SEMINAR REPORT

ON

ERLANG – CONCURRENT PROGRAMMING LANGUAGE

Submitted By
SREEJESH V. K.
M.Tech CIS
2010-2012 Batch
ABSTRACT

Erlang is a concurrent programming language designed for programming fault-tolerant distributed systems at Ericsson and has been, since 2000, freely available subject to an open-source license. More recently, we’ve seen renewed interest in Erlang, as the Erlang way of programming maps naturally to multi-core computers. In it the notion of a process is fundamental, with processes created and managed by the Erlang runtime system, not by the underlying operating system. The individual processes, which are programmed in a simple dynamically typed functional programming language, do not share memory and exchange data through message passing, simplifying the programming of multi-core computers. Its native support for distribution and fault recovery makes it a very powerful language, since the paradigm can scale from the multiple cores of a CPU to a large network-centric system.

Keywords: Concurrent programming, fault-tolerant, reliable, distributed systems, multi-core
## Contents

1. Introduction 4
2. Erlang History 4
3. Programming in Erlang 4
   - 3.1. Erlang View of the World 4
   - 3.2. Erlang Programs 5
   - 3.3. A Simple Sequential Program 5
   - 3.4. The Concurrency Primitives 6
   - 3.5. A Concurrent Program 7
4. Detecting and Handling errors 8
   - 4.1. Erlang View of Errors 8
   - 4.2. Detecting Errors 8
5. Building Reliable Systems 9
6. Dynamic Code Upgrade 10
7. Open Telecoms Platform 11
8. Distributed Erlang 12
9. Erlang and Multicore 13
10. Erlang Distribution 13
11. Existing Applications 14
12. Facebook Chat 15
13. Conclusion and Future 16
14. References 17
1. Introduction

Erlang is used for programming fault-tolerant, distributed, soft real-time, non-stop applications. What differentiates it from most other languages is that it is concurrent programming language; concurrency belongs to the language, not to the operating system. Its programs are collections of parallel processes cooperating to solve a particular problem that can be created quickly and have only limited memory overhead. All Erlang processes are isolated from one another and in principle are “thread safe.” Erlang has no mutexes, and processes cannot share memory. The sequential Erlang subset that executes within an individual process is a dynamically typed functional programming language with immutable state. It also includes sophisticated error handling, code-replacement mechanisms, and a large set of libraries.

Erlang is well-placed for programming multicore CPUs. Now a day, networked applications are extremely common and multicore computers are everywhere. As the number of cores increases so does the need for isolation throughout the system. Small isolated computations are easily allocated to a pool of cores. When Erlang applications are deployed on multicore computers, the individual Erlang processes are spread over the cores, and programmers do not have to worry about the details. The isolated processes share no data, and polymorphic messages can be sent between processes.

2. Erlang History

Erlang started life in the telecommunications industry at Ericsson in 1985-86, where their engineers sought to create a language for building telecommunications switches. They required their language to help them build highly concurrent, fault-tolerant, highly available, and distributed services that supported live upgrades and ran with virtually zero downtime. The initial version was developed by Joe Armstrong, Mike Williams and Robert Virding in 1986 at the Ericsson Computer Science Laboratory. The design of Erlang was heavily influenced by ideas from the logic and functional programming communities. It was a strange mixture, with declarative features (inherited from Prolog), multi-tasking and concurrency (inherited from EriPascal and Ada) and an original combination of error handling mechanisms. In 1998, after years of development and honing to meet these stringent requirements, Erlang became open source software.

3. Programming in Erlang

3.1. Erlang View of the World

The Erlang view of the world is that everything is a process that lacks shared memory and influences one another only by exchanging asynchronous messages. Each process has a mailbox to which messages can be sent. Messages are retrieved from the mailbox with a receive statement or pattern-matching construction that removes messages matching a particular pattern in the mailbox. Pattern matching can also be used to selectively remove messages from the mailbox.

Erlang is a functional language that wholly embraces the “shared nothing” concept – for example, unlike in Java or C++, its “variables” can’t be changed once they are bound to a
value. In Erlang, the = operator supports pattern matching rather than traditional assignment. When an unbound variable appears on the left of the = operator, Erlang sets its value to match what’s on the right, thus making the assignment succeed. On the other hand, if a bound variable appears on the left of the = operator, the statement succeeds only if the right-hand side has the same value.

Because Erlang variables are immutable, they don’t need concurrency protection. In addition, single-assignment variables simplify program debugging: if a variable is found to have an incorrect value, there’s only one place you need to look in your code to find out why. Avoiding shared variables allows for higher degrees of program parallelization, assuming threads aren’t too heavy-weight.

Erlang is similar to other functional languages in that it encourages list processing and tail recursion. Once a tail call is made there is no going back; tail calls do not create additional stack frames, since there is nowhere to return to, and a new stack frame is not required. Having made a tail call, all local variables in the current context can be garbage-collected, allowing tail-recursive loops to run indefinitely without consuming stack space.

3.2. Erlang Programs

Erlang was first implemented in Prolog in 1986, and thus many of the syntactic conventions used in Erlang come from Prolog. Erlang’s syntax is designed to make it easy to express parallel computations. Erlang’s syntactic conventions include:

**Variables:** When variables, or single-assignments (written starting with an uppercase letter like Day and File), acquire a value, that value can not be changed; variables acquire values in successful pattern-matching operations;

**Atoms:** Used to represent constants, they are similar to enumerated types in Java and C and written starting with a lower-case letter; for example, monday, orange, and cat are atoms

**Tuples:** Like structs in C and used for storing a number of items, tuples are written in curly brackets; for example, {Var, monday, 12} is a tuple containing a variable atom and an integer

**Lists:** Used for storing variable number of items, lists are written enclosed in square brackets; for example, [a,X,b,Y] is a list containing two atoms and two variables. A number of primitives are also provided for list manipulation.

Modules are the basic unit of code in Erlang. All the functions we write are stored in modules. A program will often be spread across several modules, each containing functions that are logically grouped together. Modules are stored in files with .erl extensions. Modules must be compiled before the code can be run. A compiled module has the extension .beam. The export directive contains a list of exported functions of the format Function/Arity. These functions are global, meaning they can be called from outside the module. And finally, comments in Erlang start with the percent symbol (%) and span to the end of the line.

3.3. A Simple Sequential Program

The following program computes the area of several geometric shapes.

```erlang
-module(geometry).
-export([area/1]).

area({rectangle, Width, Ht}) -> Width * Ht;
area({square, X}) -> X * X;
```
area({circle, R}) -> 3.14159 * R * R.

The first line declares the name of the module, in this case "geometry". The second line declares the functions to be exported. In this program we state that we want to export function named area whose arity is 1. The next three lines of code describe the function area itself. When we want to call this function we need to use its name and one argument which is a tuple of the form {E1,E2} or {E1,E2,E3}. The first element of these tuples is an atom. The second and third are variables. Whenever some call to the function area matches one of the three lines of the definition, the corresponding on the right of "-> " will be evaluated.

The program can be executed as follows:

1> c(geometry).
{ok,geometry}

2> geometry:area({rectangle, 10, 5}).
50

3> geometry:area({circle, 1.4}).
6.15752

In line 1 we give the command c(geometry), which compiles the code in the file geometry.erl. The compiler returns {ok,geometry}, which means that the compilation succeeded and that the module geometry has been compiled and loaded. In lines 2 and 3 we call the functions in the geometry module. We need to include the module name together with the function name in order to identify exactly which function we want to call.

3.4. The Concurrency Primitives

Erlang supports the creation and coordination of multiple processes and their results by providing the following primitives:

**Pid = spawn(Fun)**

Creates a new concurrent process that evaluates Fun. The new process runs in parallel with the caller. spawn returns a Pid (short for process identifier). We can use Pid to send messages to the process.

**Pid ! Message**

Sends Message to the process with identifier Pid. Message sending is asynchronous. The sender does not wait but continues with what it was doing. ! is called the send operator.
receive ... end

Receives a message that has been sent to a process. It has the following syntax:

```
receive
  Pattern1 -> Expressions1;
  Pattern2 -> Expressions2;
  ...
end
```

When a message arrives at the process, the system tries to match it against Pattern1; if this succeeds, it evaluates Expressions1. If the first pattern does not match, it tries Pattern2, and so on. If none of the patterns matches, the message is saved for later processing, and the process waits for the next message. `receive` automatically queues any out-of-order messages sent to the processes.

### 3.5. A Concurrent Program

```
-module(area).
-export([loop/0]).

loop() ->
  receive
    {From,rectangle, W, H} ->
      From ! {result,W*H};
      loop()
  end.
```

We can create a process that evaluates `loop/0` in the shell:

```
1> Pid = spawn(fun area_server0:loop/0).
<0.36.0>
2> Pid ! {self(),rectangle,10,5}
   {result,50}
3> Pid ! {self(),circle, 23}.
   {result,1661.90}
```

In line 1 we created a new parallel process. `spawn(Fun)` creates a parallel process that evaluates Fun; it returns Pid, which is printed as `<0.36.0>`. In line 2 we sent a message to the process. This message matches the first pattern in the receive statement in `loop/0`. Having received a
message, the process returns the area of the rectangle. Here, `self()` is a function that returns the process identifier of the calling process. This is sent to the spawned process to enable it to return the result.

4. Detecting and Handling Errors

4.1. Erlang View of Errors

Erlang differs from most other programming languages in the way it handles errors. An Erlang system typically consists of large numbers of light-weight processes. It is of no particular consequence if any one of them dies. The recommended way of programming is to let failing processes crash and other processes detect the crashes and fix them. With thousands of processes at one’s disposal one is less concerned about the failure of individual processes than about detection and correction of errors.

Erlang has a safe type system. Data structures are dynamically typed, and it is impossible to create corrupt data structures. Extensive user checking of data structures is unnecessary; since the worst that can happen is an individual process might crash if it performs an illegal operation. The important thing to note is that the crash of one process does not affect any other unlinked process in the system.

The Erlang philosophy is “Let it crash”; in fact, processes that cannot perform the task they were told to do should crash immediately. Another process will correct the error. This is exactly the opposite of defensive programming but leads to a clean separation of interest between code that does the job and code that cleans up an error when it occurs. The system is divided into worker processes that perform computations and supervisor processes that check that the worker processes are behaving correctly.

4.2. Detecting Errors

Erlang has an internal mechanism, or “link,” that provides a form of inter-process error detection and performs as an error-propagation channel. The statement `link(Pid)` sets up a link between the calling process and the process with identifier `Pid`. If process A is linked to process B and process A dies, then an error signal will be sent to process B, and vice versa. The exit signal is a tuple of the format `{EXIT, Pid, Reason}`. Processes normally die when they receive non-normal exit signals from linked processes. The exit signal propagates from a process to all other processes linked to it.

![Figure 3. Exit signals](image)

Processes can trap exit signals by executing the function call `process_flag(trap_exit,true)`. This will set the process flag `trap_exit` and the process becomes a system process. A system process can convert exit signals to messages of the format `{EXIT`,
These messages are saved in the process mailbox just like any normal message. The process can remove this message from the mailbox and examine the Reason. Depending on the reason, the system process might take different actions, such as logging an error message, sending an event message to another process, or starting a new process to replace the one that died. If an exit signal is trapped, it does not propagate further.

In order to build reliable systems, one must be able to remotely detect errors. The following code defines a function that can detect an error in a remote process and perform an action on detecting the error. on_exit(Pid, F) creates a process that monitors the process Pid. process_flag(trap_exit, true) turns the current process into a system process that can trap exit signals. The statement link(Pid) sets up a link to the process Pid. If the monitored process dies with reason Why, the newly created process evaluates the function F(Why).

```
on_exit(Pid, F) ->
    spawn(fun() -> monitor(Pid, F) end).

monitor(Pid, F) ->
    process_flag(trap_exit, true),
    link(Pid),
    receive
        {'EXIT', Pid, Why} ->
            F(Why)
    end.
```

The function on_exit is the workhorse needed to build fault-tolerant code. Using on_exit allows one to build a hierarchical tree of processes. Some processes do the work, and other processes monitor the processes that do the work and fix things up if the worker processes die.

5. Building Reliable Systems

In Erlang, we build robust systems by layering. Using processes, we create a tree in which the leaves consist of the application layer that handles the operational tasks while the interior nodes monitor the leaves and other nodes below them. Processes at any level will trap errors occurring at a level immediately below them. A process whose only task is to supervise children - in our case the nodes of the tree - is called a supervisor. A leaf process performing operational tasks is called a worker.
In well-designed systems, application programmers will not have to worry about error-handling code. If a worker crashes, the exit signal is sent to its supervisor, which isolates it from the higher levels of the system. Based on a set of preconfigured parameters and the reason for termination, the supervisor will decide whether the worker should be restarted.

Supervisors aren’t the only processes that might want to monitor other processes, however. If a process has a dependency on another process that is not necessarily its child, it will want to link itself to it. Upon abnormal termination, both processes can take appropriate action.

6. Dynamic Code Upgrade

When a system’s availability is so critical that taking it offline for fixes and upgrades simply isn’t viable, there aren’t many alternatives. Ultimately, such systems require the ability to perform live upgrades, in which the software is modified as the system keeps running. The ability of Erlang to load new and updated modules during runtime allows systems to run without interruption.

It is easy to update process code via a message if we implement it in such a way that it is ready to receive and execute a new version. As a small example, consider the following code:

```erlang
loop() ->
  receive
    {become, Something} ->
      Something()
  end.
```
This server waits for a message of the form \{become, Something\} where Something is a new definition for the server, and upon receipt, is immediately executed. This way, code upgrade is straightforward.

For example, consider the following server process. The server process has state State and a processing function F.

```
-module(my_server).
-export([loop/2]).

loop(State, F) ->
    receive
        {From, Request} ->
            {Response, State1} = F(Request, State),
            From ! {self(), Response}, loop(State1,F);
        {newFunction, Fn} -> loop(State, Fn);
    end.
```

Now a new processing function can be sent to the server without interrupting it; for example, we could write:

```
F1 = fun(N, State) -> {N*N, State+1} end,
Pid = spawn(fun() -> loop(0, F1) end),
...
... some time later
...
F2 = fun(N, State) -> {N*N*N, State+1} end,
Pid ! {newFunction, F2},
...
```

This new function dynamically upgrades the code in the server.

7. Open Telecoms Platform

OTP is a large set of libraries written mostly in Erlang bundled together with the Erlang distribution. OTP can be viewed as an application middleware package that simplifies writing large Erlang applications. The OTP libraries are an attempt to formalize a large body of design knowledge into workable libraries that provide a standardized way of performing the most common tasks needed to build a reliable system.

OTP is the third total rewrite of a system of libraries in Erlang designed for building telecom systems. The 2010 OTP system includes 49 subsystems, each a powerful tool in its own right. Typical subsystems are mnesia (a real-time relational database), megaco (an H.248 stack), and docbuilder (a tool to make documentation), along with sophisticated analysis-test and analysis tools.
Because a large number of Erlang programs are written in a pure functional programming style, they are able to perform sophisticated analysis and transformations. For example, the dialyzer is a type-checking program that performs static analysis of Erlang programs, finding type errors (if there are any) in them. The test tool QuickCheck generates random test cases from a specification of the formal properties of a program, and the tool Wrangler can be used to refactor Erlang programs.

OTP directly supports the creation of supervisor trees – hierarchies of supervisor and worker processes – via its supervisor module, and OTP’s support for reliable applications depends heavily on these trees. OTP includes significant support for in-service software upgrades. It provides for loading revised modules as well as adding and deleting modules at runtime.

8. Distributed Erlang

Erlang has distribution incorporated into the language’s syntax and semantics, allowing systems to be built with location transparency in mind. The default distribution mode is based on TCP/IP, allowing a node (or Erlang runtime system) on a heterogeneous network to connect to any other node running on any operating system. As long as these nodes are connected through a TCP/IP network and the firewall has been correctly configured, the result is a fully meshed network of nodes, where all the nodes can communicate with each other.

A distributed Erlang system consists of a number of Erlang runtime systems communicating with each other. Each such runtime system is called a node. We can spawn a process on any node and a process on one node can remotely monitor a process running on another node. These nodes, residing on the same or separate machines, help each other by sharing the message and event loads. Should one of the nodes terminate because of a software or hardware error, or simply because of lack of memory, the other nodes take over the traffic, hiding the fault from the end user.

Distributed Erlang applications run in a trusted environment, typically on clusters on the same LAN and behind a firewall. As Erlang clusters were designed to execute behind firewalls, security is based on secret cookies with very few restrictions on access rights. We can create more disparate networks of distributed Erlang nodes using gateways, and if necessary, make them communicate using secure Internet protocols such as SSL.

When supervisor trees are combined with Erlang’s ability to easily spawn processes on other nodes within a distributed system, the result is a substantial yet straightforward foundation for long-running, reliable, fault-tolerant applications. The distribution transparency of Erlang’s process-spawning capabilities means supervisors receive exit signals from worker processes even if the workers run across the network on different hosts. By spreading processes across nodes, our application keeps running even if some of the machines crash or shut down. And with appropriate network redundancy in place between the nodes, our application can continue even if the network breaks.

With distribution built into the language, operations such as clustering, load balancing, the addition of hardware and nodes, communication, and reliability come with very little overhead and correspondingly little code.
9. Erlang and Multicore

The shift to multicore is inevitable. Parallelizing legacy C and Java code is very hard, and debugging parallelized C and Java is even harder. But, the Erlang model for concurrency - separate processes with no shared memory communicating via message passing - naturally transfers to multicore processors in a way that is largely transparent to the programmer, so that we can run our Erlang programs on more powerful hardware without having to redesign them.

Symmetric multiprocessing (SMP) support in Erlang was first developed experimentally in the late 1990s, and is now an integral part of the standard release. Over recent releases, the virtual machine model has evolved from a single monolithic run queue – possibly with processes running on different processors – to a run queue for each processor, ensuring that the run queue is no longer a bottleneck for the system.

![Figure 6. Run queues on a multi-core processor](image)

The goal with Erlang’s SMP is to hide the problems and awareness of SMP from the programmer. Programmers should develop and structure their code as they have always done, optimally using concurrency and without having to worry about the underlying operating system and hardware. As a result, Erlang programs should run perfectly well on any system, regardless of the number of cores or processors.

10. Erlang Distribution

Ever since Erlang was first released into the public domain in 2000, it has been supported by an internal product-development group within Ericsson. Following the release of Open Source Erlang (http://www.erlang.org/), the language spread slowly for several years but has recently seen a dramatic upturn in the number of users and applications. Industrial projects and the formation of new companies using Erlang as core technology reflect the more interesting developments. Erlang can be downloaded from http://www.erlang.org/, including the OTP system and a large number of tools. The latest release is R14B01 (Release 14B, patch level 01), released on December 08, 2010.
11. Existing Applications

Several major product developments are based on Erlang, the largest being the AXD301 an asynchronous transfer mode (ATM) switch developed by Ericsson. Outside Ericsson, Erlang is being used by a large number of start-ups and is the principle technology of several new companies in Stockholm. Not surprising, the leading uses of erlang outside telecom all involve communications and reliable data storage.

The AXD301 ATM Switch

The AXD301, a telephony-class 10–160 Gbps ATM switch, was designed and implemented from scratch in less than three years. At the heart of the AXD301 are more than 1.5 million lines of Erlang code, handling all the complex control logic, and overseeing operations and maintenance. This integrates with about half a million lines of C/C++ implementing low-level protocol and device drivers, much of it coming from third-party sources. Based on the observation of the system performance, Ericsson has managed to achieve a reliability of 99.9999999% using Erlang in the AXD 301 ATM switching system. This corresponds to a down time of 31 msec/year, making it one of the most reliable switches ever made.

Instant messaging:

One problematic area in Internet applications where Erlang has found notable success is implementing instant-messaging systems. An IM system looks at first approximation very much like a telephone exchange. IM and telephone exchanges must both handle very large numbers of simultaneous transactions, each involving communication with a number of simultaneously open channels.

Erlang’s usefulness in IM is demonstrated by three projects:

MochiWeb (http://code.google.com/p/mochiweb): Designed for building lightweight HTTP servers developed by MochiMedia for high-throughput, low-latency analytics, and ad servers, this Erlang library helps power Facebook chat among more than 70 million users;

Ejabberd (http://www.ejabberd.im): Written by Alexey Shchepin, this Erlang implementation of the XMPP protocol is the most widely used open source XMPP server, can reliably support thousands of simultaneous users on a single node and is designed to provide exceptional standards of fault tolerance; and

RabbitMQ (http://www.rabbitmq.com): This Erlang implementation of the Advanced Message Queuing Protocol standard provides reliable asynchronous message passing at Internet scale.

Schema-free databases:

In traditional databases, data is stored in rectangular tables, where the items in a table are instances of simple types (such as integers and strings). Such storage is not particularly convenient for storing an associative array or arbitrary tree-like structure. Databases implemented in Erlang are particularly well-suited for such storage, especially when they interface with some form of communicating agent.

Three notable databases are implemented in Erlang:

CouchDB (http://incubator.apache.org/couchdb/): Written by Damien Katz, “Apache CouchDB is a distributed, fault-tolerant, schema-free document-oriented database. It provides robust, incremental replication with bidirectional conflict detection and resolution, queryable
and indexable through a table-oriented view engine, with JavaScript acting as the default view-definition language;

**Amazon SimpleDB** ([http://aws.amazon.com/simpledb/](http://aws.amazon.com/simpledb/)): This Web service provides database services as a part of the Amazon Elastic Compute Cloud (EC2) and runs queries on structured data in real time; and

**Scalaris**: This scalable, transactional, distributed key-value store has a peer-to-peer architecture for supporting reliable transactions with ACID properties.

CouchDB and Scalaris are open source projects; SimpleDB is a closed source commercial service.

Several applications that have nothing to do with fault tolerance have also gained popularity; for example, Wings ([http://www.wings3d.com](http://www.wings3d.com)), a 3D graphics modeling program written by Björn Gustavsson, and Nitrogen ([http://nitrogenproject.com/](http://nitrogenproject.com/)), a Web-development framework written by Rusty Klophaus, show that Erlang is useful as a general-purpose programming language as well.

### 12. Facebook Chat

The user interface uses a mix of client-side Javascript and server-side PHP and works around transport errors and browser differences. It uses regular AJAX for sending messages and fetching conversation history, periodic AJAX polling for list of online friends and AJAX long-polling for getting messages from server.

The back-end consists of the following: The channel servers are the most intricate pieces of the backend. They're responsible for queuing a given user's messages and pushing them to their web browser via HTTP. They are written in Erlang. Erlang allows a look at the channel server's internals at runtime; we can attach a shell to a running service to find which processes are consuming what resources, to dump the state of particular data structures, and to load new code on the fly.

![Figure 7. Facebook Chat Architecture](image_url)
The chatloggers store the state of Chat conversations between page loads. The presence servers receive periodic batched updates from the channel servers. Each cluster of channel servers keeps an array to record which users are available to chat. The presence servers build a list of a user's online friends by aggregating the information received from channel clusters. These servers support two operations: reading a list of users' statuses from the array, and writing a chunk of information shipped from the channel servers. Each presence machine receives a tremendous amount of information: they regularly receive several bits per Facebook user id, whether the corresponding user is available or not. Both the chatloggers and presence servers are coded in C++.

Glueing together PHP, Javascript, Erlang, and C++ is done through Thrift, which is a software framework for scalable cross-language services development. Thrift translates a service description into the RPC glue code necessary for making cross-language calls (marshalling arguments and responses over the wire) and has templates for servers and clients.

13. Conclusion and Future

Erlang is a language that was designed from the start to solve real and difficult problems, and to do it in an elegant and powerful way. It is highly suitable for building a high-level, concurrent, robust, soft real-time system that will scale in line with demand, make full use of multi-core processors, and integrate with components written in other languages. It handles the difficult parts of concurrency, distribution and reliability with relative ease. Erlang’s libraries and frameworks let developers build systems with extreme availability and reliability.

It must be noted that Erlang and OTP aren’t magical – they won’t automatically make your software extremely reliable. Creating reliable systems with Erlang/OTP still requires knowledge, experience, solid code, thorough testing, and general attention to detail. Nevertheless, because the language was designed with reliability as a foremost concern, the combination of Erlang and OTP definitely has advantages over other common languages when it comes to reliable systems.

Although Uppsala University has for many years led the way with research on Erlang through the High Performance Erlang Project (HiPE), many other universities around the world are not far behind. They include the University of Kent in the United Kingdom and Eötvös Loránd University in Hungary, which are both working on refactoring tools. The Universidad Politécnica de Madrid of Spain together with Chalmers University of Technology and the IT University (both in Sweden) are working on Erlang property-based testing tools that are changing the way people verify Erlang programs.
14. References


   http://www.docstoc.com/docs/24583486/Erlang

   Accessed on 17-01-2011

[7] Letuchy, E. Erlang at Facebook


   http://www.infoq.com/news/2008/05/facebookchatarchitecture

   Accessed on 08-02-2011


   http://www.insidefacebook.com/2008/05/20/details-on-facebook-chat-architecture/

   Accessed on 08-02-2011