_{t=1}^{n}\sum_{f=1}^{tp}\begin{cases} match \quad FunGuard_{\ t}\ [f](MsgContent) \rightarrow match \\ no\_match \quad FunGuard_{\ t}\ [f](MsgConent) \rightarrow no\_match \\ match \quad FunGuard_{\ t}\ [f](MsgContent) \rightarrow undefined \end{cases}$$

Equation 1: SDC Interface Finite State Machine Selection Algorithm

Equation 1 shows that the SDC iterates through the list of n tuples (the interface definition); for each tuple t, it iterates through a list of tp functions. Finally for each function, f, it calls FunGuard_t[f](MsgContent). *MsgContent* represents the content field of the message. The functions use pattern matching against the content field only.

Table 9: Example SDC Interface Finite State Machine Selection Results

Message Content	Conversation	Function Guard	Return Value	Result
{msg, one}	fsm_ex	fun({msg, one})	match	Select CM
{msg, one}	fsm_in	fun({msg, two})	undefined	Dismiss
{msg, two}	fsm_ex	fun({msg, one})	undefined	Dismiss
{msg, two}	fsm_in	fun({msg, two})	match	Select CM

Table 9 shows some SDC message to interface binding results using the interface definition provided in Figure 10. If a message causes the successful execution of at least one of the function guards and returns *match*, the SDC starts the finite state machine and sends it the message. If the function call returns *no_match* or results in a run-time error (the function call is undefined), then the message is not sent to the corresponding conversation manager.

Messages that do not initiate a conversation may still belong to an existing conversation. Using the message's *Tag* field the SDC selects which finite state machine is expecting the message. If it does not belong to a finite state machine the message is discarded by the SDC.

IV.C Business Logic Generic Server

Some Rich Services directly implement application/business logic. Designers could place this logic within the SDC finite state machines, but for the purpose of separating service interaction logic and application logic. Rich Services/Erlang provides the Business Logic generic server. Simple Rich Services use the business logic component to implement business/application logic internally and outside the SDC.

IV.D Messenger Generic Server

Registering Rich Services

Rich Application Services “connect” with, or plug-into, a higher level Rich Service by registering their Service Data Connectors with an external Messenger. The Messenger maintains a list of registered Service Data Connectors and important connection data. As discussed earlier, Rich Services can have multiple Service Data Connectors, making it possible that any one RAS have multiple connections to a single Messenger.

Un-Registering Rich Services

At the moment there is not a function for explicitly un-registering a RAS from the Messenger. Although, when a RAS’s node stops or crashes a node down message is generated and sent to the Messenger. Consequently, the RAS’s information is erased from the Messenger. When a Rich Services is stopped using the Erlang application behavior, the node is immediately stopped to generate this message.

Receiving, Routing and Processing Messages

In Rich Services/Erlang, messages are handled by the communication infrastructure in three steps. The Messenger receives a message from the source and selects one or more routers to forward it to (it could be duplicated).

Each Router processes the message and then sends it back to the Messenger. The Messenger sends the message to its destination(s).

When the Messenger receives a message it checks the message against the guard of every Router. The Messenger stores the name of an Erlang module and a list of Router identifications. The Erlang module defines at least one function for each router identification (the function name is the id) with a message pattern that the router accepts.

n = number of routers

m = Message,

rsm = Erlang Router Selection Module

RouterID = List of Router Identifications

$$\sum_{i=1}^{n} \begin{cases} selected & rsm{:}\,RouterID\left[i\right](m) \rightarrow match \\ not\ selected & rsm{:}\,RouterID\left[i\right](m) \rightarrow no\_match \\ not\ selected & rsm{:}\,RouterID\left[i\right](m) \rightarrow undefined \end{cases}$$

Equation 2: Route Selection Algorithm

The Messenger uses the algorithm in Equation 2 for selecting the appropriate routers. The router ids are Erlang atoms. For each router id there should be at least one function in the router selection module whose name is the id. Function calls that return *match* result in sending the message to the router specified by the given id; the Messenger uses the Router Id to select the reference of the corresponding Router. If the call returns anything other than match, the message is not sent to the router. The Messenger will duplicate a message when sending it to more than one Router.

Table 10: Example Router Policy List

Router Ids
[router_one, router_two]

```
-module(rsm).

-export([router_one/1, router_two/1]).

router_one({_,_,_,_,{router, one},_,_})  -> match.

router_two({_,_,_,_,{router, two},_,_}) -> match.
```

Figure 11: Example Policy Selection Module

Table 11: Example Policy Selection Results

Test Messages		
{erlang, tag, dest, src, {router, one}, ps, msg} {erlang, tag, dest, src, {router, two}, ps, msg}		
Router Selection Results		
Message Function Calls	**Return Value**	**Result**
rsm:router_one({erlang, tag, dest, src, {router, one}, ps, msg})	match	Select
rsm:router_two({erlang, tag, dest, src, {router, one}, ps, msg})	undefined	Ignore
rsm:router_one({erlang, tag, dest, src, {router, two}, ps, msg})	undefined	Ignore
rsm:router_two({erlang, tag, dest, src, {router, two}, ps, msg})	match	Select

Table 11 shows the result of sending two messages to a Messenger with the router selection module given in Figure 8. The functions (router selection rules) filter messages for the routers. The router selection rules reflect the routing strategies created by the designer of the system.

Routers pass messages to the Messenger when they are finished. When a Rich Service registers a SDC with the Messenger it provides it's *Send Parameters* - Erlang tuples containing the necessary information for the Messenger to send the Rich Service messages. For example, the *Send Parameters* of an Erlang Rich Service SDC uses the form: *{gs, Reference}*.

The Messenger knows that it is sending the message to a generic server referenced by *Reference*.

A message destination may specify a RAS or internal SDC interface without specifying an individual SDC. In this case the Messenger sends the messages to every SDC of the destination RAS or every internal SDC of the Rich Service. If a SDC is specified, the message is sent only to that SDC.

The Messenger uses the *Type* field of the message to interpret the format of the *Content* field of the message. At the moment it only supports a *Type* definition of *erlang* and does not care how the *Content* field is structured. The Messenger will eventually support communication mediums other than those based strictly on Erlang. For instance, designers may want to integrate web services using soap. The *Content* field will then be forced to contain relevant information for communicating via soap, and the *Type* field would be defined with an atom that tells the Messenger to apply this format and forward the message using soap protocols.

IV.E Router

A Router enforces routing logic and routing level policies on messages. Rich Infrastructure Services implement both concerns. Rich Service/Erlang permits multiple Routers per Messenger for the purpose of separating routing concerns to reduce the complexity of individual Routers. Figure 12 shows that one Messenger can have any number of Routers. An interesting design choice allows messages to be duplicated and sent to multiple Routers.

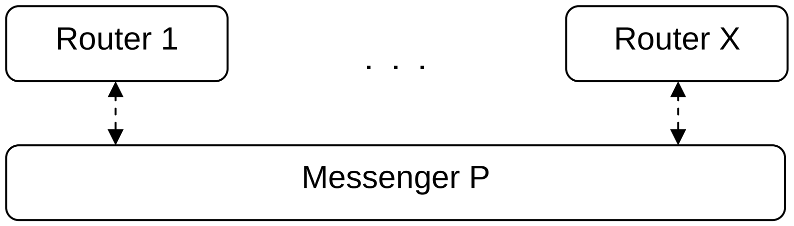


Figure 12: **Rich Services/Erlang Messenger and Router Configuration**

Routers belonging to the same Rich Servies share one code repository of Rich Infrastructure Services; these are saved in a specified location in the Rich Service's directory structure. Individual routers may include the same RIS in their specifications, although a separate, private instance runs per Router.

Starting Rich Infrastructure Services

A Rich Service designer must create a configuration file for each Router they create. This file contains a set of tuples, each being a specification for a RIS connected to the Router. The tuple has the form:

```
{Location, Name,  ConnectionCount}
```

The *Location* field specifies the directory location of the service. At the moment only a value of *local* is supported; the RIS must exist within the directory structure of the Rich Service that uses it. The *Name* field identifies the RIS with an atom. The Router uses this atom in its routing algorithm.   The ConnectionCount specifies how many SDC connections will be opened between the RIS and the Router.

Table 12: Example RIS Specification Tuples

Example RIS Specification Tuples

{local, ris_one, 1}
{local, ris_two, 2}

An example of a Router's RIS specification tuples is given in Table 12. While the values of the *Location* and *Name* fields are straightforward, the values of the *ConnectionCount* field might require further explanation. The first has a value of 1, meaning that the Rich Infrastructure Service connects to the Router with one SDC. The Router passes the RIS one connection to the RIS at start up. A connection includes a reference to the Router. The second specification tells the Router to pass two connections to the RIS.

Starting each RIS is simple. The Router must start a node, giving it a unique name. It then links each node to the code path of the Rich Service/Erlang library and to the directory of the RIS. Finally it will use Erlang rpc to start the service on the node.

Registering Rich Infrastructure Services

Newly started Rich Infrastructure Services must register their Service Data Connectors with the Router that started them. The Router saves the connection information and places a monitor on the RIS's node at registration. The RIS also places a monitor on the Router's node. If the Router dies so do the Rich Infrastructure Services. If one of the Rich Infrastructure Services dies then the Router will restart it.

Receiving Messages

A Router receives messages from its Messenger. Since it is a generic server this feature is implemented using a generic server cast.

Routing Messages and Applying Policies

This paper defines message routing as determining the destination of a message. It defines policy enforcement as determining and activating the correct policies for a message; those policies may transform the message, influence routing decisions or generate some other set of actions.

Rich Service/Erlang designers impose routing logic and policy enforcement using Rich Infrastructure Services. Each Router uses a policy selection module that must define the *ris_list/0* function and a function for each RIS. Figure 13 shows an example of such a module.

```
-module(one_policy_selection_module).

-export([ris_list/0,ris_one/1,ris_two/1]).

ris_list() -> [ris_one,ris_two].

ris_one({erlang, Tag, [[{ras, big_ben}]], Src, _, _, Content}) -> match.

ris_two({erlang, _, _, _, _, _, _}) -> match.
```

Figure 13: Example Policy Selection Module

The module name begins with the unique atom identification of the Router it belongs to and ends with *_policy_selection_module*. The ris_list/0 function returns a policy list of RIS atom identifications. These must match with the names given in the tuple specifications. Most importantly, this list determines the order in which policies are applied to messages.

For each RIS identification in the policy list there must be at least one function definition named with the identification. These functions guard the Rich Infrastructure Services they represent using Erlang function-based pattern matching.

```
n = number of policy RIS
SelectedRisList = [],
m = Message
psm = PolicySelectionModule,
RisList = rist_list(),
n = RisList.length,
```

$$\sum_{i=1}^{n} \begin{cases} SelectRisList.insert(m, back), & psm: RisList[i](m) \rightarrow match \\ SelectedRisList, & psm: RisList[i](m) \rightarrow no\_match \\ SelectedRisList, & psm: RisList[i](m) \rightarrow undefined \end{cases}$$

Equation 3: Policy Selection Algorithm

Routers use Equation 3 to filter out the appropriate policies for each message, resulting in a subset of the policies avaliable in the Router. The Router then proceeds to send the message to each RIS (one at a time) in the order they are selected. The Router uses a send and forget strategy. It removes the first item from the list, looks up the RIS's information and sends it the message as well as the remaining policies in the list. The RIS must send the message and policy list back to the Router when it is finished. The Router then continues moving through the selected policies.

Chapter V Rich Service/Erlang System Design and Implementation

The creation of a Rich Services/Erlang system first involves designing an architecture using Rich Services that meets the requirements. The next step is to implement the system using Rich Services/Erlang. This chapter provides the reader with the process for creating and implementing any Rich Services system.

V.A Design Guidelines for Rich Services Systems

The authors of Rich Services have created a service-oriented development process to be used with Rich Services that creates a clean separation between the logical model of a system and its implementation [6]. They introduce an iterative, three phase process (where each phase may have multiple iterative stages). Designers use it to refine a system's architecture so that it meets the specified requirements. Cross-cutting concerns are abstracted and dealt with in later points in the development process [4,6].

The first phase is Service Elicitation. The process results in a service repository representing system requirements and a domain model that includes cross cutting concerns (such as security and encryption). The domain model is used to identify roles - unique entities that interact with each other. Using message sequence charts the interactions between roles are specified. A service is defined as an interaction between roles. Cross cutting concerns are then injected into the interactions. The result is the system's service repository which is used to identify the role domain model.

The second phase is Rich Services Architecture. The phase uses the service repository and the role domain model to define a hierarchical set of Rich Services as a logical model of the system. One can map the services and roles to the Rich Services architecture by either mapping roles to Rich Application Services and services to the routing mechanisms in which the Rich Application Services participate and then wrapping the resulting system with an SDC or by mapping each service to a composite or simple RAS. Cross-cutting concerns should always be introduced as Rich Infrastructure Services in the router. Throughout this process the designer chooses the communication channels, the location of concerns, the routing strategies and what messages are published through the Service Data Connectors. The resulting Rich Services architecture makes up the Rich Services model.

The third phase is Rich Service Architecture Implementation. The phase establishes a relationship between the Rich Services model and its implementation. For the most part this can be done using the Rich Services/Erlang architecture. In some cases though application logic may be implemented using resources outside the Rich Services/Erlang library. The final step is to create the physical implementation using the Rich Services/Erlang library.

V.B Implementation Guidelines for Rich Services/Erlang Systems

Rich Services are concurrent, distributed systems; concurrent and distributed software systems are complex. Erlang offers the application behavior to provide a central point of control and configuration for managing complex Erlang-based software [22]. Rich Services/Erlang uses the application behavior to ease the implementation process for every Rich Service. This chapter discusses the characteristics and use of Rich Services/Erlang for Rich Services implementations.

V.B.1 Configuration Basics

Configuration File

The Rich Services programmer must create a *rich_service_app.app* configuration file, which is the appropriate place for the configuration parameters required by an Erlang application. Figure 14 shows the basic contents of every *rich_service_app.app* file. The *{env,[]}* tuple contains a list of the required and optional configuration parameters.

```
{ application,
  rich_service_app,
  [ { description, "App Config Example"},
    { vsn, "1.0"},
    { modules,
      [    rich_service_app,
           rich_service_supervisor,
           rsgscRichService,
           rsgscServiceDataConnector,
           rsgscRouter,
           rsgscMessenger,
           rsgscBusinessLogic,
           rshMessageCreator,
           rshGscCalls
      ]
    },
    { registered, []},
    { applications, [kernel, stdlib]},
    { mod, {rich_service_app, []}},
    {env, []}]
}.
```

Figure 14: Example Rich Services/Erlang Application Configuration File

Envelope Configuration Parameters

{top_level_code_path, Path}

The top level code path points to the top level directory where the Rich Service exists. The Path parameter is the absolute path of this directory.

{rich_services_code_path, Path}

The Path parameter is the absolute path where the compiled Rich Services/Erlang files are placed.

{composition, Composition}

Every Rich Service must include this parameter. Composition can be *complex* or *simple.*

{ sdc_start@initANDconnectionIStatic, List}

This tuple provides a means for defining Service Data Connectors that have statically defined connections and that are started when the Rich Service starts. The List contains a tuple defition for all such SDCs.

{ sdc_start@initANDconnectionISdynamic, List}

This tuple provides a means for defining Service Data Connectors that have dynamically defined connections and that are started when the Rich Service starts. The List contains a tuple defition for all such SDCs.

{ sdc_start@run_timeANDconnectionISdynamic, List}

This tuple provides a means for defining Service Data Connectors that have statically defined connections and that are started when a client requests a connection. The List contains a tuple defition for all such SDCs.

{router_selection_module, List}

The List parameter is the file name of the Erlang module that contains the router selection logic used by the Messenger.

{routers_ids,List}

The List parameter is a list of Erlang atoms where each atom is the unique id used by the Messenger to reference the Routers defined in the Rich Service.

{ras_specs, List}

The List parameter is a list of tuples defining the Rich Application Services that are started and connected to the Rich Framework Service at start time. These definitions include a unique id (Erlang atom) for each RAS.

{messenger_reference, Atom}

This tuple is optional. It may be the case that the Rich Service's Messenger should have a named reference. In this case the Messenger gen_server would be referenced by the Erlang atom, Atom.

{reference, Atom}

This tuple is optional. For Rich Services that should be addressable this parameter forces the initialization process to give the Rich Service gen_server the reference Atom.

Table 13 provides a quick reference for the configuration parameters. The first column lists the configuration parameters. The second column states the Rich Services roles that use the parameters. The third column specifies if a parameter is required.

Table 13: Rich Service/Erlang Application Configuration Parameter List

Configuration Tuple	**Used In**	**Required**
{top_level_code_path, Path}	RAS,RIS,RFS	RFS
{rich_services_code_path, Path}	RAS,RIS,RFS	RFS
{composition, Composition}	RAS,RIS,RFS	RFS
{ sdc_start@initANDconnectionIStatic, List}	RAS,RIS	Yes
{ sdc_start@initANDconnectionISdynamic, List}	RAS,RIS	Yes
{ sdc_start@run_timeANDconnectionISstatic, List}	RAS,RIS	Yes
{router_selection_module, List}	RAS,RIS,RFS	Composite
{routers_ids,List}	RAS,RIS,RFS	Composite
{ras_specs, List}	RAS,RIS,RFS	Composite
{messenger_reference, Atom}	RAS,RIS,RFS	No
{reference, Atom}	RAS,RIS,RFS	No

V.B.2 Component Implementation

Service Data Connector

Configuration parameters for each SDC are defined in an Erlang module. The name of this module has the form *sdcid_sdc_configuration*. The *sdcid* part of the module name is the unique identifier used by the Rich Service at run-time to reference the SDC. The module contains a definition for each internal and external conversation. This includes the name of the finite state machine module that implements each conversation and the message patterns that initiate the conversation.

The interface of each SDC is implemented using finite state machines. The system implementer must create an Erlang finite state machine module for each interface definition. The names of these modules are used in the configuration file to match messages patterns to conversations.

Messenger

Certain configuration parameters for the messenger, such as a statically defined reference, are placed inside the application file for the entire Rich Service. The router selection rules for the Messenger are defined inside of an Erlang module.

Router

Certain configuration parameters for the router are defined within the configuration file for the Rich Service. The policy selection rules are placed inside of an Erlang module.

Business Logic

The Business Logic component is implemented using an Erlang generic server. The system implementer therefore places the application logic inside a callback module that parameterizes the generic server.

V.B.3 Application Directory Structure

Erlang component implementations are saved in a pre-determined directory structure. It is a derivative of the directory structure for Erlang applications that better supports the implementation of its components. It includes the *ebin* folder and may include the following folders: *messenger, sdc*, *router, business_logic* and *ras*.

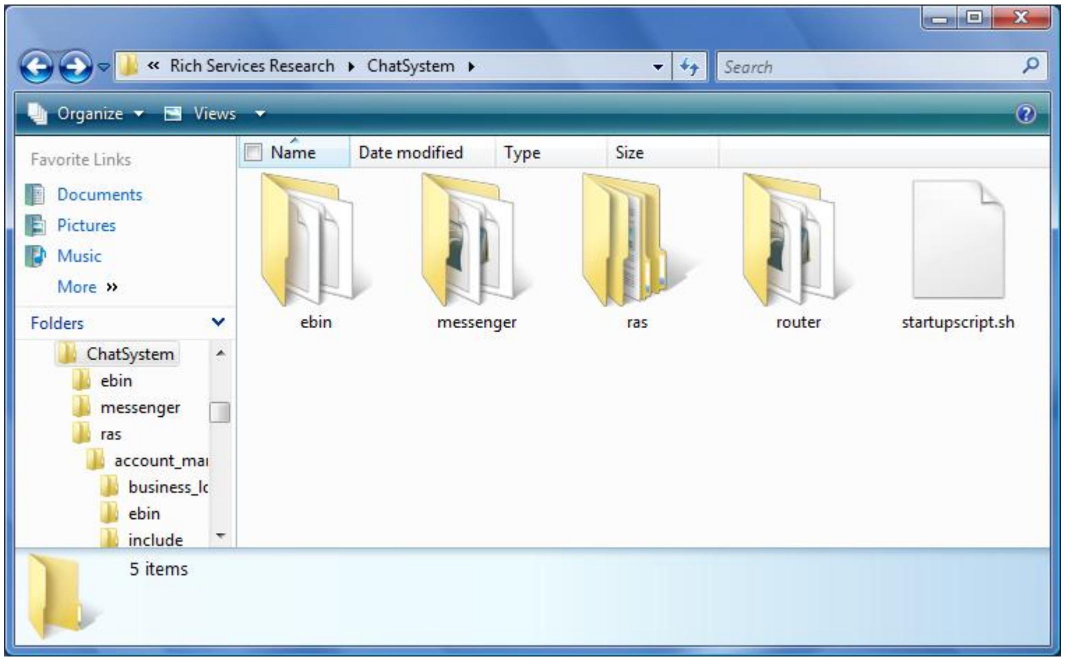


Figure 15: Example Rich Services/Erlang Application Directory

The *ebin* folder is taken from the Erlang *application* directory structure. It contains all of the compiled Erlang modules for the Rich Service. The Rich Service's node must have this directory in its code path.

The *messenger* folder contains all Erlang modules that make up the Messenger logic. At the moment it only contains the router selection module.

The sdc folder contains folders and files. For each SDC belonging to the Rich Service there is a folder containing its interface implementations. For each SDC there is a configuration file that contains the interface definition.

The router folder contains files and a folder. The configuration files for each router defined inside the Rich Service reside in the router directory. The single sub-folder in this directory is named *ris*. For each RIS there is a folder in the *ris* folder containing it application directory and files.

The configuration file for each router specifies the policies that the Router uses. For each RIS the file contains a tuple that provides the RIS's id (assigned and used by the routers) and the number of Service Data Connectors that will connect to the Router. The id is used to find the correct RIS folder in the *router* folder; the names of these folders are the ids of the RIS they contain.

The *business_logic* folder contains the business logic modules for *simple* Rich Services. The only required file is named *code_implementation,* and is the callback module for the Business Logic generic server.

The last folder is named *ras*. It contains the directories for each RAS defined in the Rich Service. These folders are named using the unique id assigned to the Rich Application Services in the configuration file of the Rich Service.

V.B.4 Miscellaneous Concepts

Naming Scheme

Every Rich Service uses the application name *rich_service_application*. Because every Rich Service executes within its own private node, there will not be a name conflict among Rich Services. Application name scoping occurs at the node level.

Execution Control (starting and stopping)

Rich Services are started and stopped using the application module. The start and stop functions are always passed the application name *rich_service_application*.

V.C Converting Traditional Erlang Process into Rich Services

How can a person convert a traditional Erlang process into a Rich Service or use it as an RIS/RAS? We have two basic options. The simple option would be to implement a simple SDC and Business Logic component. The SDC must export some interface that controls access to the original process. Active conversations started through the SDC send messages to the Business Logic component. This component cannot directly interact with the original process because it cannot handle the messages from the original process. To handle these messages communication with the original process should be handled in a process spawned by the Business Logic component. The spawned process mediates communication between the two. In this implementation the user must add the SDC interface messages, Business Logic components messages, SDC interface implementations, Business Logic component implementations and the messages and logic for the mediator between the Business Logic component and the original process.

In the second option the service of the traditional process should be converted entirely into a Rich Service. Its logic should be implemented directly in the Business Logic component. The SDC would be exactly the same as that described in the first option. The advantage is that the complexity of implementing a mediator process between the traditional process and the Business Logic component is removed.

Chapter VI Rich Services/Erlang Run-Time Description

VI.A From Design Architecture to Run-Time Implementation

The Rich Services architecture allows the system designer to create a logical view of a system. Figure 16 shows the logical architecture for an imaginary Rich Services system. A convenient property of Rich Services is that each component in the architecture is implemented one-for-one. For example, the Messenger drawn in Figure 16 could be implemented directly by an off the shelf messaging system [6]. Realizing the logical architecture with an implementation is not a complicated or mysterious process.

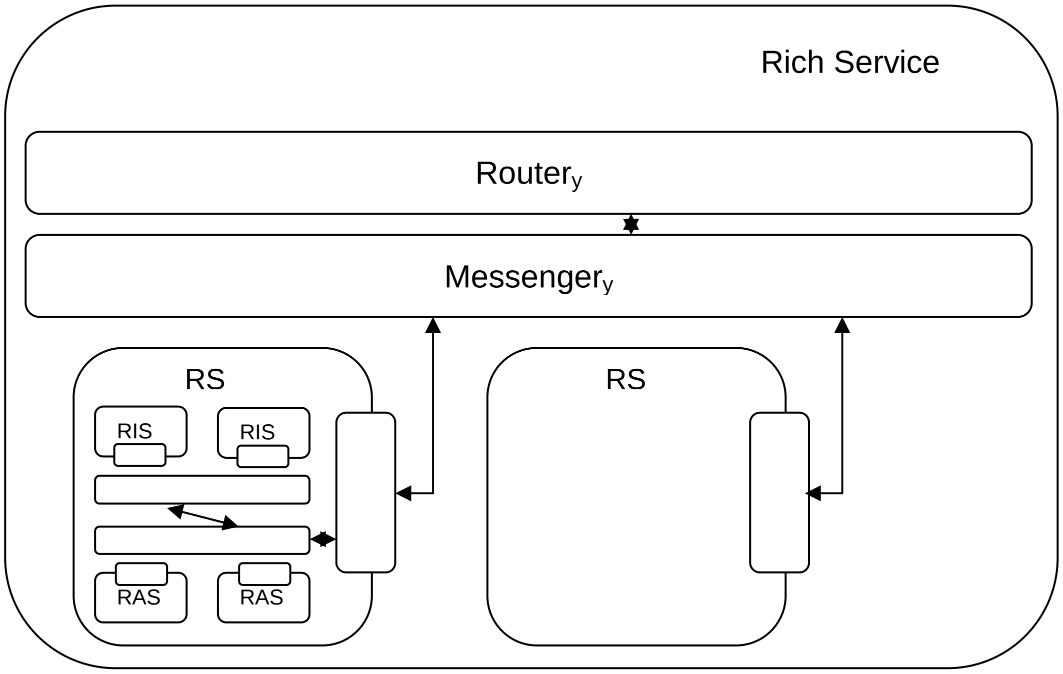


Figure 16: **Example Rich Framework Service Design**

Logically Rich Services/Erlang implements the Rich Services components one for one in software, although the implementation of any component may consist of multiple concurrent processes. Figure 17 represents the same Rich Service Y as Figure 16, accept that RS X is shown as the set of concurrent process of its running implementation in Erlang.

Rich Service Y, Rich Service X and all of the Rich Application Services and Rich Infrastructure Services in Rich Service X are distributed and running on six separate nodes. Seven processes alone realize Rich Service X. There are seven two-way paths of communication, five open node monitors and six process links.

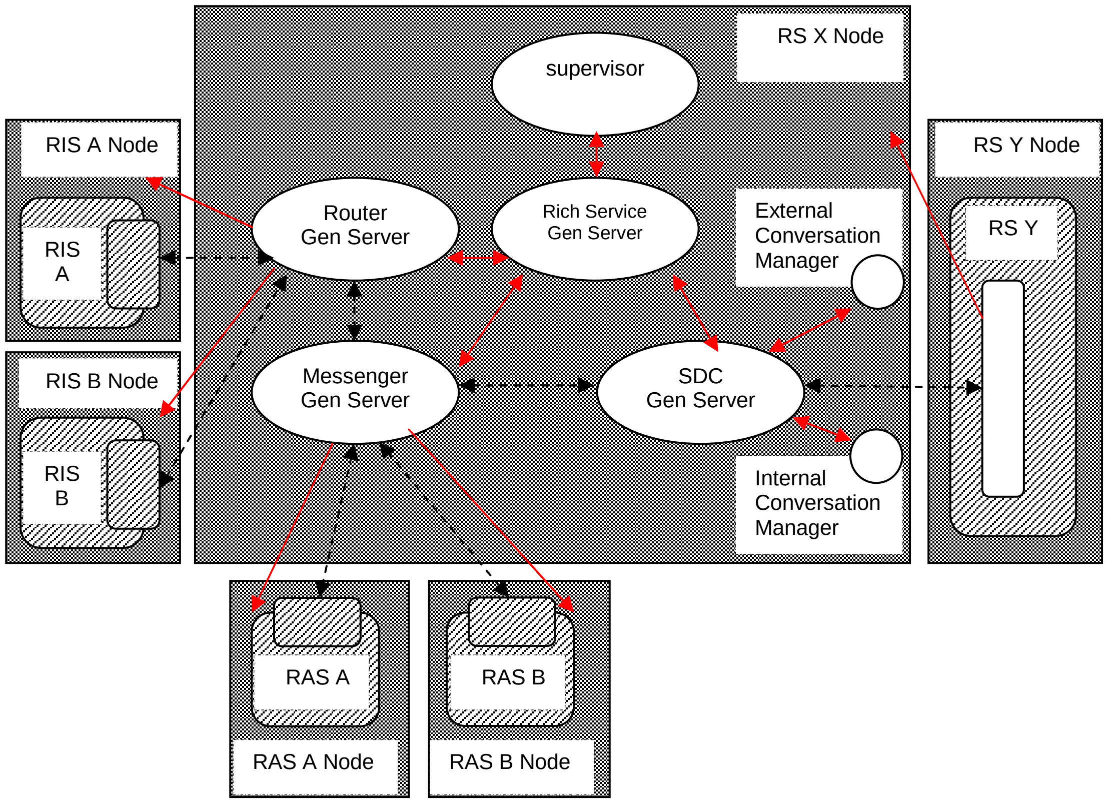


Figure 17: **Example Rich Application Service Run Time Implementation**

All of these processes, process links, node monitors and communication paths implement and integrate one node in a Rich Service system. Imagine the complexity of an implementation diagram for the entire Rich Framework Service Y. This section focuses on the run-time implementation of any Rich Service using the Erlang-based infrastructure.

VI.B Run-Time Implementation Basics

VI.B.1 Nodes

One must first start a node before starting a Rich Service application. Very important is that the node is given a short or long name that can be recognized across the network that the Rich Service use to communicate. The node for any Rich Service is either started manually or from within the Rich Service/Erlang library. The second case occurs when a RAS or RIS service is listed in the configuration of a higher-level Rich Service and is started by a Rich Service generic server or Router generic server.

VI.B.2 Processes

Each Rich service exists within a private node and executes as a number of concurrent processes. Figure 17 shows that RS X runs within a node labeled Node X. Notice also that in Node X there are a number of processes (drawn as circles and ovals).

If we abstract its processes into single entities, according to function, any Rich Service could be represented using Figure 18. These entities form a hierarchical ordering of two-way process links that improve fault-tolerance. At the top of the tree an Erlang supervisor process keeps the Rich Service running. At the next level the Rich Service Generic Server provides the single point of control for a Rich Service. At the bottom some set of peer processes implement the core Rich Service components (Messenger, Router, Service Data Connector and Business Logic Generic Server).

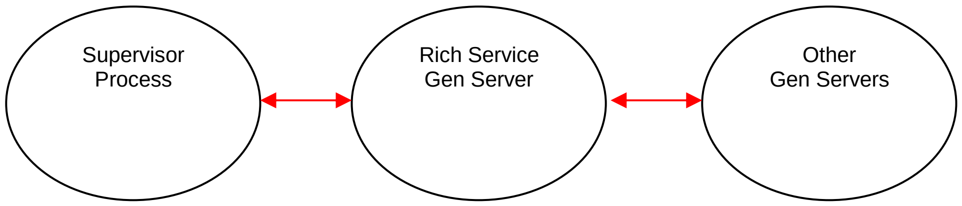


Figure 18: Rich Service Run Time Implementation Process Categories

VI.B.2.a Supervisor

Once the node begins running, the Rich Service is started by calling *application:start(rich_application_app)*. The first Rich Service process that starts is the supervisor. Its main purpose is for starting and monitoring the Rich Service Generic Server process. If the Rich Service Generic process dies then the supervisor will restart it to keep the Rich Service running. The only failure it cannot protect against is a crashed node.

VI.B.2.b Rich Service Generic Server

Every running Rich Service has a Rich Server Generic Server process. This process will implement slightly different features depending on the role of the Rich Service. This process ties the entire Rich Service together. It uses the application configuration variables and configuration files of the Rich Service to spawn and link to the next tier of generic server processes. These are the Messenger, Router, Service Data Connector and Business Logic Generic Servers of the Rich Service.

VI.B.2.c Rich Service Component Generic Servers

Service Data Connector

The SDC implementation consists of one generic server process and some number of other processes. Rich Services/Erlang refers to the *other* processes as conversation managers. They help implement the interface.

Interfaces in the SDC manage interactions (conversations) with internal Rich Application Services and external Rich Application Services rather than implement application logic; they are implemented using finite state machines. SDCs may implement one finite state machine for each accepted message or implement a smaller set of finite state machines where some accept more than one accepted message. Finite state machines may accept the same message.

An interface call is simply the receipt of a message. The SDC binds a set of message patterns to each finite state machine. The receipt of one of these messages results in the spawning of all matching Erlang finite state machine processes. A conversation manager process handles the execution of interface calls for each finite state machine. It begins a conversation by starting the finite state machine and sending it the message, terminates conversations and manages message flows through active conversations.

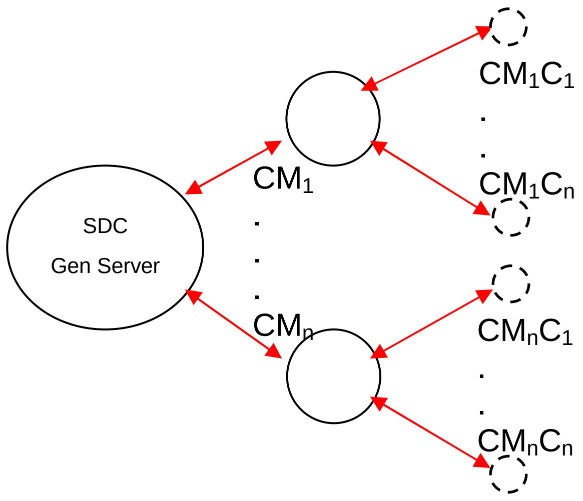


Figure 19: Service Data Connector Run Time Implementation

An executing finite state machine is a conversation. Conversations receive and send messages throughout their lifetime. When a conversation starts, the SDC generic server and the conversation manager save the *Tag* field of the original message. All outgoing messages should have this tag if they generate replies, which map to the correct conversation manager using the *Tag* value in their tag fields.

The conversation manager receives two types of messages from the SDC Generic Server. Both contain the received message. The first specifies messages that are initiating new conversations. The second specifies messages that are part of an existing conversation.

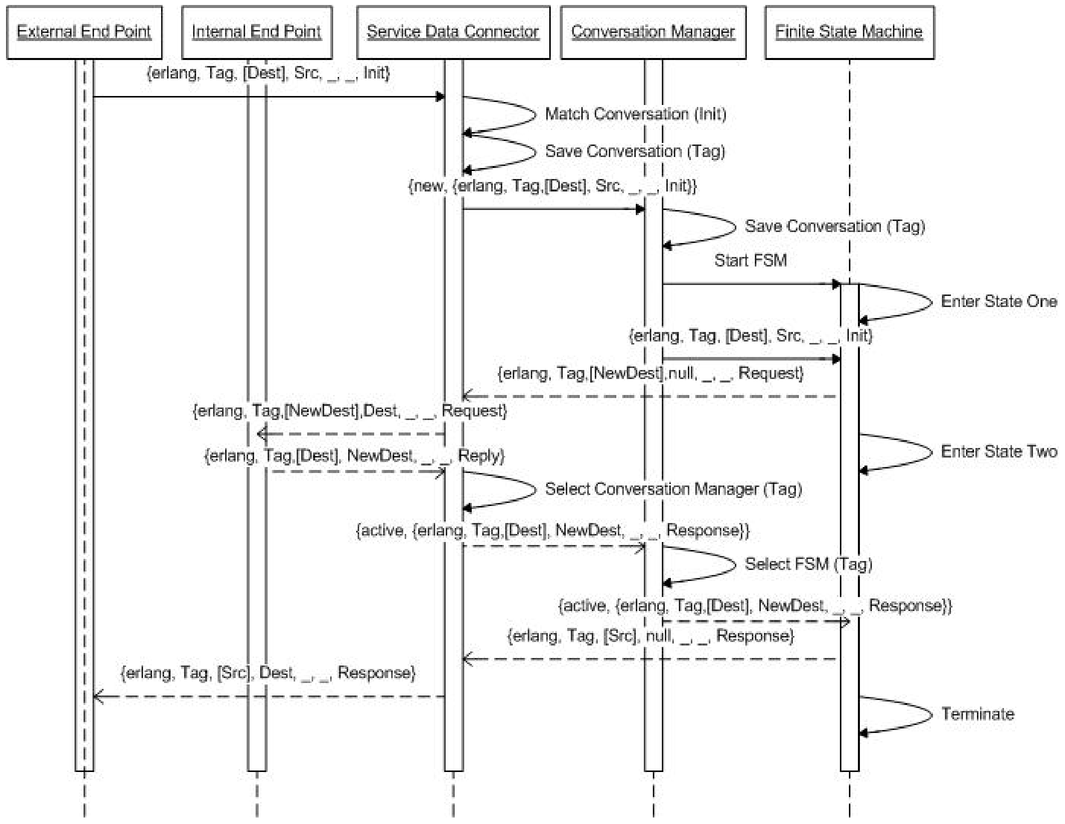

External End Point
Internal End Point
Service Data Connector
Conversation Manager
Finite State Machine
{erlang, Tag, [Dest], Src, _, _, Init}
Match Conversation (Init)
Save Conversation (Tag)
{new, {erlang, Tag,[Dest], Src, _, _, Init}}
Save Conversation (Tag)
Start FSM
Enter State One
{erlang, Tag, [Dest], Src, _, _, Init}
{erlang, Tag,[NewDest],null, _, _, Request}
{erlang, Tag,[NewDest],Dest, _, _, Request}
Enter State Two
{erlang, Tag,[Dest], NewDest, _, _, Reply}
Select Conversation Manager (Tag)
{active, {erlang, Tag,[Dest], NewDest, _, _, Response}}
Select FSM (Tag)
{active, {erlang, Tag,[Dest], NewDest, _, _, Response}}
{erlang, Tag, [Src], null, _, _, Response}
{erlang, Tag, [Src], Dest, _, _, Response}
Terminate

Figure 20: **Example Service Data Connector Interface Call**

Figure 20 shows a message sequence chart representing a small conversation within one SDC. Some external source (Src) sends a message to a Rich Service (Dest). The SDC at Dest matches the message content (Init) to the interface and saves the conversation tag (Tag). It then forwards the message to the conversation manager of the selected conversation. The conversation manager saves the conversation tag (Tag), starts the finite state machine and forwards the message. The finite state machine receives the message and initiates a request to NewDest. Eventually the NewDest sends a reply (with content Response). It is forwarded through the SDC and conversation manager to the finite state machine. The conversation sends a reply to the initial request and terminates. The external end point will forward the last reply to Src.

Business Logic Generic Server

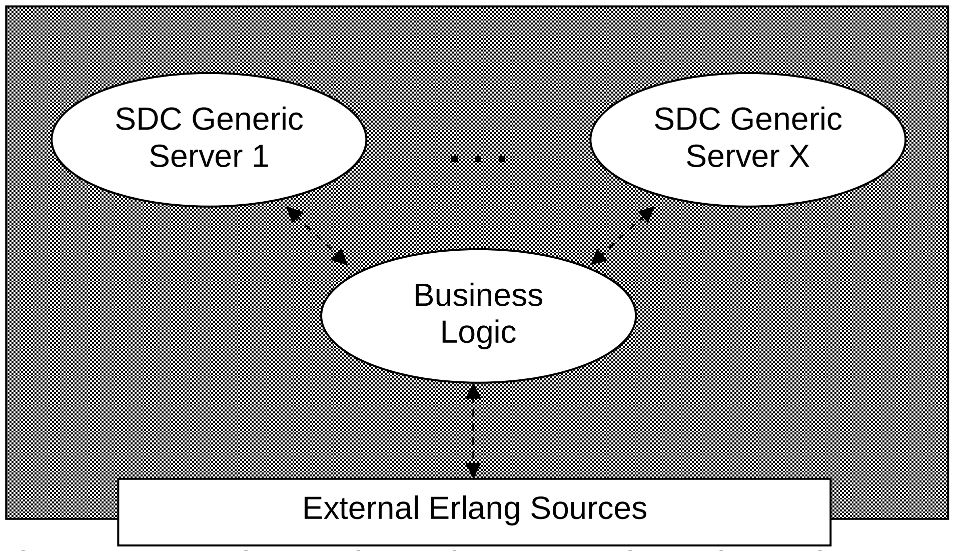


Figure 21: Business Logic Generic Server Run Time Implementation

In simple Rich Services designers should implement application logic using the Business Logic Generic Server. The application logic is placed inside the callback module of the generic server. Messages flow into the process through the Service Data Connectors and through external sources.

Messenger Generic Server

A running Messenger consists of a single generic server process. This process places monitors on the nodes of all registered Rich Application Services, as shown in Figure 17. If those nodes crash it will remove their registered data from its state and send a message to the Rich Service generic server for further fault recovery.

Router Generic Server

A single generic server process implements each Router within a Rich Service. When a Rich Infrastructure Service registers with the Router, the Router places a node monitor on its node. If the node crashes the Router will restart the RIS. The RIS also places a node monitor on the Router's node. If the Router node crashes then the RIS will shut down.

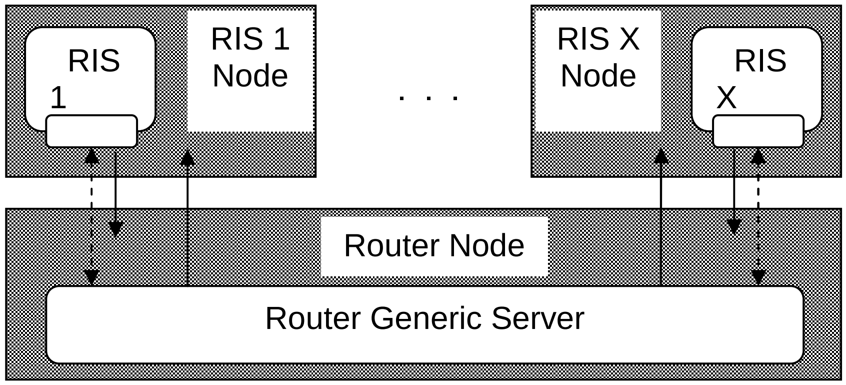


Figure 22: Rich Services/Erlang Router Implementation

Chapter VII Enterprise Integration Patterns Case Study

Rich Services/Erlang supports the use of many common software design patterns. We present the use of various enterprise integration patterns within a Rich Services/Erlang system as a case study within this chapter. Our discussion leverages material from the book Enterprise Integration Patterns.

The authors classify enterprise integration patterns into six categories. Messaging end points, message construction, message routing, message transformation, messaging channels and system management make up the complete list [3]. They specify at least sixty different patterns. Time constraints prevented this paper from presenting the use of each of these patterns. Therefore we attempted to introduce an interesting subset that highlights the features of Rich Services and/or was implemented in our case study.

We would like to point out a few patterns that would be difficult, impossible, or irrelevant to implement using Rich Services/Erlang. The first is a dead letter channel, which handles messages that the messaging system cannot deliver. At the moment undeliverable messages are dropped and the system designer cannot specify actions for those messages. It is impossible to implement a dead letter channel. The file transfer pattern specifies how software systems interact using shared files. This pattern is entirely irrelevant for software systems implemented as Rich Services and interacting through Service Data Connectors.

VII.A Messaging Endpoints

Messaging Endpoints

Messaging systems provide an API that clients communicate with. But the client software must still use some custom piece of code that connects itself to the messaging system. This piece of code is a messaging endpoint.

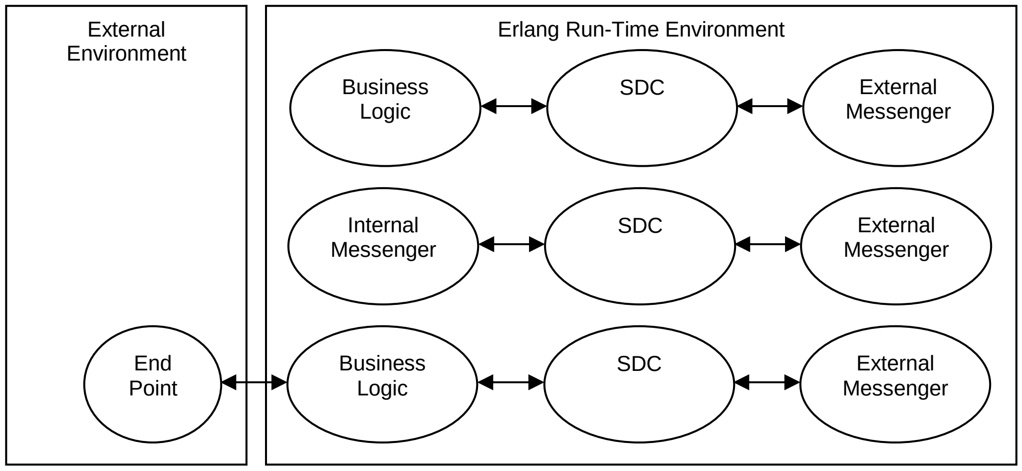


Figure 23: Example Rich Services/Erlang Message Endpoint

The first two rows in Figure 23 show pure Erlang implementations – application logic is code entirely written in Erlang. In these configurations the SDC provides the services of a message endpoint. It connects the Rich Service to the external Messenger.

In certain situations external operating system processes implement the Rich Service's application logic. The third row in Figure 23 demonstrates this situation. Logically the Business Logic component and SDC belong to the messaging system. A messaging endpoint is needed to connect the external environment to the erlang run-time system and the Rich Services/Erlang messaging layer.

Erlang Ports provide the primary mechanism for communicating with non-Erlang processes. Unfortunately they are low-level and primitive, but a number of more sophisticated libraries hide their use. These libraries support the construction of messaging endpoints that know how to communicate with Rich Services/Erlang and include the jinterface, erl_interface and Erlang IDL compiler libraries.

VII.B Message Construction

Command Message

Command messages invoke functions in remote applications. These are easily created and used in Rich Services/Erlang systems. Our chat system uses command messages in multiple interfaces. As an example, consider the message for resetting a session timer for a user in the session manager. The message causes the session manager to invoke a method in the Business Logic component that resets the session timer for the specified user.

Document Message

Document messages transfer data between applications. These are easily created and used in Rich Services/Erlang systems. Our chat system uses documents messages to push chat messages between users.

Event Message

Event messages move event notifications between applications. These are easily created and used in Rich Services/Erlang systems. Our chat system uses event messages to log out users when they timeout. When a user's session timer expires, the session manager sends a message to the *chat_server*. The Rich Infrastructure Services consume the messages and notify the user and person the user is chatting with of the timeout.

Request-Reply

Request-reply messaging describes simple two-way communication between two applications and includes three different patterns. Messaging RPC mimics remote procedure calls. The request is a command message and the reply is a document message with a return value or exception. Messaging Query is a type of remote query. The request is a command message and the reply is the result. In Notify/Acknowledge the request is an event message and the reply is a document message to acknowledge the event.

Our chat system implements the first two messaging patterns discussed. The account manager uses messaging rpc to add user accounts. The friendship manager uses messaging query to return the friend list of any existing user account.

Notify/Acknowledge could be easily implemented as well. These patterns are implemented using finite state machines within the Service Data

Connectors. The designer of the Rich Service can implement any messaging pattern described by a finite state machine.

VII.C Message Channel

Applications communicate through various statically defined and controlled connections within the messaging system. These connections are generally referred to as channels or logical addresses. Channels support multiple communication patterns, a few of which this paper discusses in this section. By default the Rich Services/Erlang library implements a messaging bus channel, but logical implementations for a number of other channel strategies could be created using a combination of Rich Application Services, Routers and Rich Infrastructure Services.

Publish-Subscribe

With publish-subscribe channels zero or more subscribers consume messages from a publisher. There is generally one input channel and multiple output channels, one for each subscriber. Messages are duplicated and sent to multiple receivers; each channel receives a copy.

Our case study does not use a publish-subscribe channel, although an implementation for one is straight forward. Figure 24 through Figure 26 show the basic configuration and messages required to implement publish-subscribe. The Channel Coordinator drives the publish-subscribe implementation. The Register Channel, Subscribe and Publish messages support the entire scheme. Register Channel allows publishers to create channels. Subscribe allows recipients to subscribe to those channels. Publish allows publishers to push out messages that the Channel Coordinator forwards to registered receivers.

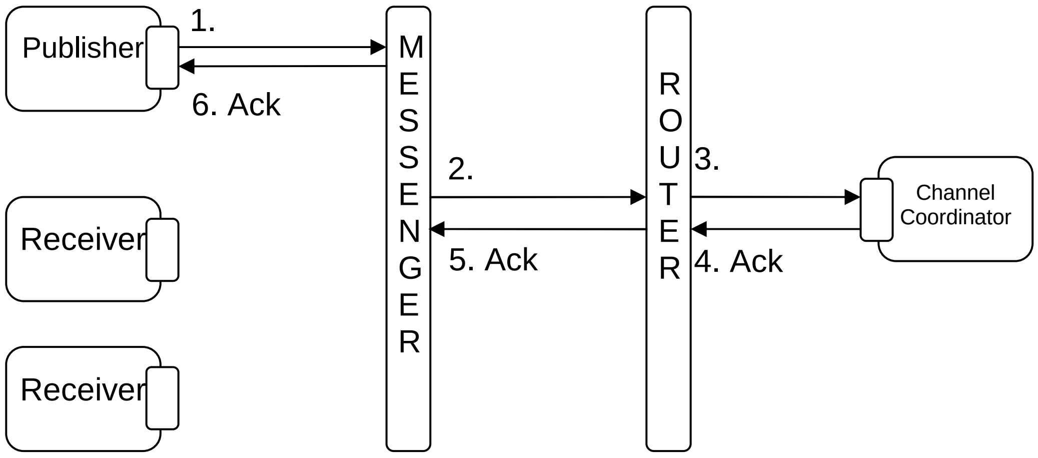


Figure 24: Publish Subscribe Register Channel Interaction

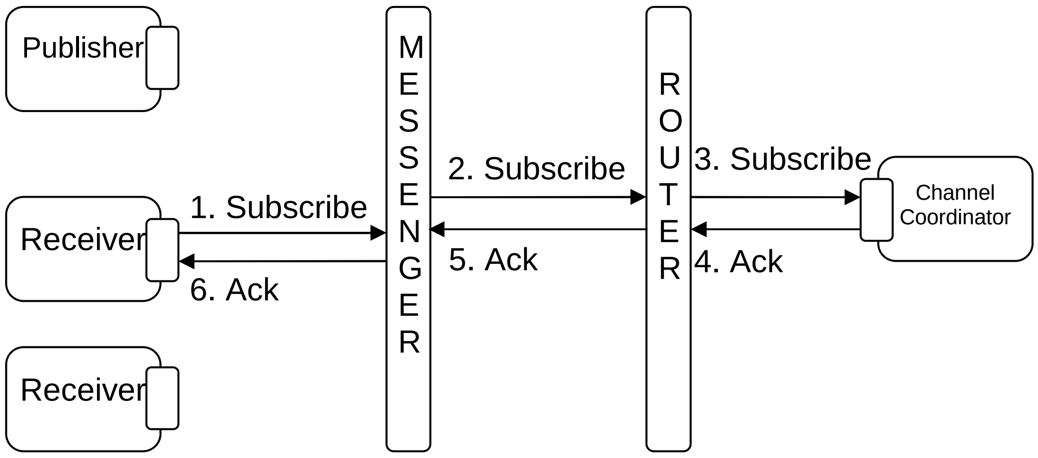


Figure 25: Publish Subscribe Subscribe to Channel Interaction

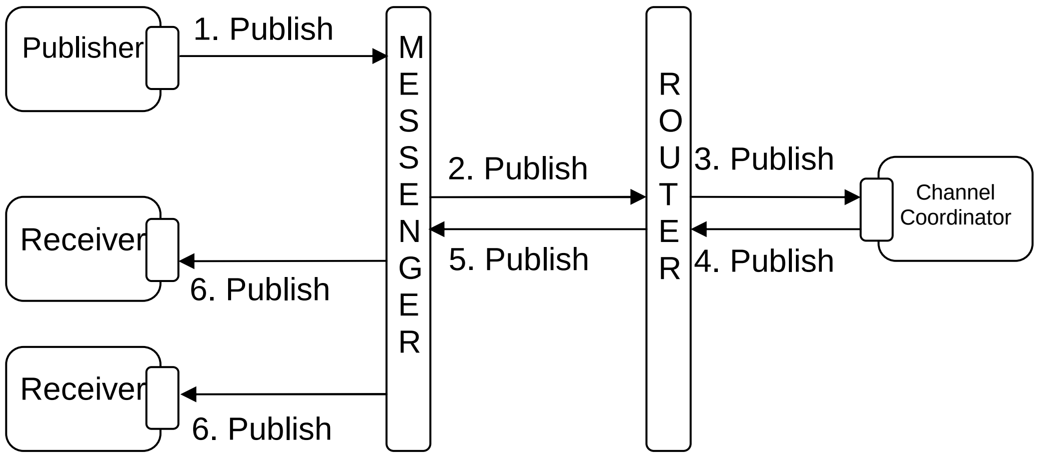


Figure 26: Publish Subscribe Publish to Channel Interaction

Invalid Message Channel

Sometimes a receiver may receive a message it does not recognize. In Rich Services/Erlang, messages that do not match at least one of the conversations defined in a SDC are consumed and dropped by the receiver. In certain situations these lost messages should be handled in some specific way, especially to help solve errors in the messaging system. An Invalid Message Channel accepts these messages for such processing.

Our case study does not implement an Invalid Message Channel, but one could easily be done. One possible implementation includes a RAS that accepts and processes invalid messages. Each participating receiver requires a special conversation that catches all unexpected messages. The receivers and consumer must be connected using some channel implementation as well. The easiest would be to statically name the consumer of invalid messages and then have all invalid messages sent directly to it by the message receivers. Of course a change to the invalid message consumer interface or address would require changes in every receiver. A slightly more complicated scheme could use the router and a RIS to create a virtual channel that redirects the messages to any physical consumer.

VII.D Message Routing

Content-Based Router

Content-based routers use the content of messages to determine their appropriate destination channel. They are purely static and must be changed as recipients change. Our case study does not use content-based routing to select destination Rich Application Services. But Erlang pattern matching easily supports content-based routing logic in the RISs.

Dynamic Router

With Dynamic Routing recipients notify the router of their presence and provide a set of conditions under which they will accept a message. One could define special conversations that support dynamic routing, possibly by saving information in a database. These conversations match control messages that recipients use to send the conditions under which they will accept messages. In other words, the recipients send messages directly to a routing policy in the router. The policy would then use the saved routing conditions to select the destinations of messages.

Splitter

A Splitter divides single message into multiple messages that are processed by multiple, usually different consumers. Think of an internet purchase filled with multiple items. Each item may be of a particular type, containing different pieces of information. Each type of item may be processed by a different application. But the order is sent to the messaging system in one message. A splitter could divide the order into separate messages, one for each processing application.

Our case study does not implement a splitter, but one is easily added to a Rich Services/Erlang system. Rich Infrastructure Services can place any number of messages into the router. To create a splitter a RIS must collect messages in the router and split their contents into multiple messages. It can then place these new messages back into the router.

Aggregator

Aggregators collect multiple related messages and combine them into one message. Our case study does not use an aggregator, but one is easily implemented using Rich Infrastructure Services. The aggregator needs a way to capture related messages in the same conversation finite state machine instance when the messages may not have the same Tag. This can be done by first defining a set of messages in the SDC interface that bind to one finite state machine. If the conversation is defined as open then a single executing conversation constantly consumes and processes every instance of these messages. Once the state machine has captured the necessary set of messages it can combine their contents into one message and place it back into the router.

VII.E System Management

Detour

Sometimes intermediate processes should be performed on messages as they travel between endpoints (for validation, testing, debugging). These processes are referred to as detours. Routing policies are a core component in Rich Services and naturally fill the detour role. One can use routing level policies to implement any concern addressed with detours.

Wire Tap

Wire taps use a recipient list to duplicate a message and publish it to multiple output channels. Duplicate messages are consumed and processed for the purpose of inspecting, testing, monitoring, etc. In Rich Services/Erlang a wire tap could be implemented in multiple ways, depending on where the message duplication should occur. One could design two routers, one for the intended destination and one for the wire tap. The Messenger would match a message to both routers, giving each a copy. One could design a RIS that creates a copy of the original message and sends it to additional destinations. In either case the extra processing should be encapsulated within a Rich Application Service.

VII.F Message Transformation

Content Enricher

Content Enrichers use external data sources to supplement the information contained within messages. This capability may be needed for a variety of reasons. Usually the publisher does not contain all of the information required by the receiver.

Our case study does not implement a Content Enricher. Of course one is easily implemented using routing level policies. One can imagine at least three different configurations for a content enricher. Figure 27 through Figure 29 show each of these.

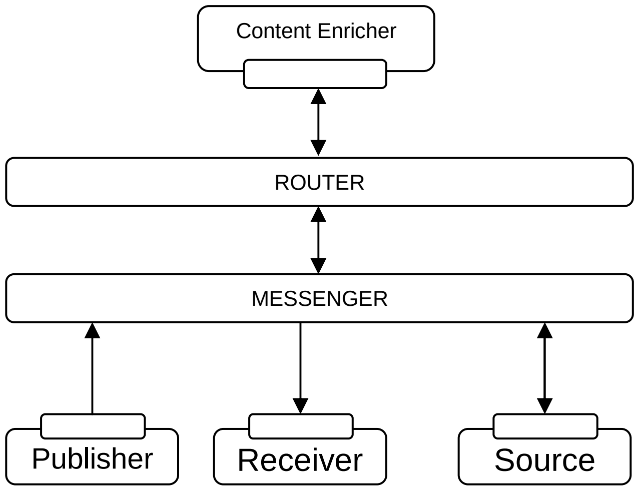


Figure 27: Example Rich Services/Erlang Conent-Enricher Implementation

Figure 27 shows a RIS and RAS that implement the content enricher. The RIS intercepts the message and requests the extra information from the outside source RAS by sending it a message. The source must send a response back. The RIS can then construct a new message with the combined information and send the new message to the receiver.

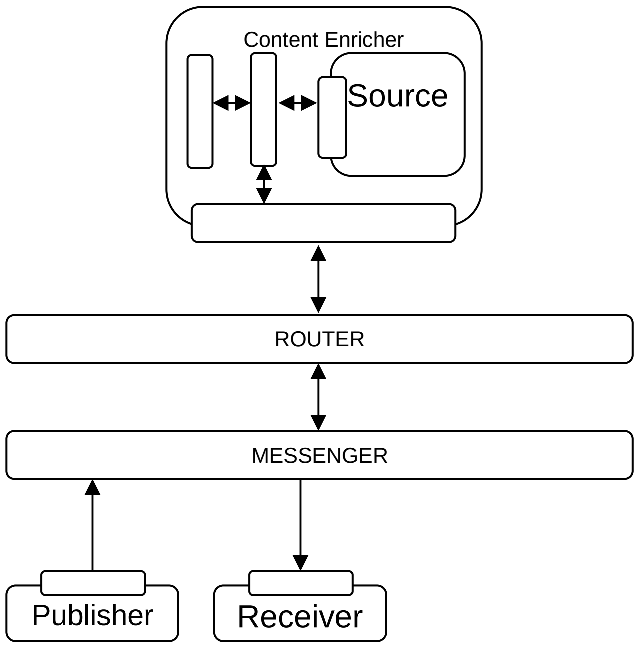


Figure 28: Example Rich Services/Erlang Conent-Enricher Implementation

Figure 28 shows a second implementation for a content enricher. This time only a RIS is used. The RIS encapsulates Rich Application Services that provide the supplemental content. The conversation handling the enrichment sends an internal messages and collects internal responses. It then forms a new message with the additional content and republishes it into the external router.

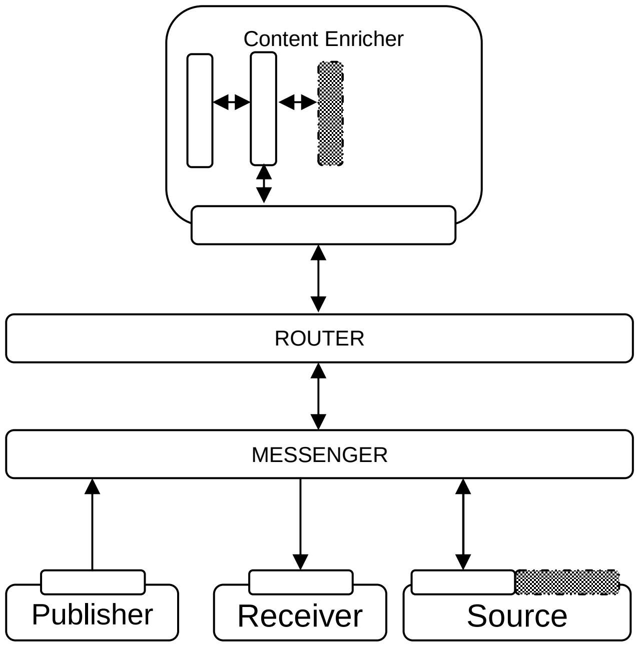


Figure 29: Example Rich Services/Erlang Conent-Enricher Implementation

Figure 29 shows a third configuration for implementing a content enricher. It uses some more interesting features available within Rich Services/Erlang. Rich Services can ask a statically addressable Rich Service to open a SDC that communicates with its own internal messenger or router. The source could be a peer RAS of the publisher and receiver. The content enricher RIS could open a private SDC to the source that connects to its own internal messenger. The advantage is that the high level communication infrastructure is not aware of the interaction.

Content-Filter

Content-filters remove information from messages before they reach their destination. Our case study does not implement a content-filter, but one is easily created using routing level policies. The designer must simply create a RIS that collects the messages and constructs a new message with only the required content. It then can publish the new message to the router

Chapter VIII Chat System Case Study

We evaluated Rich Services/Erlang through the development of a distributed chat system. Specifically the chat system is used to evaluate Rich Services with respect to its benefits as presented by the Rich Services creators (such as separation of concerns and system development process). Furthermore the case study allowed us to look at the complexity of implementing Rich Service using Rich Services/Erlang. The evaluation is presented in Chapter IX of this paper.

A Rich Framework Service consisting of four Rich Application Services, one Router and two Rich Infrastructure Services implement the Chat Server functionality. A fifth complex Rich Application Service, the Chat Client, introduces a hierarchical depth of two to the entire Rich Service and connects users through a command line interface and multiple chat windows. Figure 30 the high level architecture view of the chat system.

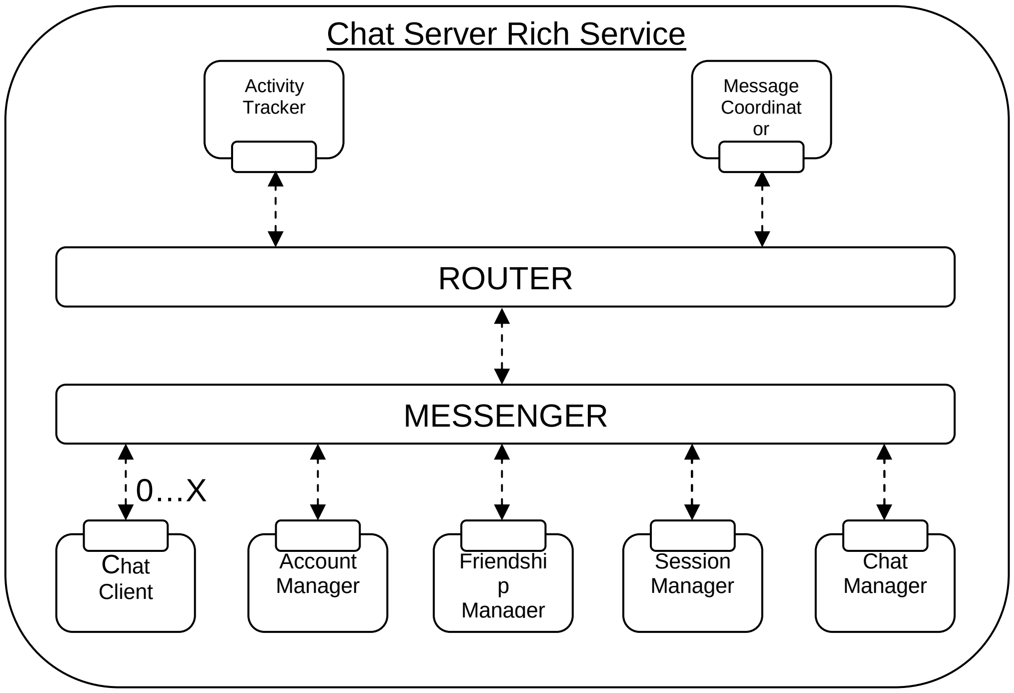


Figure 30: **Rich Services Chat System (Case Study) Design**

VIII.A Chat System Rich Infrastructure Services Descriptions

Chat System Session Activity Tracker

The session activity tracker intercepts certain user messages and generates a message for the session manager to register user activity. It guarantees that a user will not timeout if they are actively using the chat system.

Chat System Message Coordinator

The Chat Client must set the destination of chat system messages to *[[chat_server]]*. There is not any particular entity that this address belongs to; instead the routing logic and policy selection logic determine how messages sent to this address are processed.

Currently the Chat System Message Coordinator Rich Infrastructure Service accepts all messages sent to *[[chat_server]]*. It consumes the chat system messages and coordinates the use of the chat system's Rich Application Services to provide the features expected from those messages. Table 14 shows a list of the messages that the chat system recognizes (these should be placed in a message's content field).

Table 14: Chat System Messages

Chat System Messages
[From, To, [startFriendship, FriendName]]
[From, To, [add_user_account, UserName, Password]]
[From, To, [delete_user_account, UserName, Password]]
[From, To, [findAllFriendships]]
[From, To, [logIn, UserName, Password]]
[From, To, [logOut]]
[From, To, [endFriendship, FriendName]]
[From, To, [startChat, FriendName]]

The messages are all a list with three fields: From, To and Request. The *From* field is always one of the patterns {user, UserName}, {user, anonymous} or server. The *To* field is always one of the patterns {user, UserName} or server. The *Request* contains the request.

VIII.B Chat System Rich Application Services Descriptions

Account Manager

The Account Manager manages user accounts in the chat system. Besides the basic capabilities of adding and deleting user accounts, one can also search for an account name; these basic features help to support additional chat system level policies such as session management.

Session Manager

The Session Manager helps maintain sessions for logged in users. Capabilities include explicitly starting or stopping session when asked, generating session timeouts, resetting session timers for users and overriding session timer values. The session manager is extremely important for timing out inactive users.

Friendship Manager

This service administers the friendships that users create between themselves. The interface is fairly simple. It allows for creating and removing friends as well as viewing one’s friendship list.

Chat Manager

The chat manager provides administrative services over the chats that users open amongst themselves. Most critically it saves important state for chats, including the participants and the chat client addresses of the participants. This state supports starting, ending and timing out chat sessions.

VIII.C Chat Client RAS

The chat client allows users to connect to the chat system. It is a complex RAS. Internally the chat client Rich Services includes one Command Window RAS and zero or more Chat Window Rich Application Services. The former provides a graphical user interface that the user needs to communicate with the chat system while the latter provides a graphical user interface for each chat the user joins.

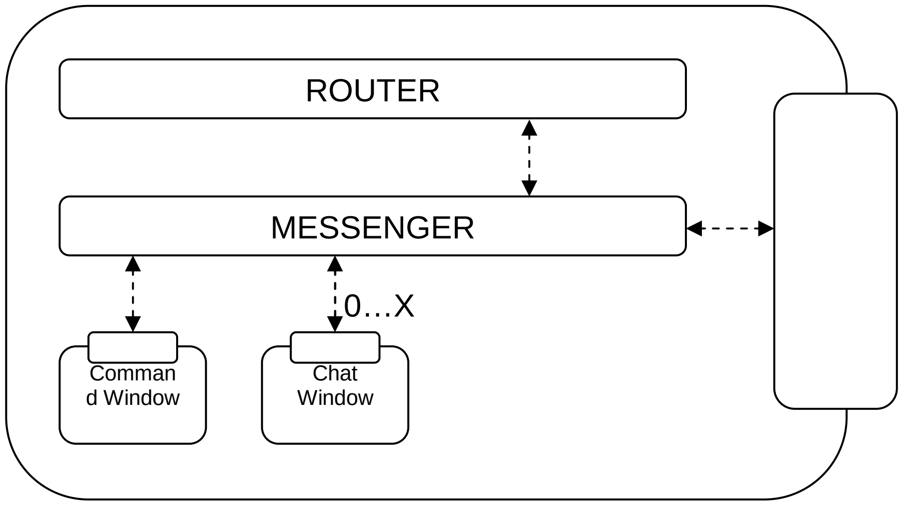


Figure 31: Rich Services Chat System Chat Client RAS Design

The internal interface of the Chat Client RAS converts messages from the Command Window RAS and Chat Window RAS into chat system messages. It must implement one finite state machine for each chat system message. The Command Window RAS and Chat Window RAS are unaware of the chat system messages; instead they are only aware of the internal SDC interface.

Chat Client Command Window

The Chat Client Command Window is a *simple* RAS. Users submit text commands through a graphical user interface that communicates with the Business Logic component of the Chat Client Rich Service. The GUI serves as an example of an external source that connects to a Rich Service directly through a Business Logic component.

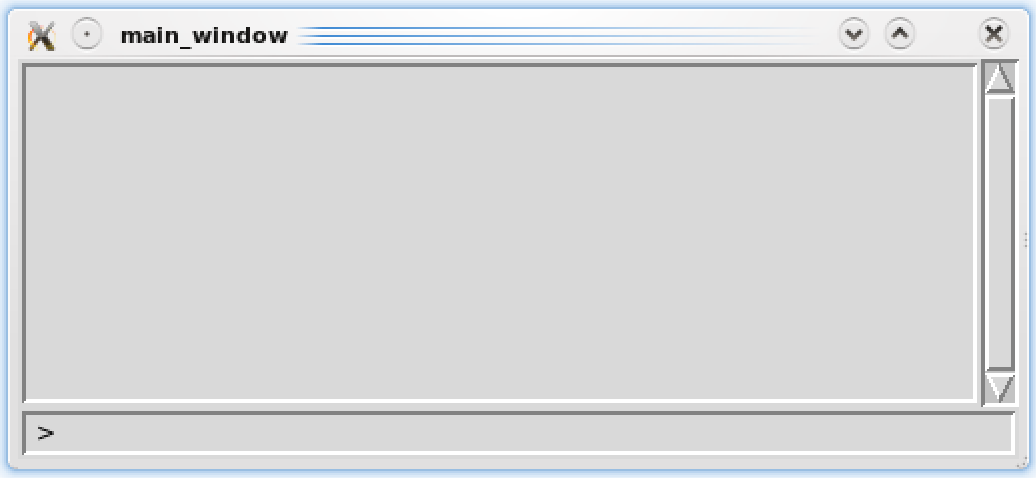


Figure 32: **Rich Services/Erlang Chat Client Command Window RAS GUI**

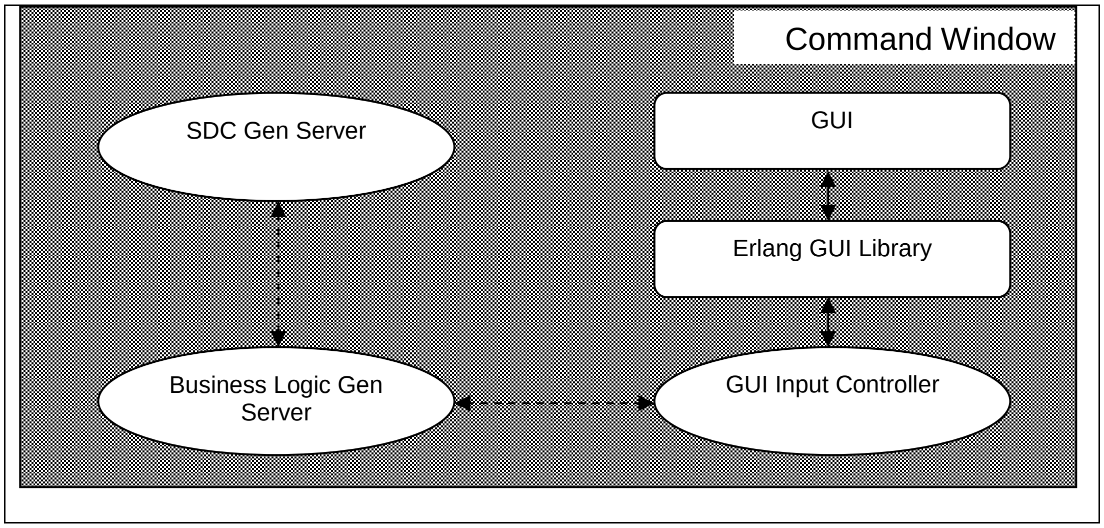


Figure 33: **Rich Services/Erlang Chat Client Command Window RAS Run Time Implementation**

Figure 33 shows the run-time implementation of the Command Window RAS. User input flows from the gui command line and through some Erlang libraries to a process that mediates communication between the gui and the Business Logic component of the Command Window. This process has the reference of the generic server and the generic server has the Pid of this process. String output can flow from the Rich Service to the gui as well.

The Business Logic component forwards user input to the SDC (listed in Table 15). The SDC matches user input to a command in the set of chat system commands listed in Table 15. Successful matches cause the SDC to generate the messages that call into the internal interface of the Chat Client RAS. Unsuccessful matches result in an error message being displayed in the GUI.

Table 15: Rich Services/Erlang Chat Client Command Window RAS Commands

Chat Client Command Window GUI Commands
command create user account USERNAME PASSWORD
command delete user account USERNAME PASSWORD
command login USERNAME PASWORD
command logout
command add friend FRIENDNAME
command remove friend FRIENDNAME
command view friend list
command start chat FRIENDNAME

Chat Client Chat Window

The Chat Window RAS collects and forwards string input from the user and displays string input from the person the user is chatting with. It is the chat window. The internal interface of its SDC implements a finite state machine for collecting the text input and publishing it to the internal SDC interface of the Chat Client RAS. The external interface of the SDC collects messages sent to it from that Chat Client RAS SDC and forwards the string input in their contents to the Business Logic component, which passes it to the GUI controller. These inputs are the chat messages from the person the user is chatting with.

VIII.D Chat System Router Selection Rules

The router selection rules for the chat system are fairly simple because one router processes all messages. One router is sent all messages which are all processed by at least one RIS (depending on the message). The three router selection rules are listed in Figure 34.

```
{Type, Tag, [[chat_server]], Src, RouterSelection, PolicySelection, Content}

{Type, Tag, Dest, Src, _, _, [From, To, [message, Message]]}

{Type, Tag, [[{ris,_}]], Src, _, _, Content}
```

Figure 34: Rich Services/Erlang Chat System Router Selection Rules

VIII.E Chat System Policy Selection

Chat Clients can send any message from the set of chat system messages; these are addressed to [[chat_server]]. This is not a valid RAS address. A valid address is not used because there is not a physical RAS that administers the features of the chat system. Instead the chat system uses a policy RIS (Chat System Coordinator) to intercept these messages and to coordinate the chat system RASs.

Table 16 shows the policy list in the router; if a message will be sent to both policies the chat_server_activity_tracker is always the first to receive the message. Table 17 shows the message patterns that the chat_server_activity_racker matches and Table 18 shows the patterns that the chat_server_services_coordinator matches.

Table 16: Rich Services/Erlang Chat System RIS List

Chat System RIS List
[chat_server_activity_tracker, chat_server_services_coordinator]

Table 17: Rich Services/Erlang System Activity Tracker RIS Policy Selection Rules

Policy Selection Rules For chat_server_activy_tracker

{_,_,_,_,_,_,[{user, anonymous},_,_]}
 	
{_,_,_,_,_,_,[{user,_},_,_]}
 	
{_,_,_,_,_,_,[timeout, UserName, Address]}
 	
{_,_,[[{ris,chat_server_activity_tracker}]],_,_,_,_}

Table 18: Rich Services/Erlang Chat System Message Coordinator RIS Policy Selection Rules

Policy Selection Rules for chat_server_services_coordinator
{_,_,_,_,_,_, [From, To, [login, UserName, Password]]}
{_,_,_,_,_,_, [From, To, [logOut]]}
{_,_,_,_,_,_, [From, To, [add_user_account, UserName, Password]]}
{_,_,_,_,_,_,[From,To,[delete_user_account, UserName, Password]]}
{_,_,_,_,_,_, [From, To, [findAllFriendships]]}
{_,_,_,_,_,_, [From, To, [startFriendship, FriendName]]}
{_,_,_,_,_,_, [From, To, [endFriendship, FriendName]]}
{_,_,_,_,_,_, [From, To, startChat, FriendName]}
{_,_,[[{ris,chat_server_services_coordinator}]],_,_,_,_}

Chapter IX Evaluation

IX.A Rich Services

IX.A.1 Separation of Concerns

System Design and Implementation

The chat system case study is a simple system, and we therefore did not explicitly follow the design process suggested by the Rich Services authors. But the resulting system is conveniently divided into a horizontal and vertical set of services that meet the basic functionality of any chat system. All session management logic was implemented by the Session Manager and all user account logic was implemented by the Account Manager. The physical separation of services by concerns made the system more manageable to implement. Unrelated features were completely isolated from each other. Different Rich Applications Services could be implemented concurrently by multiple people and in complete isolation, without worry that the work of one person might break the work of another.

Integration of Cross-cutting Concerns

The chat system benefited greatly from the placement of cross-cutting concerns in the Router. For example, user timeouts involved the Session Manager, Chat Manager, Friendship Manager and Chat Client Rich Application Services. Each only implemented its own logic for dealing with timed out users without knowledge of the others. A policy RIS (Message Coordinator) coordinated the activity of these services to implement the system feature for timing out users. Furthermore, changes to this feature could be changed in the single RIS without affecting any of these RASs.

Debugging

The division of Rich Services according to the separation of concerns makes debugging and validation of systems more efficient. Specifically, testing the correctness of a Rich Service when its internal structure changes requires that only its Service Data Connector's interface be validated. If the interface returns the correct results for all inputs then one must only concern himself with validating the correctness of the level that the Rich Service connects with.

Debugging of the chat system went well due to the separation of features into Rich Application Services and Rich Infrastructure Services. By placing related features of the chat system into different Rich Application Services, one could be confident that working Rich Application Services would not be affected by logic implemented in other Rich Application Services. Changes to the session manager could not break the account manager.

A properly design Rich Service makes debugging more manageable. The process of debugging could be divided into several related tasks, one for each RAS and one for the Rich Framework Service coordinating the chat system Rich Application Services. This makes the assignment of debugging amongst multiple persons efficient. Furthermore each debugging task need only to worry about the relevant concerns of the Rich Service being debugged.

IX.A.2 One-for-One Design to Implementation

The one-for-one design to implementation property of Rich Services make the design and implementation process more efficient. The logical and physical organization of our chat system was exactly specified using the architecture. The specification of the Rich Services in our chat system was used by directly defining the SDC interfaces. The interactions of the Rich Services in our chat system were defined by directly placing Rich Service components one-for-one in message sequence charts. The programming of the code for our chat system corresponded exactly to the components of the Rich Services.

IX.B Erlang

Concurrent Programming

Writing concurrent programs is simple in Erlang. The programmer needs to know how to spawn processes (which can be done in one line), send messages (which can be in one line) and receive messages (which requires only a two line structure). A one page introduction of these three concepts is enough to begin writing concurrent software [1].

Joe Armstrong claims concurrent software programs written in Erlang are efficient. Joe spawns 300,000 Erlang processes in 74 microseconds and 20,000 processes in 9 microseconds. In one published benchmark, Erlang consistently sends messages 6x faster than Java [23].

Distributed Programming

The distributed programming primitives are directly built into the concurrent communication primitives. Programmers only need to assign nodes names that can be used across a network, set up the correct permissions on host machines and register atom names with processes where appropriate. The distributed communication primitives are simple and do not require a steep learning curve.

Fault-Tolerance Support

The fault-tolerance primitives are simple and easy to work with. The supervisor behavior, process links, process monitors and node monitors seem sufficient to create fault-tolerance strategies for any set of executing processes.

Documentation

The online documentation covers Erlang in great breadth; unfortunately, programming more complicated tasks could be daunting. The man pages completely specified the available modules but often lacked examples. The reference manual provided examples for only the core features.

Community Support

The Erlang website did not have a community forum and web searches were usually lacking. Expect to spend the majority of your time reading through documentation on the Erlang website.

Line Count

Depending on the application Erlang requires 4-10x less lines of code [23]. Erlang reduces the line count for programs that require concurrent, distributed features. For example, processes send messages to other processes using a single line.

Single assignment increased the line count in code blocks where the values of the fields in a tuple change multiple times. When a tuple with many fields was constructed doing so in a single at a time line could mean long lines. One could either break the line into multiple lines, sometimes reducing the readability. One could make one change at a time, assigning each result to a new variable. In either case one often wished for a more object oriented solution.

Many modules in the Rich Services/Erlang library and chat system have multiple lines that are at least twice as long as lines in other languages. Rich Services/Erlang messages consist of seven fields. Patterns written to match particular messages often extend beyond 100 characters. Breaking the patterns into multiple lines increases the difficulty of reading and understanding the code.

Debugging

Erlang/OTP provides a debugger but it was not used throughout the research process presented in this paper [7]. We debugged our libraries using print statements to display information in a terminal window. The debugging process was often frustrating.

Each Rich Service executes as multiple processes and if more than one of these processes printed to the same terminal, the output would be interleaved. Picking out the desired lines could take time. The print statements in working sections of code were removed to reduce the clutter in the terminal windows. Unfortunately when these sections were later modified the print statements would need to be reinserted, increasing the inefficiency of debugging.

In Erlang function callers always handle function errors. Many errors messages included information compounded through multi-level function calls. These could be difficult to understand. In addition the online documentation did not completely specify the meaning of errors, making it difficult to determine why functions would crash.

Porting and Interfacing

The case study does not interface with an external operating system process. But the Rich Services/Erlang library was used to successfully create a Rich Services system that integrated a physical wireless sensor network. The sensor devices in the network communicated with a base station device plugged into a computer via USB. A Java program communicated with the base station via the USB Port and used the JInterface library to connect to Erlang. JInterface allowed the the Java program to call into an Erlang generic server as easily as a native Erlang program. JInterface simplifies the process of connecting Java and Erlang via message passing.

IX.C Rich Services/Erlang

Implementation Complexity

The code of a Rich Service is spread through multiple files. One must understand the organization and purpose of these files as well as how Erlang applications work. Rich Services/Erlang requires a relatively steep learning curve. Fortunately when the process for creating a Rich Service is understood they seem to form quickly.

SDC Implementation

Using the finite state machine behavior to implement an SDC interface makes defining interactions extremely easy. One advantage is that an SDC conversation can coordinate its internal Rich Application Services; this can be more convenient than implementing routing policies to implement the same logic. In our chat system the finite state machine conversations made the coordination of multiple Rich Application Services simple. Finally, the finite state machine enables the creation of a variety of enterprise integration patterns (such as the aggregator).

Business Logic Implementation

Implementing simple application logic in the Business Logic component was often an annoying task during the development of the chat system. But the separation of interaction logic and application logic generally improves the readability of both. In addition, the Business Logic component handles one request at a time on a first come first serve basis and saves state for the Rich

Service. Programmers can leverage this feature to provide sequential, uninterrupted access to the Rich Service state.

Router Implementation

Allowing the specification of multiple routers improves the potential of a Rich Services system. They support the integration of various enterprise integration patterns. In addition the implementation of multiple, unrelated routing strategies reduces the complexity of individual routers and improves opportunities for router reuse in different Rich Services systems.

The use of Rich Infrastructure Services within the router is not efficient and difficult to understand. Creating a workflow from a set of Rich Infrastructure Services cannot be done using one mechanism but rather requires creating Rich Infrastructure Services that are aware of the work flow and manipulate knowledge of policy selection in the router. This is not appropriate.

Messenger Implementation

Programming for the Messenger requires minimal effort; the router selection module is the only significant programming needed. The features implemented are less than desired. Unfortunately the Messenger only communicates with Rich Services implemented using Rich Services/Erlang. It should be extended to communicate with other services (such as web services).

The type field of the message could support communication protocols other than those based on Erlang. At the moment this has not been implemented. For example, existing web services are useful resources. Web services do not register with the Messenger. The type field could force the content field to contain the necessary information to communicate with a web service. The Messenger needs modifications that support communication with the web services.

Chapter X Thesis Summary

X.A Conclusion

This paper introduces Rich Services/Erlang, the first software library enabling the implementation of Rich Services systems. It presents a description of the Rich Services architecture and an overview of the functional programming language Erlang; we show how the Rich Services/Erlang library is implemented and how designers use it to create Rich Services systems. The chat system case study and the discussion of enterprise integration patterns case study evaluate and validate the usefulness of the Rich Services/Erlang library.

Rich Services/Erlang provides Rich Services designers with a few convenient features. One can easily design multiple Service Data Connectors per each service; affording the opportunity to separate interfaces according to concerns. Similarly, designers can create Rich Services with multiple routers. Most important the library allows the implementation of actual Rich Services systems.

Erlang brings multiple features to the Rich Services/Erlang library. Most important Erlang supports efficient and simple primitives for writing concurrent, distributes software. Rich Services/Erlang leverages these features to implement services that run in physically separated locations and are protected via fault-tolerance primitives. In addition Erlang pattern matching and finite state machines provide a powerful combination for managing the interaction of services through the SDC.

Our enterprise integration patterns case study shows the flexibility for Rich Services/Erlang to implement systems using common strategies. The chat system case study evaluates the Rich Services/Erlang library according to a set of criteria. In both instances we find that Rich Services/Erlang supports the creation of flexible and complex distributed software systems.

Our evaluation discusses Rich Services, Erlang and Rich Services/Erlang. In one statement we can say that Rich Services/Erlang proves to provide a positive opportunity for creating Rich Services systems. The library, the chat system case study and the enterprise integration case study support the published arguments for Rich Services and show potential for additional opportunities, such as the implications of multiple routers and multiple SDCs.

X.B Future Work

Evaluation of Fault-Tolerance Capabilities

Every Rich Services implemented using the Rich Services/Erlang library has multiple process links and node monitors. The library implements a specific crash recovery strategy. A number of systems and tests need to be designed to validate the strategy implementation.

Dynamic Reconfiguration

The Rich Services/Erlang library was developed with the intent of providing dynamic code loading and dynamic reconfiguration. The specification for Routers, Rich Application Services and Services Data Connectors should be changeable at run-time. Currently the configuration modules of every Rich Service compile at start time. The logic for compiling these files and dynamically changing the configuration of any Rich Service is programmed into the library code but not tested.

Messenger Feature Set

The Messengers needs to support different communication protocols (such as soap) so that Erlang Rich Service can communicate directly with existing services (such as web service) through the Messenger.

Federation of Single Rich Services onto Multiple Nodes

Every Rich Service runs on its own node, but if the node crashes and cannot be restarted then access to it is lost. Erlang could easily support the federation of a single Rich Service onto multiple nodes to protect against this situation. We would like to duplicate the service among multiple nodes to increase redundancy and thereby increase availability.

We need to address a few different concerns for management of the node cluster for each Rich Service. The first concern is for how to specify the location of the nodes in the cluster. The second concern is for how to manage the cluster (starting, stopping, reconfiguring). The third concern is for how to deal with the flow of messages through the cluster. We feel that these concerns are easily dealt with in changes to the Rich Services/Erlang library.

A specification for the cluster would be done using a configuration setting. This specification could be added to the configuration file to support static settings. It could be passed to the service at start up, or it could be specified dynamically at run time. In each case the specification is saved using a configuration setting.

The configuration for the cluster is consumed by some mechanism for starting, stopping and dynamically reconfiguring the cluster. This mechanism should also contain the logic for dealing with crashed nodes. A sub-concern is how to delegate authority and control across the cluster. We suggest that configuring the cluster using a ring and electing a leader would work well. If the leader crashes the instance of the Rich Service monitoring the crashed instance would become the leader. This mechanism guarantees that only one point of control (or Rich Service) takes action to manage the cluster, simplifying the implementation. The logic for managing the cluster should be built into the Rich Service generic server process.

A third concern is for how to handle message flows through the cluster. We suggest one of two implementations. Messages can be duplicated and sent to each instance of the Rich Service. Or messages can flow through one physical path. In either case a single path is chosen as the valid path. The advantage of the second is that if a link in the valid path crashes, the messages flow does not break because a redundant link can immediately join the valid path. The second option has the advantage of simplicity but on crashes the message flows may be lost. The logic for handling the message flows should be built directly into the Messenger.

Messenger AMQP Implementation

We would like to explore the use of an existing messaging system, such as RabbitMQ, as an implementation for the Messenger in Rich Services/Erlang. RabbitMQ is a promising substitution for the Messenger as it is built using Erlang/OTP and supports Erlang based clients [24]. To support RabbitMQ the SDC and Router implementations must be changed. Both must connect to a RabbitMQ server. RabbitMQ provides libraries to connect both as a client to the server. In addition to establishing the connection the logic in both components must be changed to receive and send messages to and from RabbitMQ rather than an Erlang generic server.

In addition to connecting the Router and SDCs to RabbitMQ there are a few more concerns to be addressed. We need a solution for logically implementing the Rich Services architecture. Should the Routers and Rich Application Services connect to the server as peers? In this case the Rich Infrastructure Services connect directly to the Routers and the RabbitMQ server sends messages to the appropriate Router(s) before sending them to their destinations. In another instance the Rich Infrastructure Services, Routers and Rich Application Services all connect to the server as peers, which handles every physical transmission of a message and directs the flow between the destinations, routers and Rich Infrastructure Services.

The address and naming scheme used in Rich Services/Erlang may need adjustment to support RabbitMQ, The configuration parameters will be extended to save the state for connections to a RabbitMQ server. The server itself must somehow be packaged with the RabbitMQ library.

A RabbitMQ Messenger implementation will not always be the best option. The server is not lightweight compared to the current implementation. For Rich Services that are started and stopped regularly it would be convenient to support a version of the current Messenger. At the very least a strategy for allowing multiple Rich Services to share the same server would help to reduce the overhead of using RabbitMQ.

Router RIS Selection and Application

To create workflows involving a set of Rich Infrastructure Services the router must be reengineered. Using Erlang finite state machines, similarly to the way they are used in Service Data Connectors, seems to be the best solution. A workflow is simply the application of a series of policies on the receipt of a message in the router. Incoming messages should be matched with a single workflow definition (an Erlang finite state machine module). The router and finite state machines can use the message tag field to correlate messages with active workflows, similar to the way the SDC handles a message flow through a conversation. This mechanism allows one to construct complex workflows through the use of finite state machines and removes the need to implement workflow logic in the Rich Infrastructure Services (which is currently required).

Potential Performance Issues

This paper does not present a quantitative evaluation for the performance of systems implemented using Rich Services/Erlang. But performance factors should be explored in the future. In general we would like to consider throughput through the Messenger, Router and Service Data Connectors as well as message latencies through these components. In addition the fault-tolerance features should be evaluated.

Throughput and latency through the various components is important. At the moment the Messenger is implemented using one process but to increase throughput the logic for selecting a message’s router and RAS destination is handled in a separately spawned process. The Messenger simply receives messages and spawns the correct process.

The Router is implemented using a single process. Policy selection and policy application are done using spawned processes started when the Router receives a message. We need to know the throughput through the Router as well as the latencies of receiving a message, selecting policies for a message, applying policies on a message or as a result of a message and forwarding a message back to the Messenger.

Currently the SDC interfaces are implemented using Erlang finite state machines and calls into the SDC spawn those finite state machines. The function of the SDC generic server process is to match incoming messages with the correct interface call and to send messages externally and internally. Besides finding the maximum throughput it would interesting to know what the upper bounds are for how many conversations can be active in a single instance of time.

The fault-tolerance features in Rich Services/Erlang should guarantee that any crashed process be restarted so that live Rich Services do not completely fail. These features have not been tested. The impact of crashed processes should be evaluated. In addition the restart strategies should be tested and new restart strategies should be implemented for comparative purposes.

Rich Service Nodes

In the version of Rich Services/Erlang described in this paper, each Rich Service runs within its own node. In Erlang the node is the heavyweight deployment concept. We can get away with running each Rich Service on its own node when they are realized as continuously running instances. But if one would like an approach where Rich Service are started when needed and stopped when not needed, then running them on private nodes will hurt performance

The library needs slight alterations to support running multiple Rich Services on a single node. Each Rich Service is implemented as an Erlang application. The module for this implementation needs to be changed so that instead of controlling the deployment of a single Rich Service, it manages the deployment of multiple Rich Services. In other words, starting the Rich Services application does not start a single Rich Service. It should start a central point of control that manages the deployment of multiple Rich Services. In addition the start logic for each Rich Service needs to be slightly changed; at the moment it starts a node for each Rich Service. This would no longer be required.

Appendix A Chat System Rich Service Interface Specifications

A.I Chat System Rich Infrastructure Services

Activity Tracker

SDC ID	one	
External Interface		
1	Message Pattern	[{user, UserName}, To, Content]
	Description	This conversation intercepts user messages to assist in resetting timeouts.
2	Message Pattern	[timeout, UserName, Address]
	Description	This conversation handles timeout message from the session manager.

Message Coordinator

SDC ID	one	
External Interface		
1	Message Pattern	[From, To, [startFriendship, FriendName]]
	Description	This conversation communicates with the Friendship Manager to add a friend for the requesting user.
2	Message Pattern	[From, To, [add_user_account, UserName, Password]]
	Description	This conversation coordinates the appropriate

		services to add a user account.
3	Message Pattern	[From, To, [delete_user_account, UserName, Password]]
	Description	This conversation coordinates the services that delete user accounts.
4	Message Pattern	[From, To, [findAllFriendships]]
	Description	This conversation coordinates the services that find all friendships for the user specified in the *From* field.
5	Message Pattern	[From, To, [logIn, UserName, Password]]
	Description	This conversation coordinates the appropriate services to log a user in.
6	Message Pattern	[{user, UserName}, To, [logOut]]
	Description	This conversation coordinates the services that log out the user.
7	Message Pattern	[From, To, [endFriendship, FriendName]]
	Description	This conversation coordinates the services that remove a specified friendship.
8	Message Pattern	[From, To, [startChat, FriendName]]
	Description	This conversation communicates with the Friendship Manager to add a friend for the user.

A.II Chat System Rich Application Services

Account Manager

SDC ID	one
External Interface	

1	Message Pattern	[add_user_account, UserName, Password]
	Description	Adds the user *UserName* with password *Password* to the user account database.
	Return Messages	[ok, [add_user_account, UserName]] [fail, [add_user_account, UserName, dbManagerFailure]] [fail, [add_user_account, UserName, timeout]] [fail, [add_user_account, UserName, Reason]]
2	Message Pattern	[delete_user_account, UserName, Password]
	Description	Deletes the user *UserName* with the password *Password* from the user account database.
	Return Messages	[ok, [delete_user_account, UserName]] [fail, [delete_user_account, UserName, dbManagerFailure]] [fail, [delete_user_account, userManagerFailure]] [fail, [delete_user_account, timeout]]
3	Message Pattern	[findUserAccount, UserName, Password]
	Description	Searches for the specified user and password in the user account database.
	Return Messages	[ok, [findUserAccount, UserName]] [fail, [findUserAccount, UserName, dbManagerFailure]] [fail, [findUserAccount, UserName, timeout]] [fail, [findUserAccount, UserName, Reason]]

Session Manager

SDC ID	one	
External Interface		
1	Message Pattern	[startSession, UserName, Address]
	Description	Stars a session for the specified user and saves the address of their chat client Rich Service.
	Return Messages	[ok, [startSession, UserName]] [fail, [startSession, UserName, Reason]]
2	Message Pattern	[stopSession, UserName]
	Description	Stops the session of the specified user.
	Return Messages	[ok, [stopSession, UserName]] [fail, [stopSession, UserName, Reason]]
3	Message Pattern	[resetTimer, UserName]
	Description	Resets the timer of the specified user.
	Return Messages	
4	Message Pattern	[checkUserActivity, UserName]
	Description	Tells the inquirer if the user is logged in or not.
	Return Messages	[ok, [checkUserActivity, UserChatClientAddress]] [ok, [checkUserActivity, inactive]] [fail, [checkUserActivity, Reason]]
Internal Interface		
	Message Pattern	[timeout, UserName, Address]
	Description	This message is generated internally. The interface implementation forwards the message externally so

		the chat system can deal with the user’s timeout.
	Return Messages	

SDC ID	two	
External Interface		
1	Message Pattern	[setTimeout, Timeout]
	Description	Sets the timeout length to *Timeout*.
	Return Messages	[ok, [setTimeout, Timeout]] [fail, [setTimeout, Timeout, Reason]]

Friendship Manager

SDC ID	one	
External Interface		
1	Message Pattern	[startFriendship, UserName, FriendName]
	Description	Starts a friendship where the user specified by *UserName* has request to be friend with the user specified by *FriendName*.
	Return Messages	[ok, [startFriendship]] [fail, [startFriendship, UserName, dbManagerFailure]] [fail, [startFriendship, timeout]] [fail, [startFriendship, Reason]]
2	Message Pattern	[endFriendship, UserName, FriendName]
	Description	Ends the friendship of the two specified users.
	Return Messages	[ok, [endFriendship]]

		[fail, [endFriendship, UserName, dbManagerFailure]] [fail, [endFriendship, timeout]] [fail, [endFriendship, Reason]]
3	Message Pattern	[findAllFriendships, UserName]
	Description	Returns a list of the user's friend's name specified by *UserName*.
	Return Messages	[ok, [findAllFriendships, FriendList]] [fail, [findAllFriendships, UserName, dbManagerFailure]] [fail, [findAllFriendships, timeout]] [fail, [findAllFriendships, Reason]]

Chat Manager

SDC ID	one	
External Interface		
1	Message Pattern	[addChat, Initiator, Initiatee]
	Description	Adds a chat to the database with the users specified by *Initiator* and *Initiatee*.
	Return Messages	[ok, addChat] [fail, [addChat, dbManagerFailure]] [fail, [addChat, Reason]
2	Message Pattern	[findChats, Participant]
	Description	Finds all of the chats that the user specified by *Participant* is involved in.
	Return Messages	[ok, [findChats, Chats]]

		[fail, [findChats, dbManagerFailure]] [fail, [findChats, Reason]]
3	Message Pattern	[removeChat, Initiator, Initiatee]
	Description	The message causes the chat start by *Initiator* and involving *Initiatee* to be removed.
	Return Messages	[ok, removeChat] [fail, [removeChat, dbManagerFailure]] [fail, [removeChat, timeout]] [fail, [removeChat, Reason]]
4	Message Pattern	[removeChats, Participant]
	Description	This messages causes all chats to be removed that the user specified by *Participant* is involved in.
	Return Messages	[ok, removeChats] [fail, [removeChats, dbManagerFailure]] [fail, [removeChats, Reason]]

Chat Client

SDC ID	one	
External Interface		
1	Message Pattern	[server, {user,UserName},[log_out, IChats, AChats]]
	Description	This conversation tells the chat client that the user had been timeout out. It sends a message for display at the command window and sends an exit message to all open chat windows.
	Return	

	Messages	
2	Message Pattern	[chatInvitation, HostName, HostSrc]
	Description	This conversation process chat invitations directed at the user. It attempts to open a chat window and if successful sends an accept message to the source of the initiating message. Otherwise its send a failure message.
	Return Messages	{acceptChatInvite} {chatInvitation, error}
3	Message Pattern	[server, {user, UserName}, [offline, FriendName]]
	Description	This conversation processes timeout notifications for persion the user is chatting with.
	Return Messages	
4	Message Pattern	[{user, FriendName}, {user, UserName}, [message,Msg]]
	Description	This conversation processes messages sent from persons the user is chatting with.
	Return Messages	[{user, UserName},{user,FriendName},[message,error]]
Internal Interface		
1	Message Pattern	[addFriend, FriendName]
	Description	This conversation handles communicating with the chat system to add a friend.
	Return	[displayMessage, “Successfully added friend.”]

	Messages	[displayMessage, “Failure: Could not send message to chat server.”] [displayMessage, “Failure: You must be logged on to complete that action.”] [displayMessage, “Unable to add friend.”]
2	Message Pattern	[message, Address, Message]
	Description	This conversation handles sending a message from the user to a person he is chatting with.
	Return Messages	[displayMessage, “Failure: Could not send message to chat server.”] [displayMessage, “Failure: You must be logged on to complete that action.”]
3	Message Pattern	[createUserAccount, UserName, Password]
	Description	This conversation handles creating a user account in the chat system.
	Return Messages	[displayMessage, “User account successfully created.”] [displayMessage, “Failure: Could not create user account .”] [displayMessage, “User account not created: “+Reason] [displayMessage, “User account not created: command timeout out.”]
4	Message Pattern	[deleteUserAccount, UserName, Password]
	Description	This conversation handles asking the chat system to delete a use account.

	Return Messages	[displayMessage, “User account successfully deleted.”] [displayMessage, “Failure: Could not send message to chat server.”] [displayMessage, “User account not deleted: “+Reason] [displayMessage, “User account not deleted: command time out.”]
5	Message Pattern	[login, UserName, Password]
	Description	This conversation handles asking the chat system to log a user in.
	Return Messages	[displayMessage, “Logged In Successfully.”] [displayMessage, “Failure: Could not log in.”] [displayMessage, “Failure: You must be logged off to complete that action.”] [displayMessage, “LogIn failed.”]
6	Message Pattern	[logOut]
	Description	This conversation handles asking the chat system to log the user out.
	Return Messages	[displayMessage, “Logged Out Successfully.”] [displayMessage, “Failure: You must be logged on to complete that action.”] [displayMessage, “Failure: Could not send message to chat server.”] [displayMessage, “LogOut failed.”]
7	Message Pattern	[removeFriend, FriendName]

	Description	This conversation handles communicating with the chat system to remove a user's friend from his friend list.
	Return Messages	[displayMessage, "Successfully removed friend."] [displayMessage, "Failure: You must be logged on to complete that action."] [displayMessage, "Failure: Could not send message to chat server."] [displayMessage, "Unable to remove friend."]
8	Message Pattern	[startChat, FriendName]
	Description	This conversation handles asking the chat server to start a chat with a friend.
	Return Messages	[displayMessage, "Failure: You must be logged on to complete that action."] [displayMessage, "Failure: You could not send that message to the chat server."] [displayMessage, "Failure: Could not start chat."] [displayMessage, "Failure: timeout."]
9	Message Pattern	[viewFriendList]
	Description	This conversation handles asking the chat server to send the user his friend list, including the status of each friend.
	Return Messages	[displayMessage, FriendsList] [displayMessage, "Failure: You must be logged on to complete that action."] [displayMessage, "Failure: Could not send message to

		chat server.”] [displayMessage, “Unable to find friends.”]

A.III Chat Client internal Rich Application Services

Command Window

SDC ID	one	
External Interface		
	Message Pattern	[displayMessage, Msg]
	Description	This conversation forwards the string message *Msg* to the business logic generic server to process for displaying on the gui
	Return Messages	
Internal Interface		
	Message Pattern	[parseInput, Input]
	Description	This conversation takes input from the gui, matches it to a command and forwards the corresponding message. Non matches result in an error message being sent back to the gui.
	Return Messages	{display, “Command Not Recognized.”} {display, “Failed to publish message.”}

Chat Window

SDC ID	one	
External Interface		
1	Message Pattern	[display, Msg]
	Description	This conversation accepts and processes a message

		to display in the GUI.
	Return Messages	
	Message Pattern	[exit, normal]
	Description	This conversation shuts down the Chat Window.
	Return Messages	
Internal Interface		
1	Message Pattern	[message, Address, Message]
	Description	This conversation forwards a string message to the person identified at the address *Address*.
	Return Messages	

Appendix B Chat System Case Study Interaction Diagrams

B.I Chat System Interaction Diagrams

Create Account

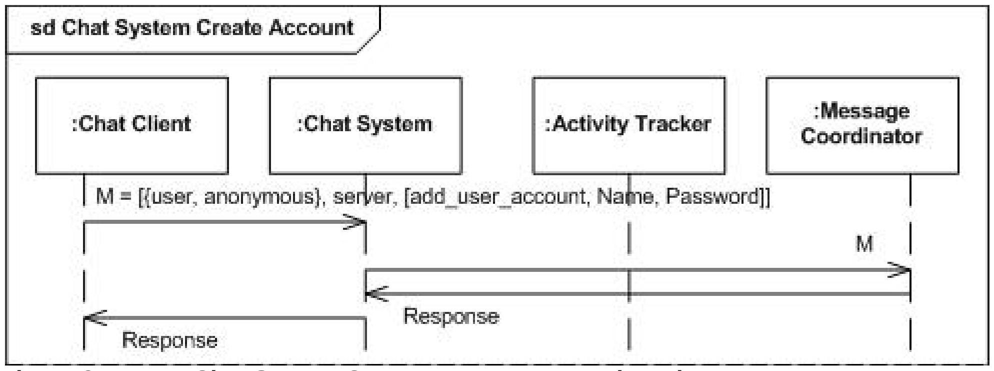

sd Chat System Create Account
:Chat Client
:Chat System
:Activity Tracker
:Message Coordinator
M = [{user, anonymous}, server, [add_user_account, Name, Password]]
M
Response
Response

Figure 35: Chat System Create Account Interaction Diagram

Delete Account

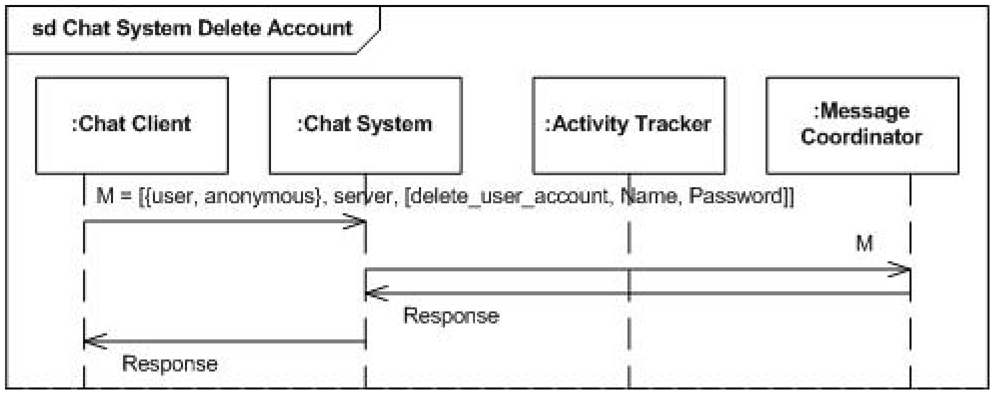

sd Chat System Delete Account
:Chat Client
:Chat System
:Activity Tracker
:Message Coordinator
M = [{user, anonymous}, server, [delete_user_account, Name, Password]]
M
Response
Response

Figure 36: Chat System Delete Account Interaction Diagram

Log In

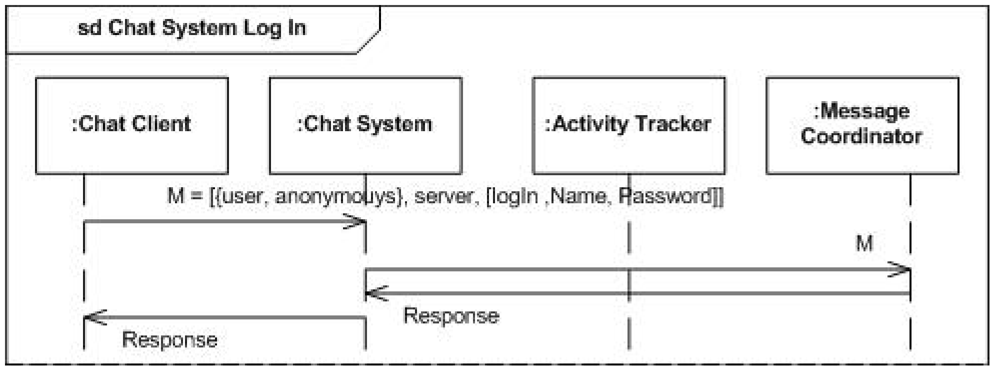


Figure 37: Chat System Log In Interaction Diagram

Create Friendship

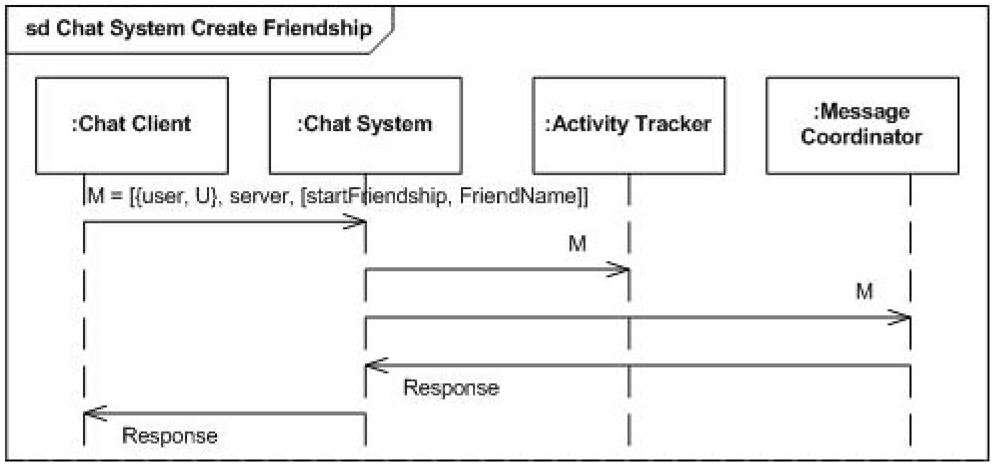

sd Chat System Create Friendship
:Chat Client
:Chat System
:Activity Tracker
:Message Coordinator
M = [{user, U}, server, [startFriendship, FriendName]]
M
M
Response
Response

Figure 38: Chat System Create Friendship Interaction Diagram

Delete Friendship

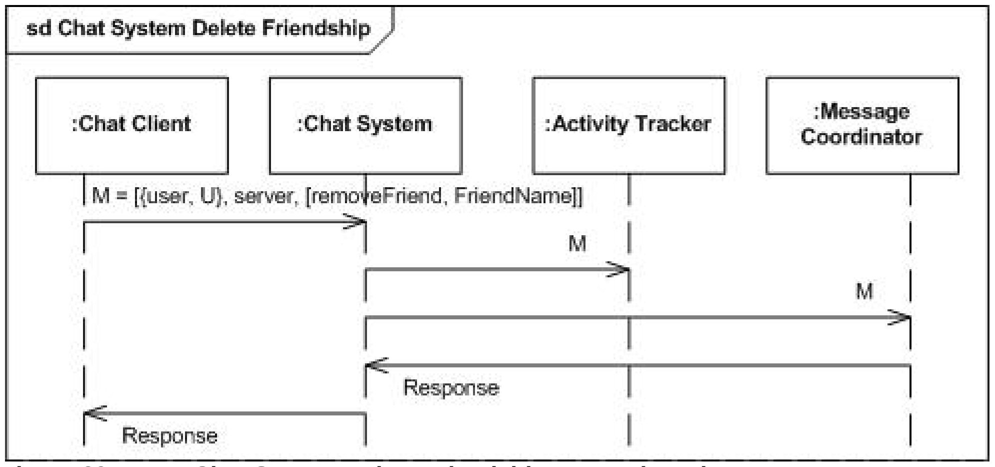

sd Chat System Delete Friendship
:Chat Client
:Chat System
:Activity Tracker
:Message Coordinator
M = [{user, U}, server, [removeFriend, FriendName]]
M
M
Response
Response

Figure 39: Chat System Delete Friendship Interaction Diagram

Start Chat

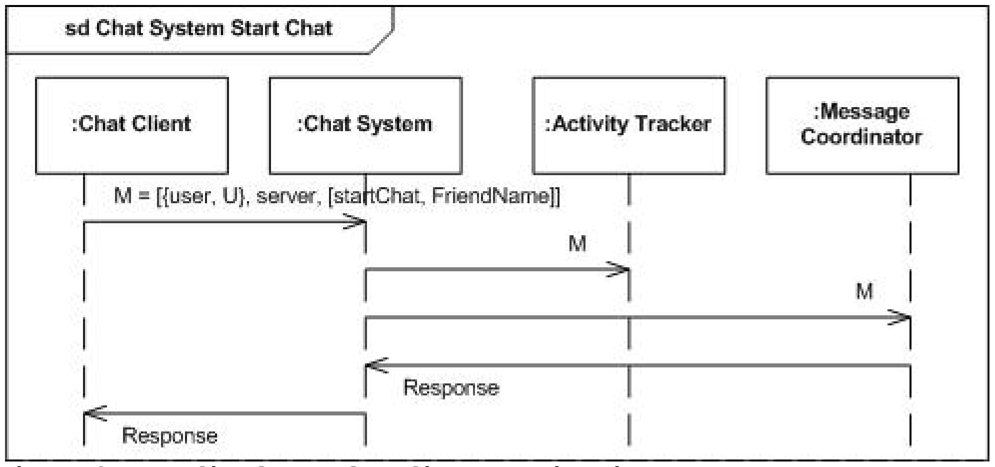


Figure 40: Chat System Start Chat Interaction Diagram

Get Friendship List

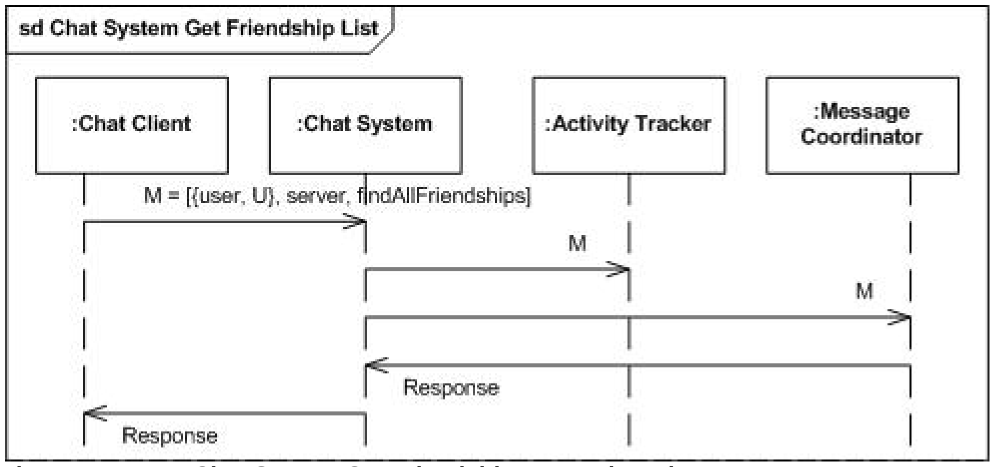


Figure 41: Chat System Get Friendship Interaction Diagram

Log Out

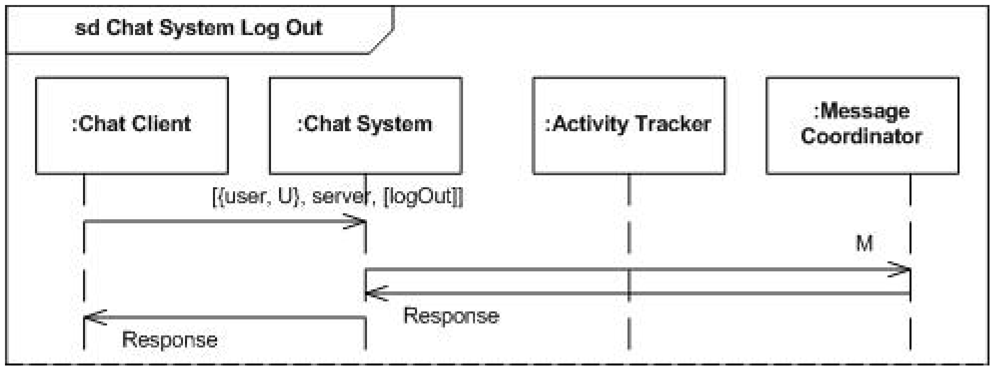


Figure 42: Chat System Log Out Interaction Diagram

B.II Activity Tracker Interaction Diagrams

Log Activity

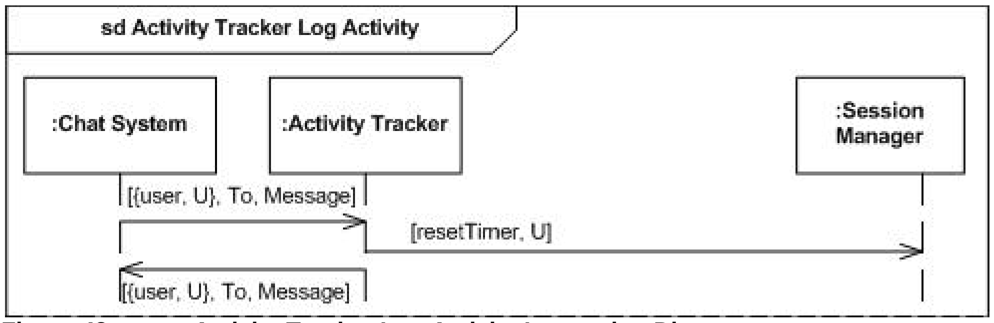


Figure 43: Activity Tracker Log Activity Interaction Diagram

Handle Timeout

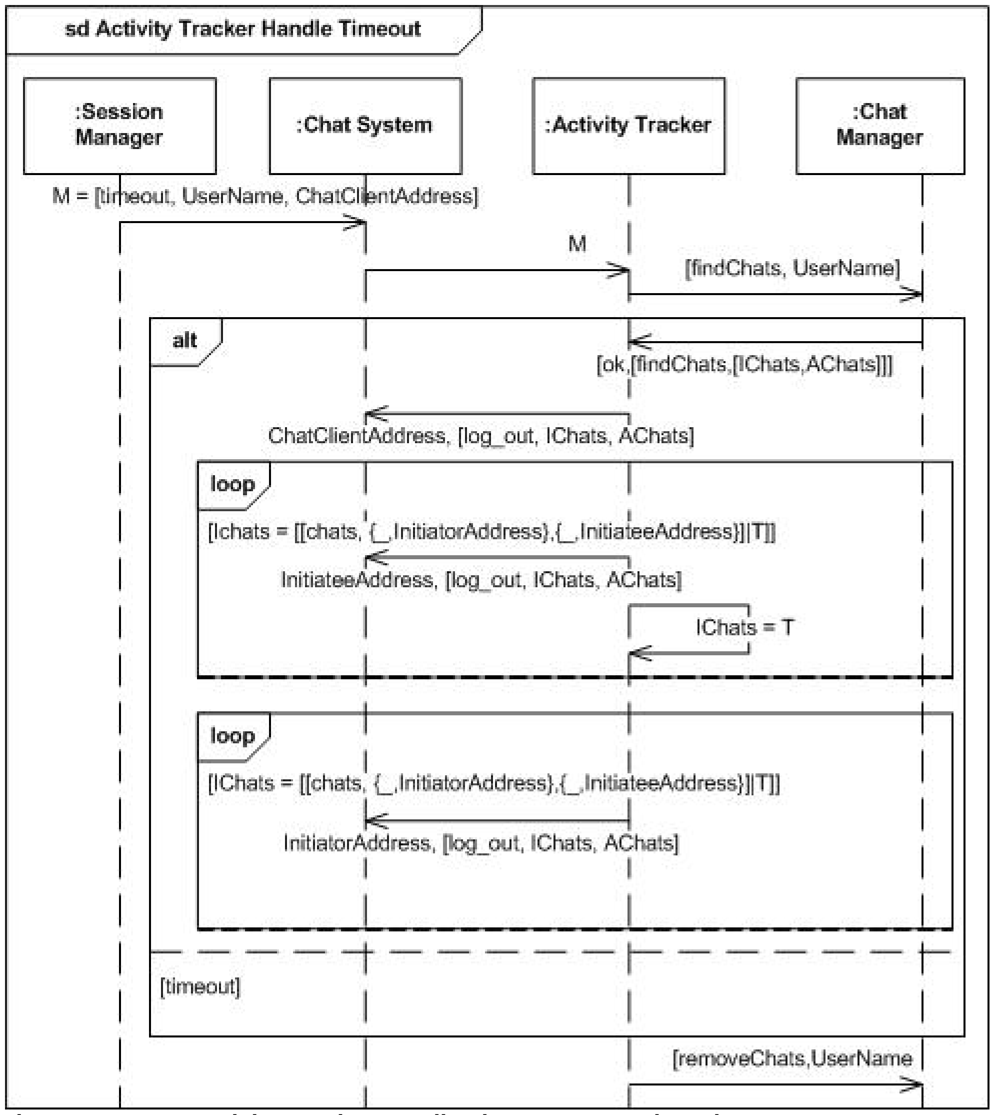

sd Activity Tracker Handle Timeout
:Session Manager
:Chat System
:Activity Tracker
:Chat Manager
M = [timeout, UserName, ChatClientAddress]
M
[findChats, UserName]
alt
[ok,[findChats,[lChats,AChats]]]
ChatClientAddress, [log_out, lChats, AChats]
loop
[lchats = [[chats, {_,InitiatorAddress},{_,InitiateeAddress}]|T]]
InitiateeAddress, [log_out, lChats, AChats]
lChats = T
loop
[lChats = [[chats, {_,InitiatorAddress},{_,InitiateeAddress}]|T]]
InitiatorAddress, [log_out, lChats, AChats]
[timeout]
[removeChats,UserName

Figure 44: **Activity Tracker Handle Timeout Interaction Diagram**

B.III Message Coordinator Interaction Diagrams

Create Account

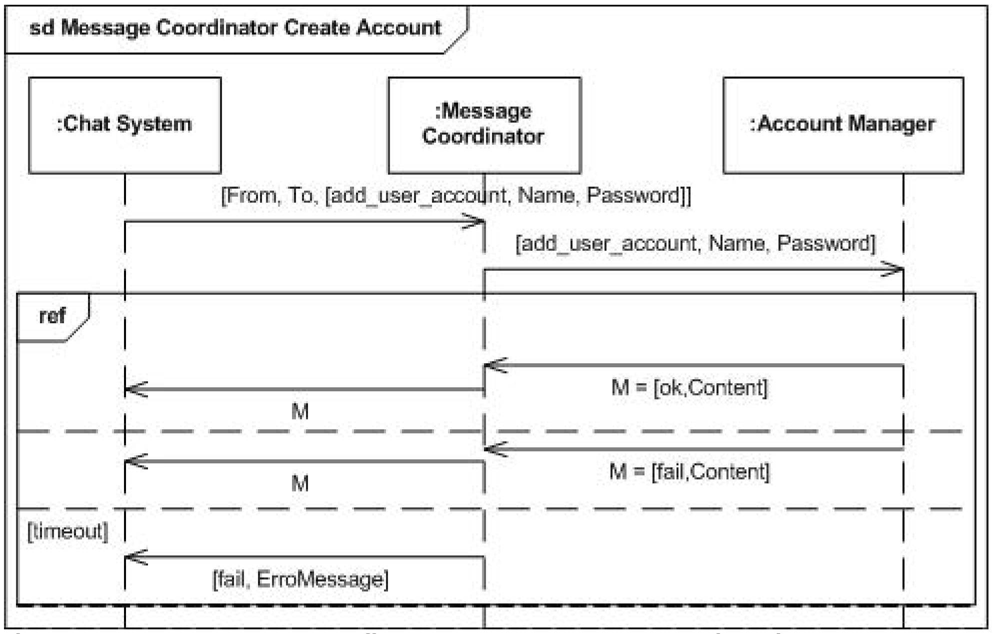

sd Message Coordinator Create Account
:Chat System
:Message Coordinator
:Account Manager
[From, To, [add_user_account, Name, Password]]
[add_user_account, Name, Password]
ref
M = [ok,Content]
M
M = [fail,Content]
M
[timeout]
[fail, ErroMessage]

Figure 45: Message Coordinator Create Account Interaction Diagram

Delete Account

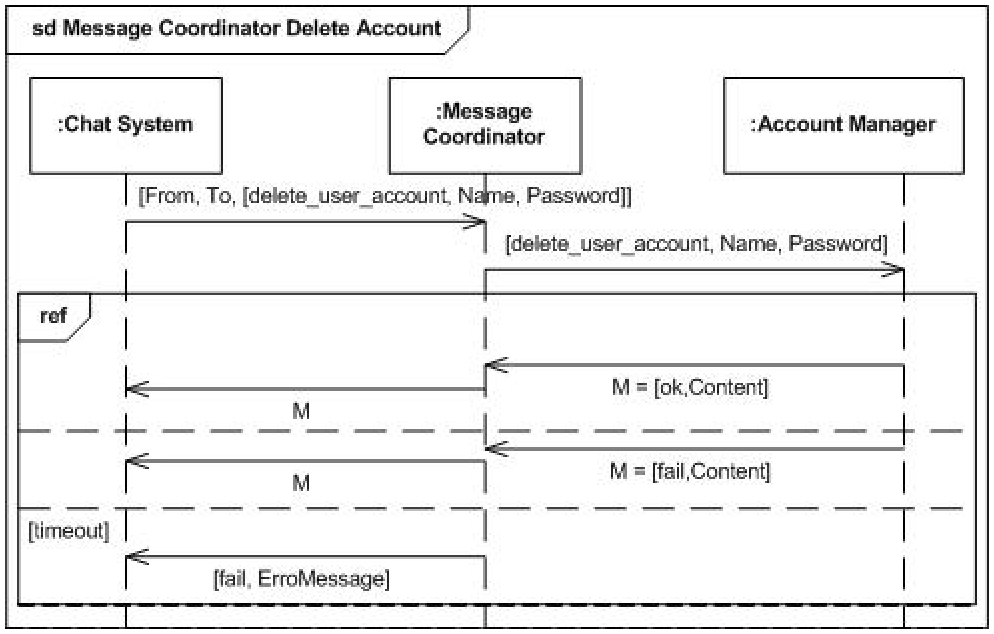

sd Message Coordinator Delete Account
:Chat System
:Message Coordinator
:Account Manager
[From, To, [delete_user_account, Name, Password]]
[delete_user_account, Name, Password]
ref
M = [ok,Content]
M
M = [fail,Content]
M
[timeout]
[fail, ErroMessage]

Figure 46: **Message Coordinator Delete Account Interaction Diagram**

Log In

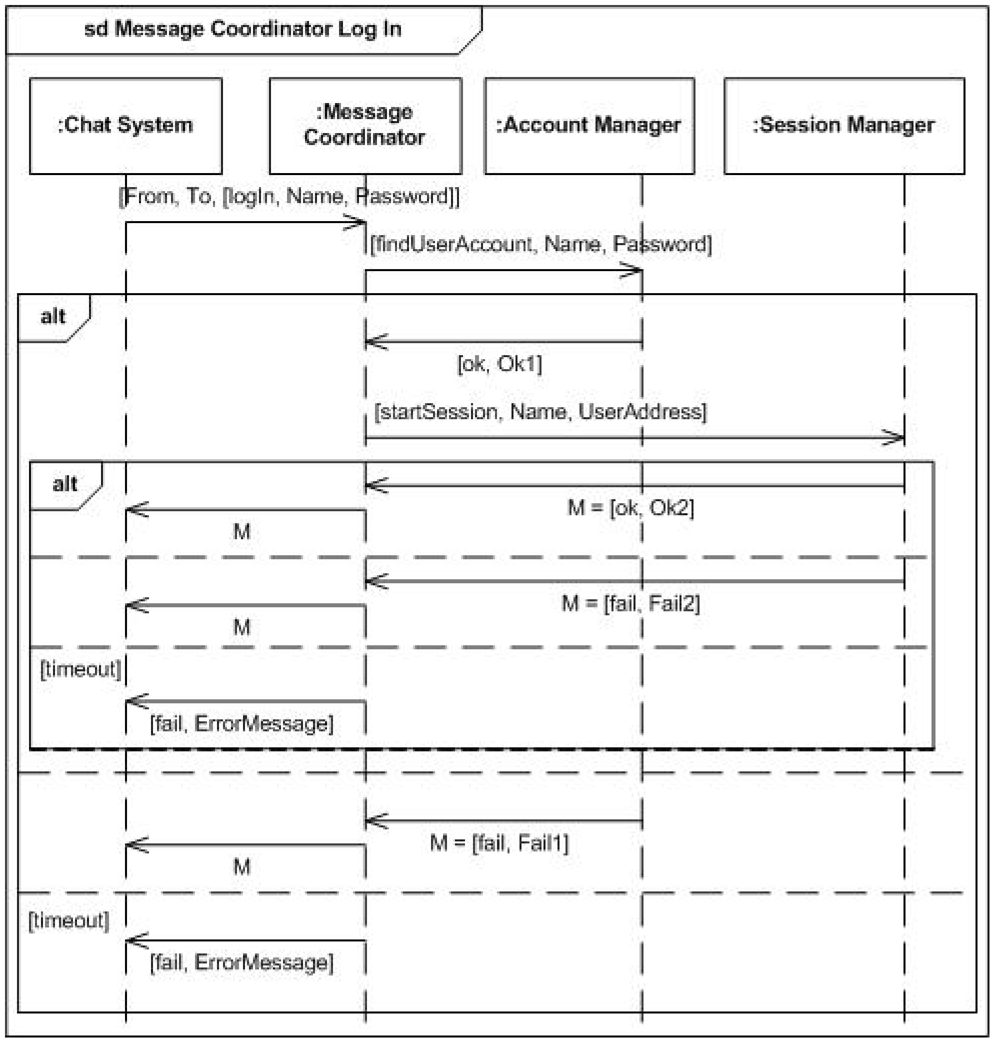


Figure 47: **Message Coordinator Log In Interaction Diagram**

Create Friendship

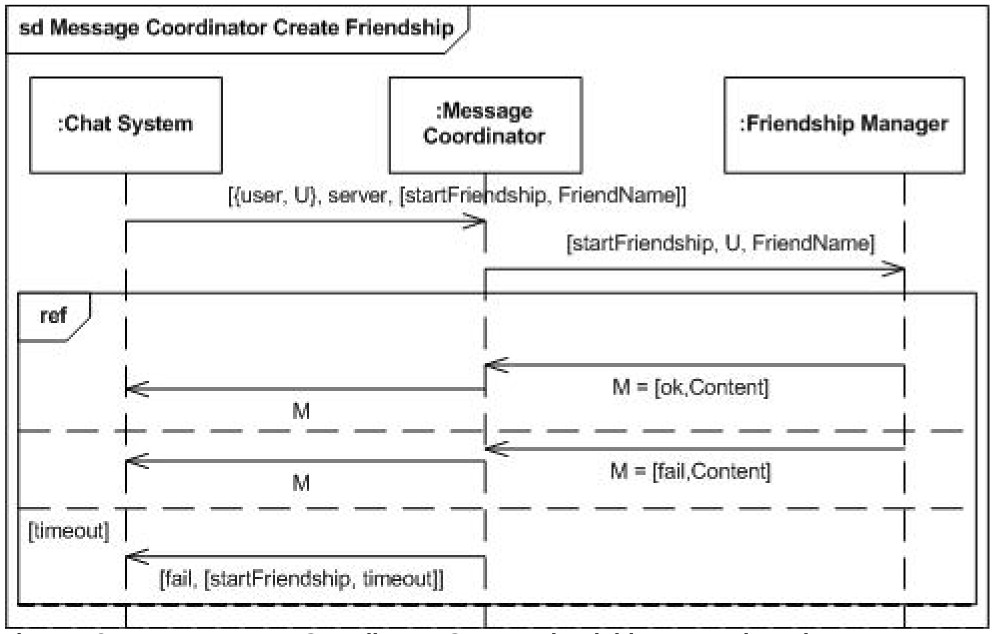

sd Message Coordinator Create Friendship
:Chat System
:Message Coordinator
:Friendship Manager
[{user, U}, server, [startFriendship, FriendName]]
[startFriendship, U, FriendName]
ref
M = [ok,Content]
M
M = [fail,Content]
M
[timeout]
[fail, [startFriendship, timeout]]

Figure 48: Message Coordinator Create Friendship Interaction Diagram

Delete Friendship

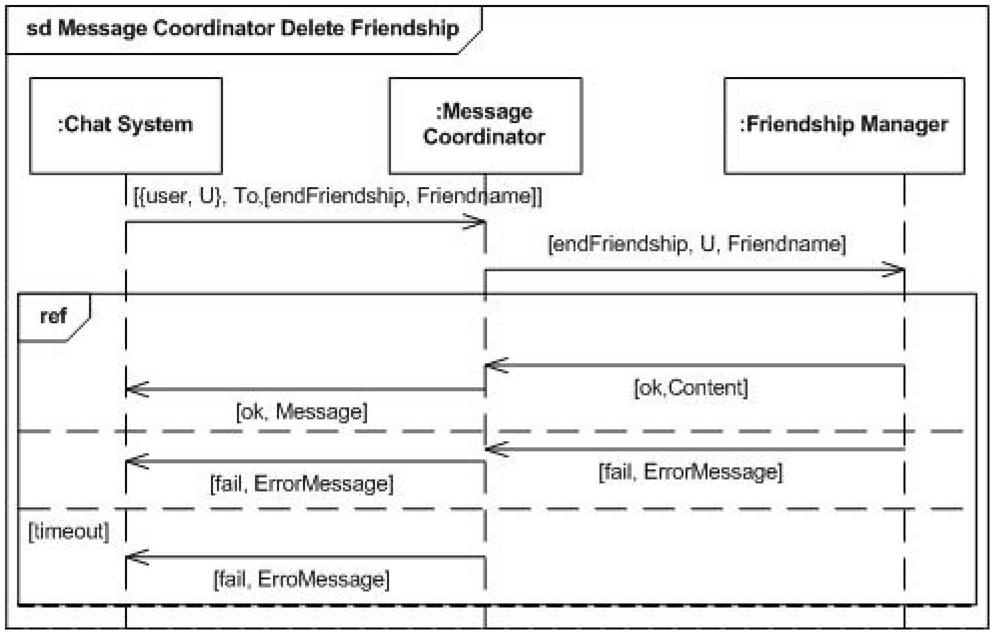


Figure 49: Message Coordinator Delete Friendship Interaction Diagram

Get Friendship List

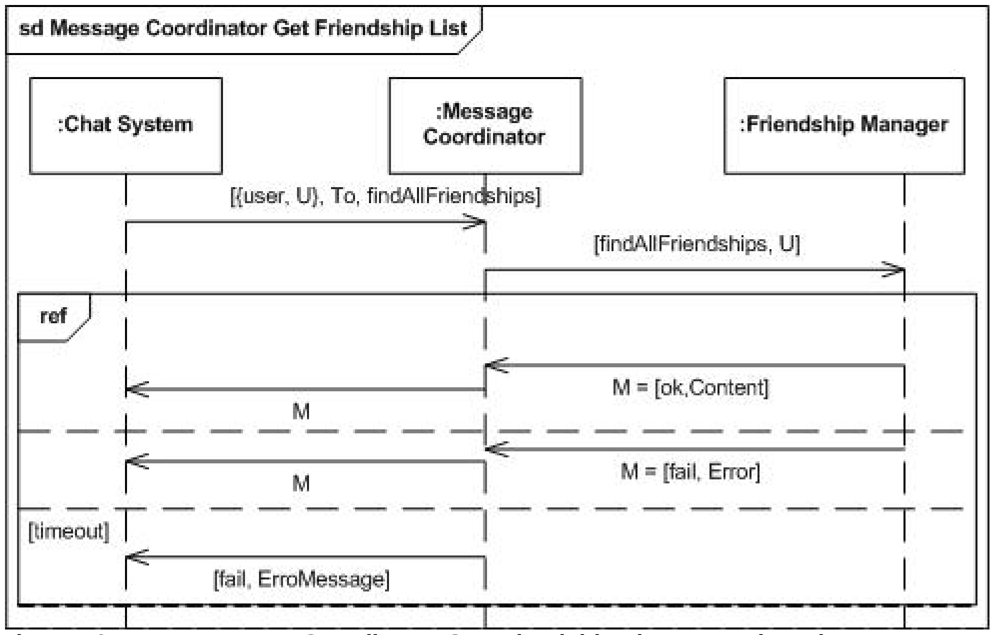


Figure 50: Message Coordinator Get Friendship List Interaction Diagram

Log Out

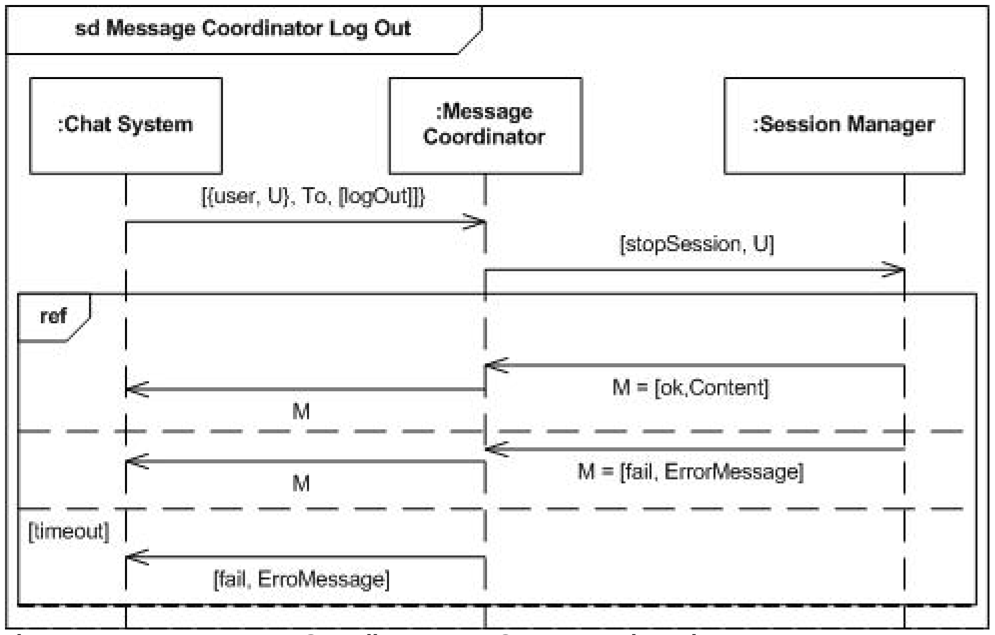


Figure 51: Message Coordinator Log Out Interaction Diagram

Start Chat

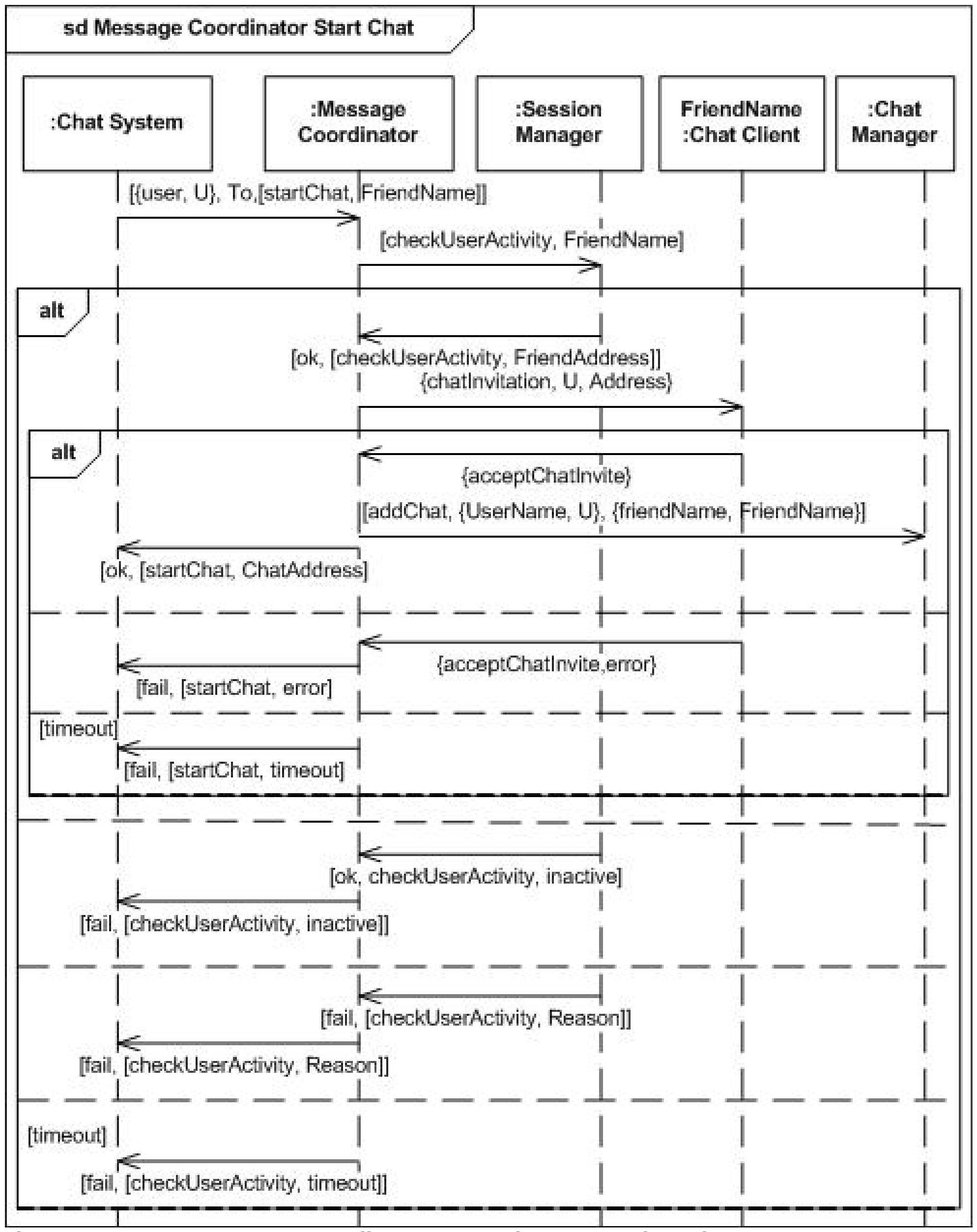

sd Message Coordinator Start Chat
:Chat System
:Message Coordinator
:Session Manager
FriendName :Chat Client
:Chat Manager
[{user, U}, To,[startChat, FriendName]]
[checkUserActivity, FriendName]
alt
[ok, [checkUserActivity, FriendAddress]]
{chatInvitation, U, Address}
alt
{acceptChatInvite}
[addChat, {UserName, U}, {friendName, FriendName}]
[ok, [startChat, ChatAddress]
{acceptChatInvite,error}
[fail, [startChat, error]
[timeout]
[fail, [startChat, timeout]
[ok, checkUserActivity, inactive]
[fail, [checkUserActivity, inactive]]
[fail, [checkUserActivity, Reason]]
[fail, [checkUserActivity, Reason]]
[timeout]
[fail, [checkUserActivity, timeout]]

Figure 52: Message Coordinator Start Chat Interaction Diagram

B.IV Chat Client Interaction Diagrams

Add Friend

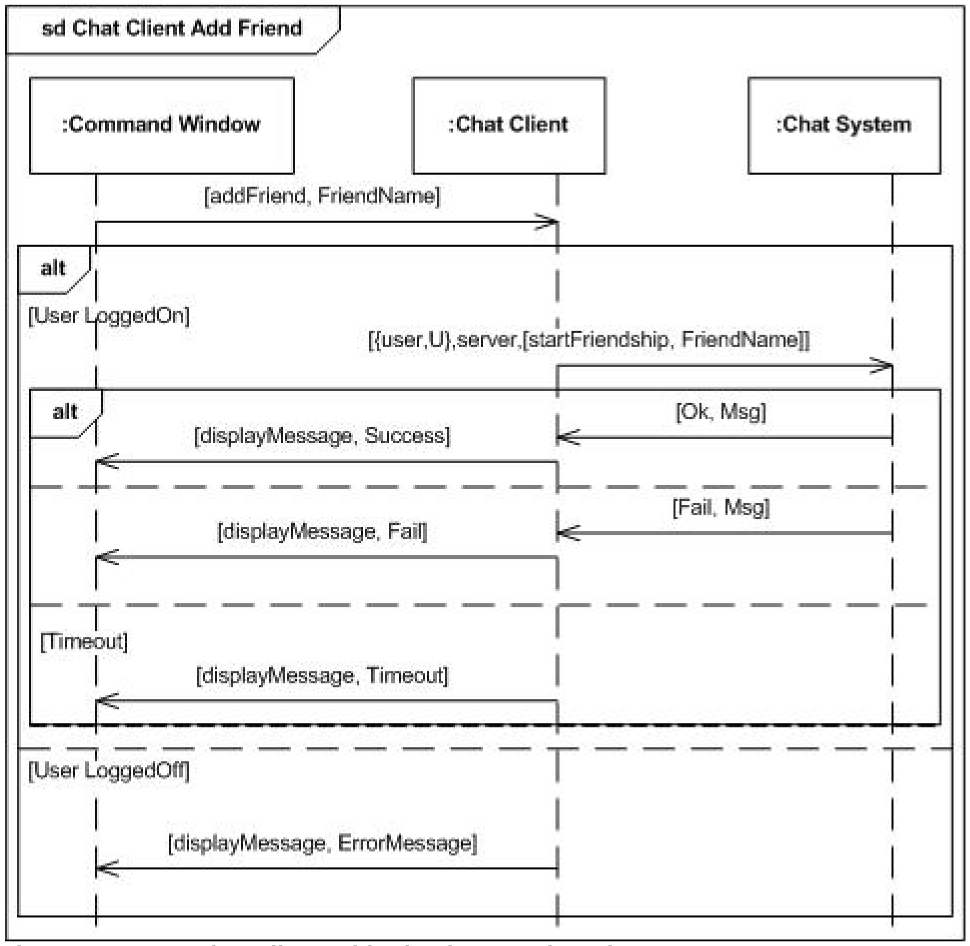

sd Chat Client Add Friend
:Command Window
:Chat Client
:Chat System
[addFriend, FriendName]
alt
[User LoggedOn]
[{user,U},server,[startFriendship, FriendName]]
alt
[Ok, Msg]
[displayMessage, Success]
[Fail, Msg]
[displayMessage, Fail]
[Timeout]
[displayMessage, Timeout]
[User LoggedOff]
[displayMessage, ErrorMessage]

Figure 53: Chat Client Add Friend Interaction Diagram

Broadcast Message

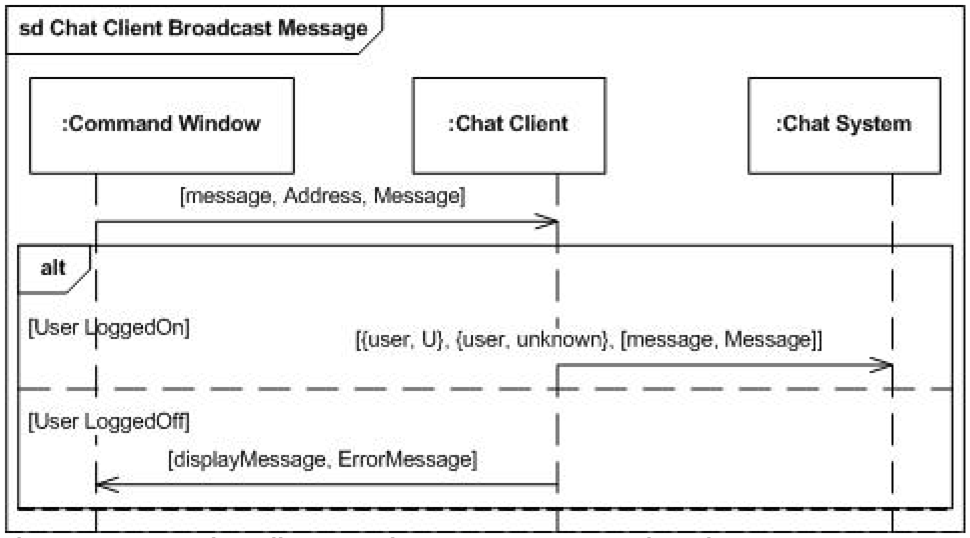

sd Chat Client Broadcast Message
:Command Window
:Chat Client
:Chat System
[message, Address, Message]
alt
[User LoggedOn]
[{user, U}, {user, unknown}, [message, Message]]
[User LoggedOff]
[displayMessage, ErrorMessage]

Figure 54: Chat Client Broadcast Message Interaction Diagram

Create Account

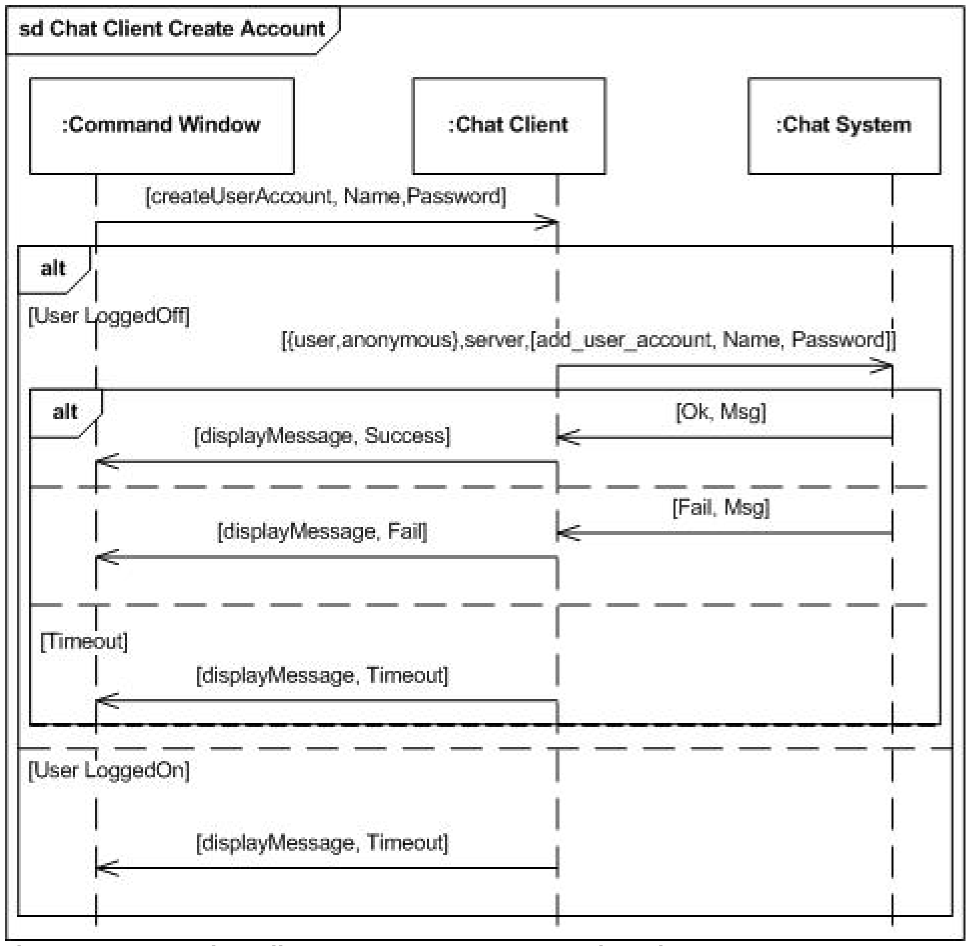

sd Chat Client Create Account
:Command Window
:Chat Client
:Chat System
[createUserAccount, Name,Password]
alt
[User LoggedOff]
[{user,anonymous},server,[add_user_account, Name, Password]]
alt
[Ok, Msg]
[displayMessage, Success]
[Fail, Msg]
[displayMessage, Fail]
[Timeout]
[displayMessage, Timeout]
[User LoggedOn]
[displayMessage, Timeout]

Figure 55: Chat Client Create Account Interaction Diagram

Delete Account

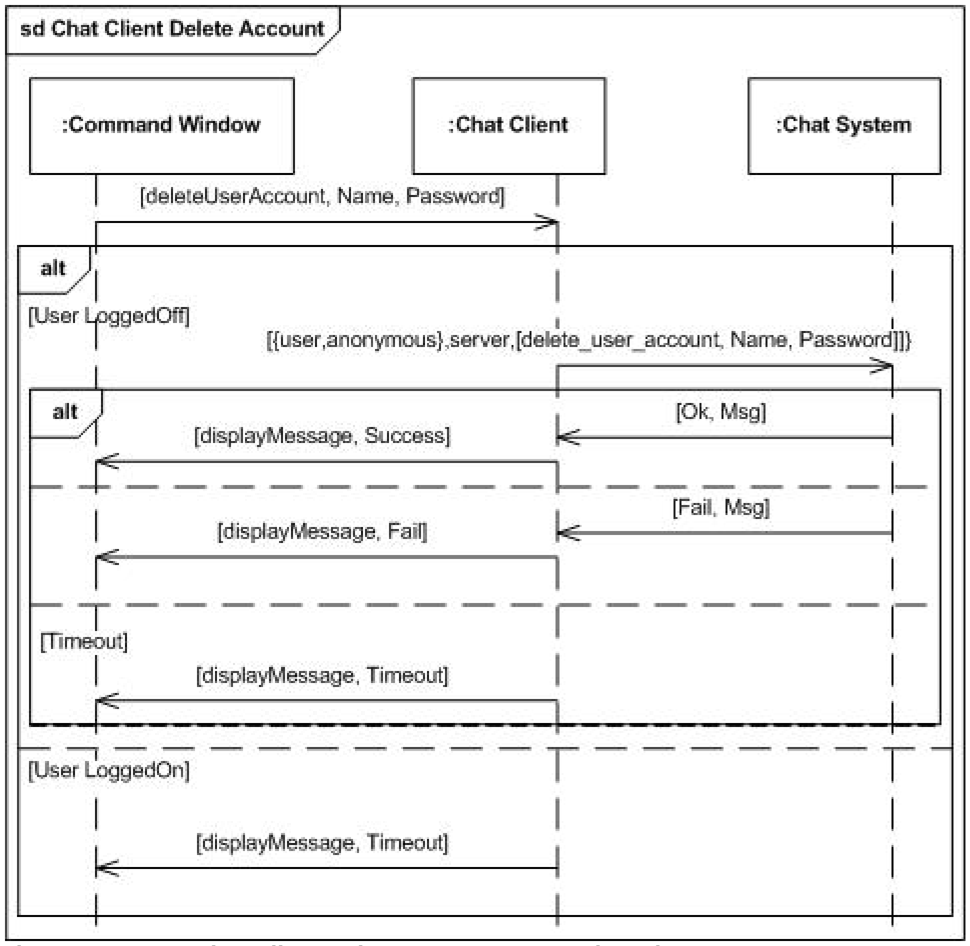

sd Chat Client Delete Account
:Command Window
:Chat Client
:Chat System
[deleteUserAccount, Name, Password]
alt
[User LoggedOff]
[{user,anonymous},server,[delete_user_account, Name, Password]]}
alt
[Ok, Msg]
[displayMessage, Success]
[Fail, Msg]
[displayMessage, Fail]
[Timeout]
[displayMessage, Timeout]
[User LoggedOn]
[displayMessage, Timeout]

Figure 56: Chat Client Delete Account Interaction Diagram

Delete Friendship

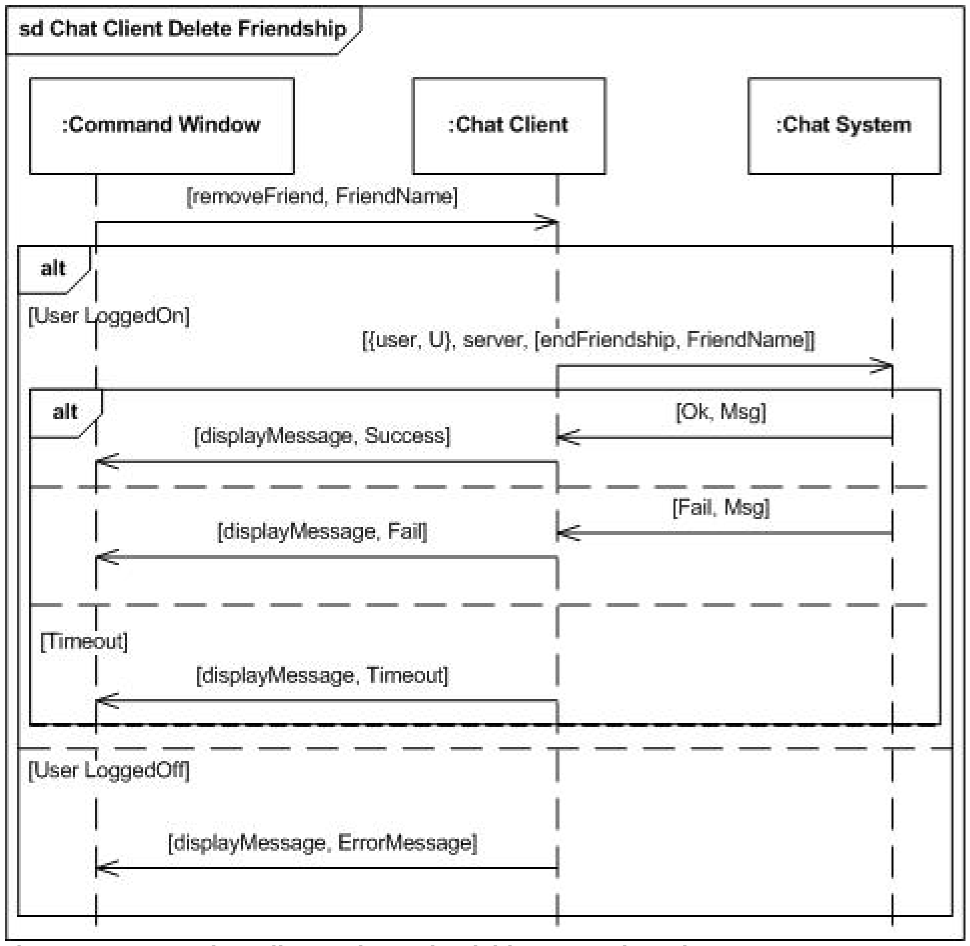

sd Chat Client Delete Friendship
:Command Window
:Chat Client
:Chat System
[removeFriend, FriendName]
alt
[User LoggedOn]
[{user, U}, server, [endFriendship, FriendName]]
alt
[Ok, Msg]
[displayMessage, Success]
[Fail, Msg]
[displayMessage, Fail]
[Timeout]
[displayMessage, Timeout]
[User LoggedOff]
[displayMessage, ErrorMessage]

Figure 57: **Chat Client Delete Friendship Interaction Diagram**

Get Friendship List

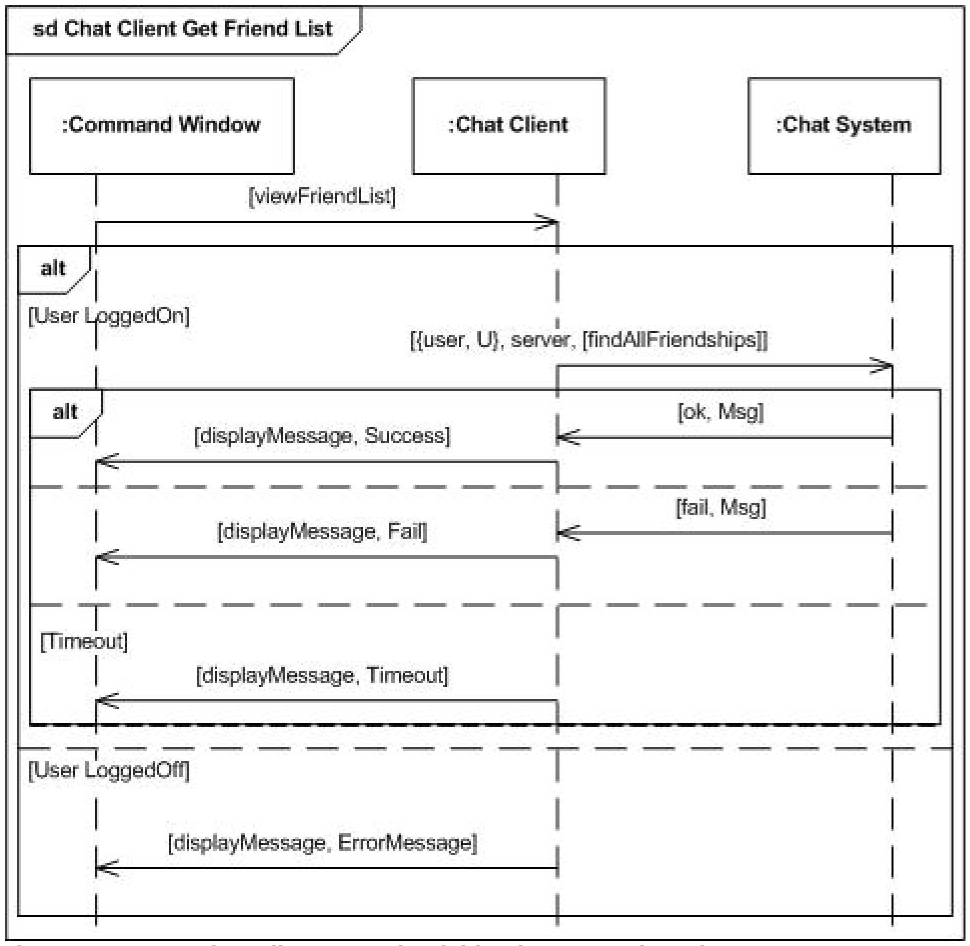

sd Chat Client Get Friend List
:Command Window
:Chat Client
:Chat System
[viewFriendList]
alt
[User LoggedOn]
[{user, U}, server, [findAllFriendships]]
alt
[ok, Msg]
[displayMessage, Success]
[fail, Msg]
[displayMessage, Fail]
[Timeout]
[displayMessage, Timeout]
[User LoggedOff]
[displayMessage, ErrorMessage]

Figure 58: **Chat Client Get Friendship List Interaction Diagram**

Log In

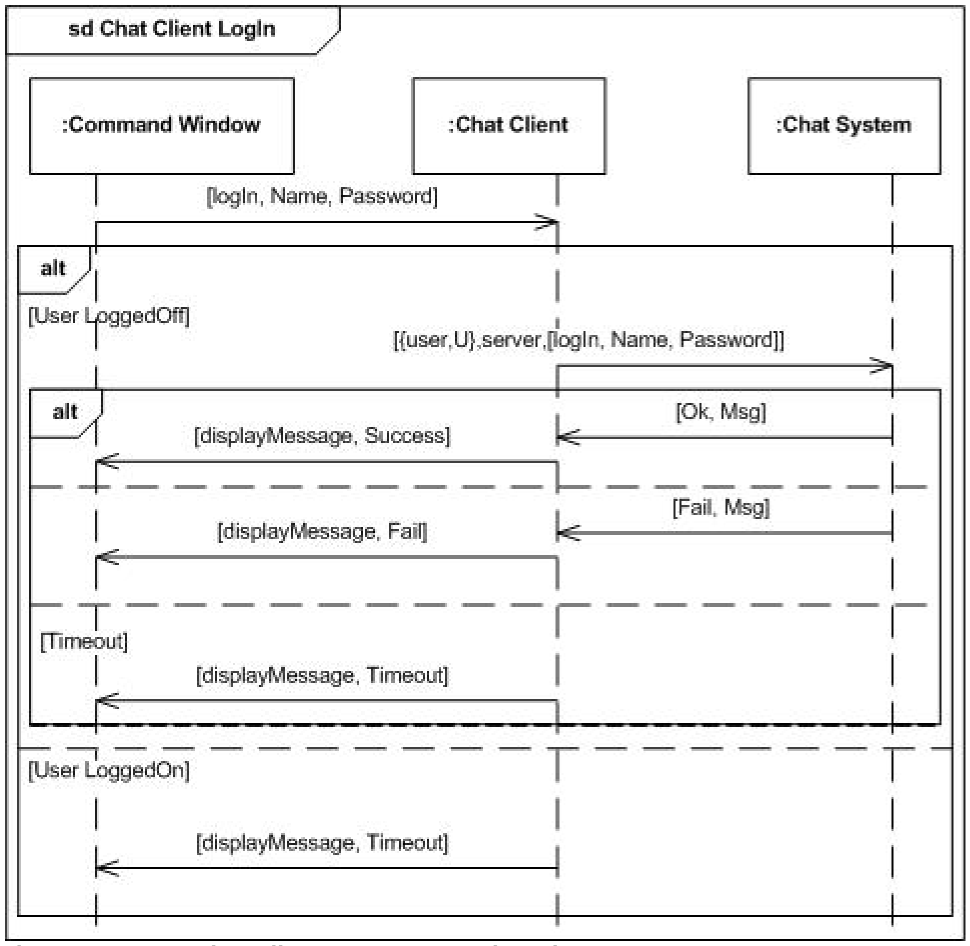


Figure 59: **Chat Client Log In Interaction Diagram**

Log Out

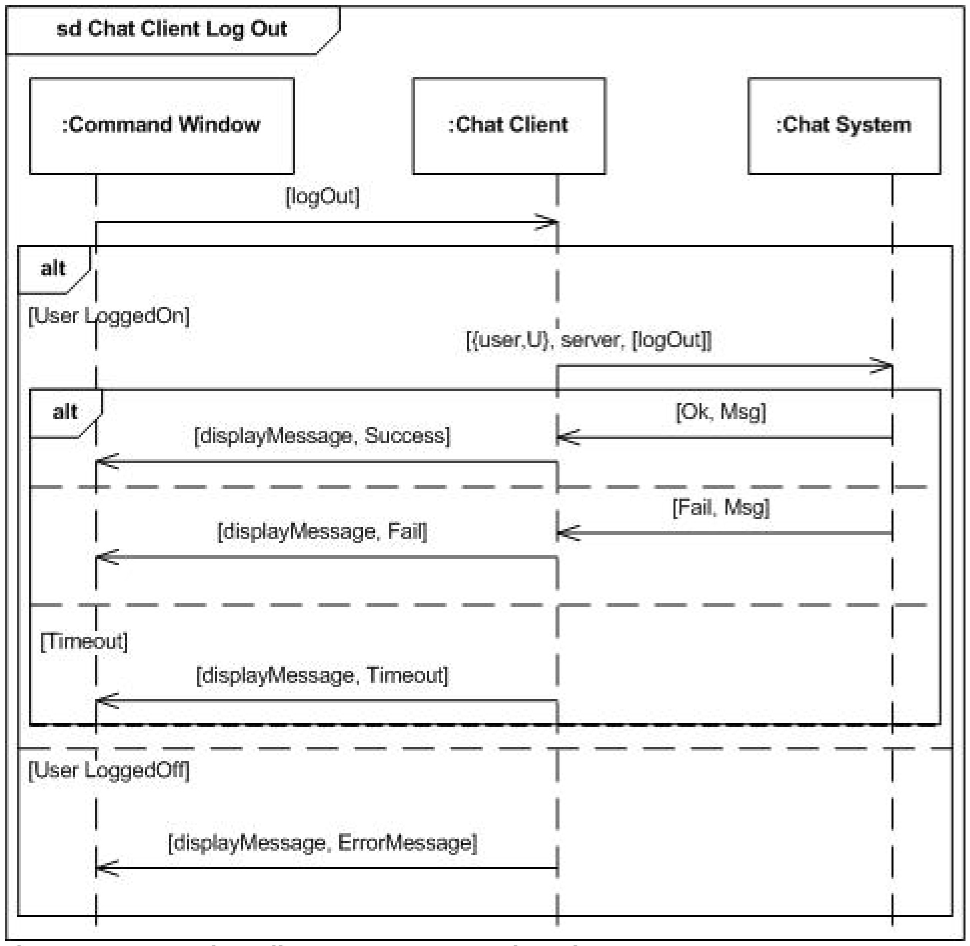


Figure 60: Chat Client Log Out Interaction Diagram

Process Chat Invitation

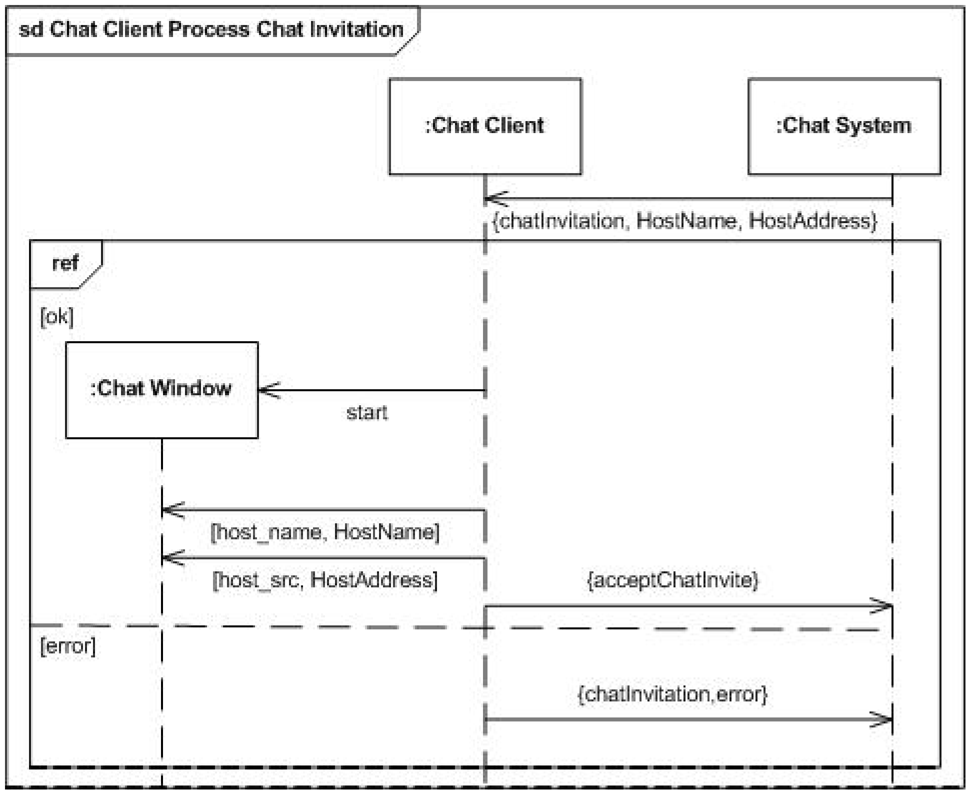


Figure 61: Chat Client Process Chat Invite Interaction Diagram

Session Timeout

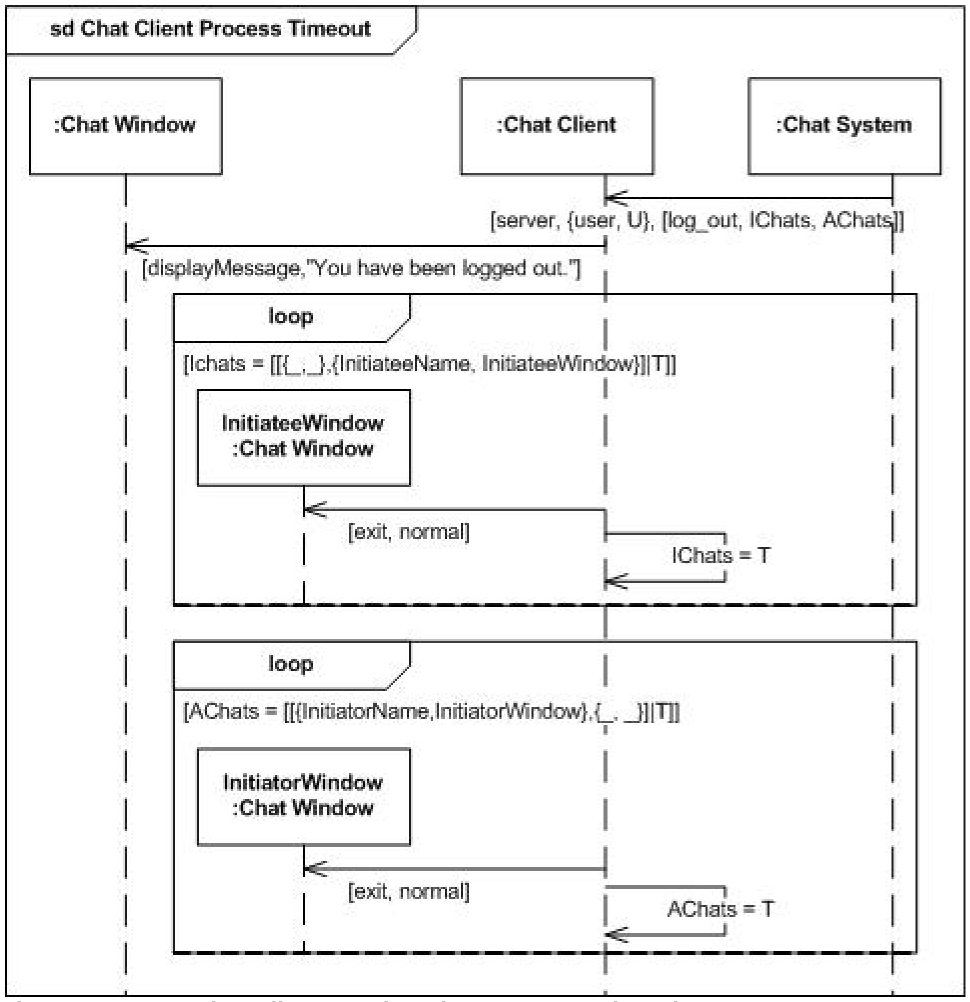

sd Chat Client Process Timeout
:Chat Window
:Chat Client
:Chat System
[server, {user, U}, [log_out, IChats, AChats]]
[displayMessage,"You have been logged out."]
loop
[Ichats = [[{_,_},{InitiateeName, InitiateeWindow}]|T]]
InitiateeWindow :Chat Window
[exit, normal]
IChats = T
loop
[AChats = [[{InitiatorName,InitiatorWindow},{_, _}]|T]]
InitiatorWindow :Chat Window
[exit, normal]
AChats = T

Figure 62: Chat Client Session Timeout Interaction Diagram

Process User Timeout

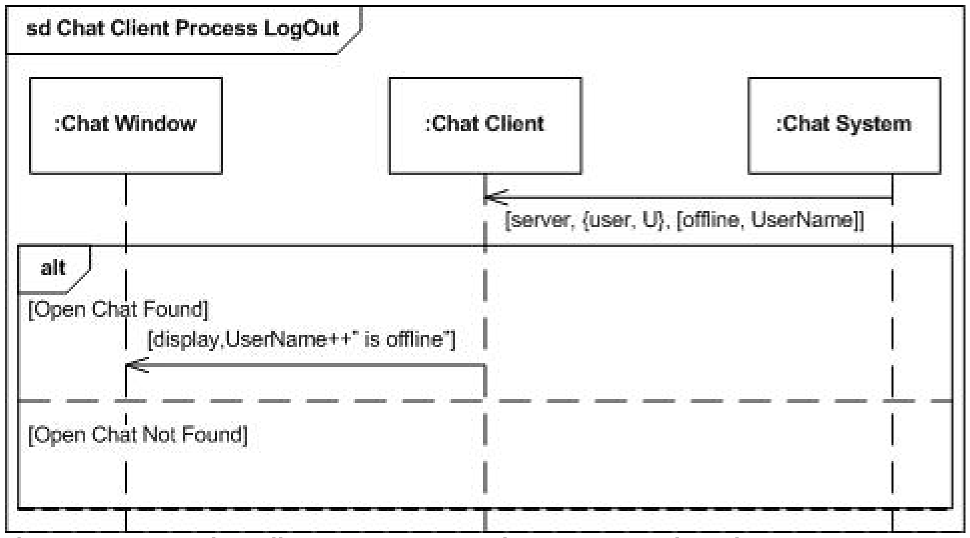

sd Chat Client Process LogOut
:Chat Window
:Chat Client
:Chat System
[server, {user, U}, [offline, UserName]]
alt
[Open Chat Found]
[display,UserName++" is offline"]
[Open Chat Not Found]

Figure 63: Chat Client Process User Timeout Interaction Diagram

Receive Message

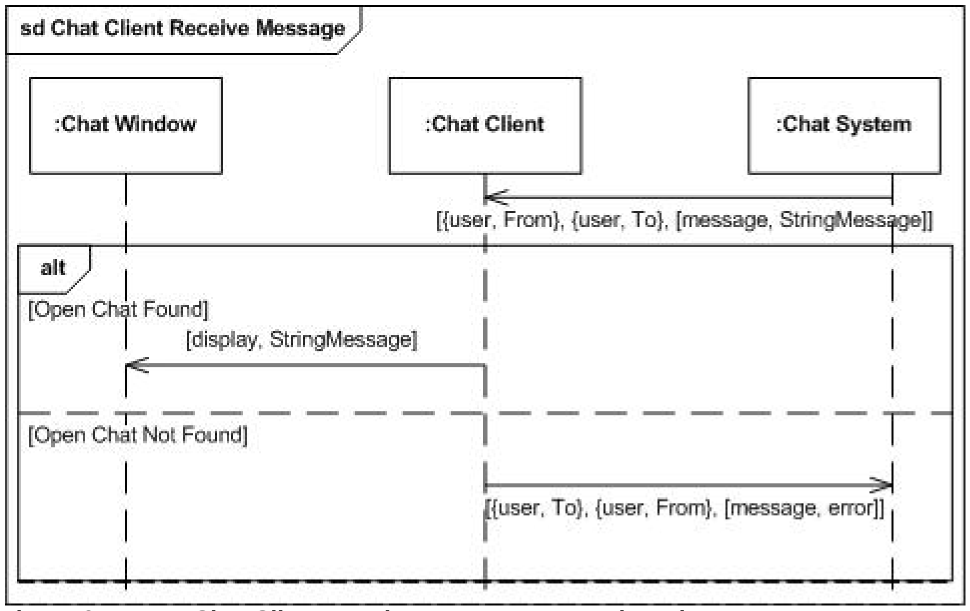

sd Chat Client Receive Message
:Chat Window
:Chat Client
:Chat System
[{user, From}, {user, To}, [message, StringMessage]]
alt
[Open Chat Found]
[display, StringMessage]
[Open Chat Not Found]
[{user, To}, {user, From}, [message, error]]

Figure 64: Chat Client Receive Message Interaction Diagram

Start Chat

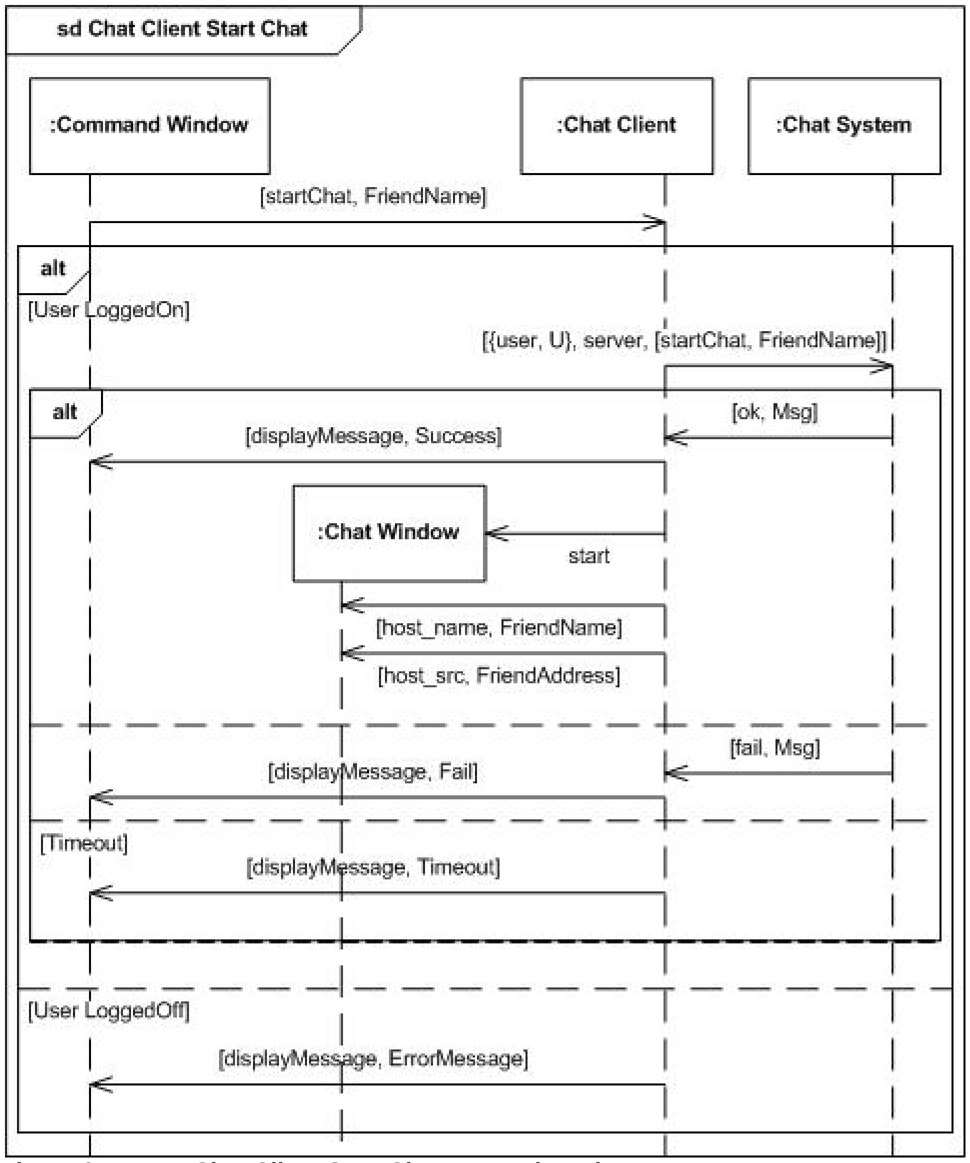


Figure 65: Chat Client Start Chat Interaction Diagram

Bibliography

[1] J. Armstrong. Programming Erlang. The Pragmatic Bookshelf, 2007.

[2] M. Arrott, B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, “Rich Services: The Integration Piece of the SOA Puzzle,” in *Proceedings of the IEEE International Conference on Web Services (ICWS),* Salt Lake City, Utah, USA. IEEE, Jul. 2007, pp. 176-183.

[3] K. Brown, C D’Cruz, M. Fowler, et. al. Enterprise Integration Patterns. Pearson Education Inc, 2004.

[4] B. Demchak, V. Ermagan, E. Farcas, T.-J. Huang, I. Krüger, and M. Menarini, “A Rich Services Approach to CoCoME,” *The Common Component Modeling Example: Comparing Software Component Models,* A. Rausch, R. Reussner, R. Mirandola, and F. Plasil (Eds.), Lecture Notes in Computer Science, no. 5153, ch. 5, pp. 85-115, Berlin/Heidelberg: Springer-Verlag, Aug. 2008.

[5] B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, “Rich Services: Addressing Challenges of Ultra-Large-Scale Software-Intensive Systems,” in *Proceedings of the ICSE 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008),* Leipzig, Germany. New York, NY, USA: ACM Press, May 2008, pp. 29-32.

[6] B. Demchak, C. Farcas, E. Farcas, and I. H. Krüger, “The Treasure Map for Rich Services,” in *Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI),* Las Vegas, USA. IEEE, Aug. 2007, pp. 400-405.

[7] Comprehensive Erlang Archive Network. http://cean.process-one.net/packages/index.yaws?action=category&name=Erlang/OTP

[8] Erlang Online Publication. http://www.erlang.se/publications/Ulf_Wiger.ppt

[9] Erlang Reference Manual. http://erlang.org/doc/reference_manual/expressions.html#6

[10] Erlang Reference Manual. http://erlang.org/doc/reference_manual/data_types.html#2.1

[11] Erlang Reference Manual. http://erlang.org/doc/reference_manual/data_types.html#2.8

[12] Erlang Reference Manual. http://erlang.org/doc/reference_manual/data_types.html#2.9

[13] Erlang Reference Manual. http://erlang.org/doc/reference_manual/data_types.html#2.10

[14] Erlang Reference Manual. http://erlang.org/doc/reference_manual/data_types.html#2.11

[15] Erlang Reference Manual. http://erlang.org/doc/reference_manual/distributed.html#11.2

[16] Erlang Reference Manual. http://erlang.org/doc/reference_manual/expressions.html#funs

[17] Erlang Reference Manual. http://erlang.org/doc/reference_manual/expressions.html#6.4

[18] Erlang Reference Manual. http://erlang.org/doc/reference_manual/modules.htm#4

[19] Erlang Reference Manual. http://erlang.org/doc/design_principles/gen_server_concepts.html#2

[20] Erlang Reference Manual. http://erlang.org/doc/reference_manual/ports.html#13

[21] Erlang Reference Manual. http://erlang.org/doc/design_principles/fsm.html#3

[22] Erlang Reference Manual. http://erlang.org/doc/design_principles/applications.html#7

[23] Performance Measurements of Threads in Java and Processes in Erlang. http://www.sics.se/~joe/ericsson/du98024.html

[24] RabbitMQ FAQ. http://www.rabbitmq.com/faq.html#performance-latency