OTP in Server Farms

Hal Snyder
Vail Systems
570 Lake Cook Rd, STE 408
Deerfield, IL 60015
hal@vailsys.com

Michael Bruening
Vail Systems
570 Lake Cook Rd, STE 408
Deerfield, IL 60015
mike@vailsys.com

Martin Logan
Vail Systems
570 Lake Cook Rd, STE 408
Deerfield, IL 60015
martin@vailsys.com

ABSTRACT

Ericsson's OTP (Open Telecom Platform) offers a number of attractive features if you want to provide a variety of information services on a network with high availability, scalability, and extensibility. However, the major uses of OTP have been in closed, relay-rack systems, rather than in clusters of general-purpose servers. We describe issues we encountered while bringing up applications on OTP in the latter environment at a computer telephony company.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

Keywords

functional programming, Erlang

1. WHY USE ERLANG AND OTP?

Vail Systems uses computers to answer and originate calls for our customers. During the course of a call, various network services are consulted to connect a human caller with a nearest available service provider, determine the most economical outbound circuit available, update calling card information, and so on.

We often need to develop and deploy custom network resources on a short timetable. Examples are:

- Proxy for remote custom (non-SQL) database queries needed for servicing calls in progress.
- Routing of calls to preferred call center based on customer's real time configuration changes.
- Least cost routing of outbound calls.
- Allocation of outbound ports to running telephony applications.
- Spooling and monitoring of call events.
- Proxying of VoiceXML server access to allow redundancy and failure-mode content.

We saw the following advantages for basing these services on Erlang and OTP:

- Distribution without agonizing over protocols and socket-level programming.
- Rapid development in Erlang for those comfortable with functional programming. It seems that Erlang offers for distributed programming the same sort of advantages one gets from scripting languages in a conventional setting; we see analogies with the discussion in [7].
- Multiple lightweight processes without the pitfalls of using threads - see [6].
- Mnesia - an integrated, fast, replicated database.
- Large amounts of code, including design patterns and administrivia, already done for us with OTP's infrastructure.
- An overall structure that lets us work with architectural issues in terms of dynamically changing sets of consumers and producers, as opposed to dealing with individual client and server computers.

We completed a successful pilot project in Erlang, without making full use of OTP; that is to say, we did not use behaviours, supervision, releases, and higher-level modules such as inets. It was clear to us that we wanted the additional gains of using full OTP and not just the Erlang language - see [2].

Our platform consists of a number of servers running some variant of Unix. A single server may run many applications, some OTP-based, others not - determined by the hardware, software, and network resources available on that server. Some applications will be long-lived, probably outliving most individual servers that run them; other applications are temporary based on the short-term needs of a customer. We do not want to take shared resources offline or reboot servers when changing the application mix. In addition, all servers need to connect to generic network systems for OAM (Operations, Admin, and Maintenance).

We will next discuss our current use of OTP. In effect, this recounts those problems that were tractable from our point on the learning curve. After that, we'll describe aspects of the platform that continue to be a challenge.

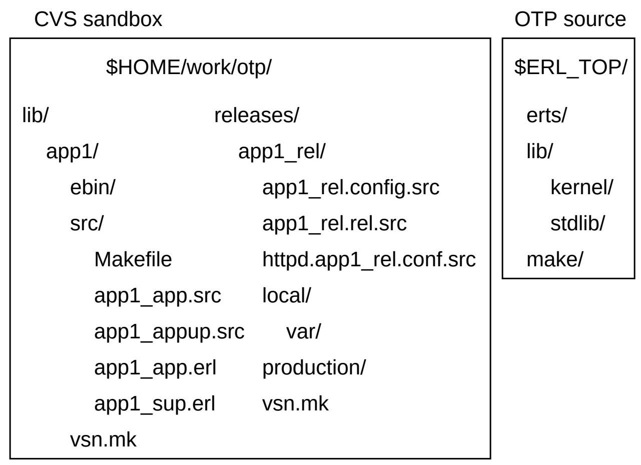

Figure 1: File Layout for Development Environment

2. PRESENT OTP PLATFORM

2.1 Build System

Our OTP build system has the following properties:

- Version control based on CVS. A heavy-duty version control system such as ClearCase was deemed a poor fit for our software engineering operation.
- Supports two environments, development and production. In development mode, a programmer runs OTP and gets results in his home directory. In production mode, a pseudouser runs OTP. Details include placement, ownership, and permissions of files, selection of output paths, TCP/UDP port numbers, and Erlang cookies.
- Frees most programmers from the task of dealing with .app and .rel files and such.
- Supports all the following operating systems: Solaris, Linux, FreeBSD, OpenBSD, NetBSD.

A simplified view of the directory structure for the development environment is shown in Figure 1. Our source code is organized into applications under the `lib` directory: `app1`, `app2` and so on, as well as one directory of utility functions (or "library modules"). Nodes are configured within the `releases` directory. In the example, a release is shown whose main purpose is running only one of our applications, namely `app1`. We expect to configure nodes which will run multiple applications, but have not done so yet. Running the `Makefile` for `app1_rel` will produce `.boot`, `.script`, `.config`, and startup shell files in the `local/` directory for testing. Logs and on-disk data tables during test runs are written below this directory as well. The `production/` directory is a staging area for scripts and configuration files that will be installed in a global area on production servers.

OTP source code releases include GNU autoconf information and supporting constructs to allow for differences among Unix variants and server configuration. Our Erlang code needs to work in the same varied environment. Rather than maintain yet another autoconf subsystem, we simply include the necessary variable settings from files that

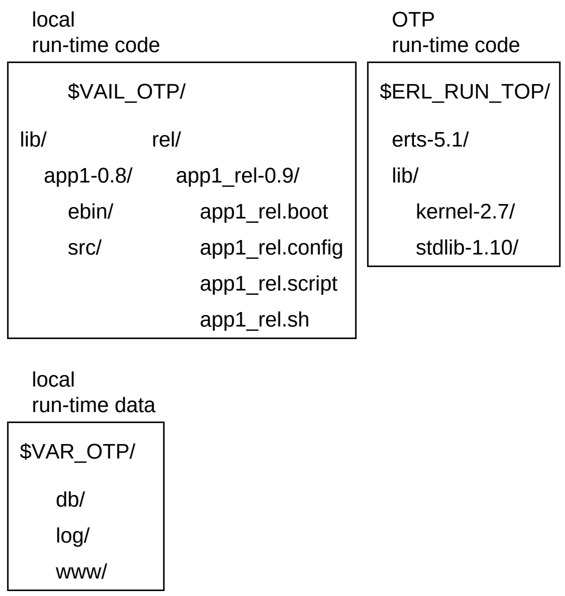

Figure 2: File Layout for Production Environment

were generated within the OTP source tree by the OTP `configure` script.

Our library application, `vaillib`, like OTP's `stdlib`, consists of several sets of files, each set providing some feature set. Keeping all the modules in a single application reduces the number of entries in the `lib` directory and simplifies structure of releases at the expense of clutter and unwieldy file names within `vaillib/src/`. Modules in the library application are intended to be usable by more than one application within the same node; they are library modules in the sense of [4] and do not participate explicitly in a supervision tree.

Directory structure for the production environment is shown in Figure 2. Running `make install` with appropriate permissions from `lib/app1/src` and `releases/app1_rel` will install all the files needed, other than OTP run-times. At present, a snapshot is created manually of the installed files for deployment on production servers.

2.2 Configurable Parameters

Most applications need some sort of configuration information - which files, port numbers, URLs, or servers to access and such. Placing such information in local configuration files has several problems:

- An application has to be told to parse the configuration file whenever a change is made. Often this task requires taking the application offline, especially if resources are allocated statically during initial scan of the configuration file.
- Changes have to be replicated to all servers running an instance of the application being configured.
- Configuration changes can be made only by persons with sufficient authorization and understanding of file format.

We are now putting most configuration information in a distributed database. Some tables are replicated on all nodes which use them; other tables are kept on a small number of servers due to the high ratio of writes to reads. Only a minimum of rarely changed boostrap information remains in the startup files.

An `inets` interface is used to allow operations staff to maintain configuration data. In some cases, the `inets` interface is invoked by another non-OTP web server, allowing us to embed OTP status and forms in the customer web interface, which uses mod_perl and non-OTP-based authentication and session tracking.

2.3 Resource Discovery

An issue related to configuration in a distributed system is, how do producers and consumers of a resource find each other? Examples of resources are telephony ports, spoolers, and various proxies.

We would like to avoid manual configuration of lists of producers or consumers on every host. Producers of a resource should register their presence automatically when coming online and should be dropped from the registry when they go offline. Consumers of a resource should have timely information about available producers.

We can visualize two tables: a table of producers, whose keys are resource types and whose values are lists of nodes offering that resource; and a table of consumers, whose keys are resources and whose values are lists of nodes requiring that resource.

The system is implemented using an application which runs as a gen_server on each node, maintaining local data corresponding to the description above. The local data is kept in ets tables.

There is a chicken-and-egg problem associated with resource discovery: how do new nodes find the resource discovery system? A new OTP node starting up uses DNS to look for SRV records[3] for the OTP cluster to which it belongs[1]. Static configuration for the node is thus limited to 1) the server's `resolv.conf` file, and 2) the name of the cluster.

2.4 Logging

The main purpose of logging on the platform is to assist in diagnosing problems. Other systems exist for collection of data for billing purposes and for real-time monitoring. For each subsystem on the platform we would like to define a set of basic transactions. Whenever one of these transactions occurs, a single log record is written. We use local code patterned on logging modules within OTP, to provide features described below.

Log format should be amenable to processing with general-purpose Unix commands and scripting languages. Therefore, text is written as opposed to binary formats, with one line per record.

Logs are truncated automatically to avoid filling the host filesystem. The maximum size of a log file and number of historic log files are configurable. Files are rotated in a manner compatible with today's smarter `tail` utilities: `filename` is always the most recent log, with `filename.0`, `filename.1`, and such being progressively older archives—and, a running `tail -f filename` can continue to function after log file rotation.

[1]Probably `inet_res:getbyname` or equivalent should be made part of the documented programmer interface; there's no simple way to query SRV records without it.

2.5 Monitoring

Monitoring a system component answers the two questions, first, is the component available, and second, how much work is it doing? Monitoring services must also issue alerts if components fail or are at risk of failure.

In order to see if a component is available, a set of test transactions is run at regular intervals (cron jobs) and operations staff are alerted if the result is not what is expected. Within OTP, we register event handlers to notify us in case of such events as partitioning of replicated mnesia nodes.

Each system component keeps counts of the number of various transactions it has performed since startup. We have not implemented a full-fledged SNMP management platform, with ASN.1, BER, custom MIBs, and so forth. It has been more efficient to use an `inets` interface into each OTP application. Current counts are retrieved via HTTP, then external utilities such as RRD[5] track historical data, create charts and summarize results.

2.6 Interaction with Non-OTP Systems

Here are the interfaces we use between OTP and non-OTP:

2.6.1 C++ Shared Object

A shared object interface links OTP to SIP gateways, proxies, and application servers which are implemented in C++.

The OTP side of the interface is named the Erlang request broker. It listens on an agreed-upon TCP port for work to do. We require an OTP node on the same server as each external program, and have the request broker listen on the IP loopback address. Therefore, discovery of the OTP cluster by external programs is not required.

Communication consists of M,F,A (module, function, argument list) requests by the external program and results from OTP. Requests are ASCII encoded with a request identifier and a flag indicating whether the request is synchronous (wait for OTP to complete and return a result), asynchronous (fire and forget), or hand-off (wait for the request broker to acknowledge the request, but don't wait for full remote processing).

2.6.2 UDP

Some servers, such as legacy systems, do not run OTP but host applications that make use of services now on the OTP platform. The query sent to OTP and the response each consist of a single UDP packet. In case of a timeout, secondary servers are used. As the connection takes place on a local network, the timeout interval is kept to less than a second.

2.6.3 HTTP

As stated above, our principal interfaces for configuration and monitoring use `inets`. We would like to do a telnet version of the generic interface offered by `inets`, with command history and completion, as time permits.

2.6.4 SMTP

Some OTP applications generate email for notifications to operations staff. The `email` module contributed by Joe Armstrong[1] is useful for such things.

2.6.5 ODBCserver

One application uses OTP's `ODBCserver` to proxy queries sent to remote Oracle installations.

2.6.6 *Filesystem*

As mentioned above, some non-OTP tools are used for processing logs written by OTP processes.

3. UNSOLVED PROBLEMS

3.1 Resuming Replication after Partitioning

A significant barrier to more extensive use of Mnesia is the lack of convention for resynchronizing tables automatically after partitioning. The problem is more difficult if you do not want to distinguish nodes as masters or slaves.

Most tables that are replicated could be resynchronized by playback of transaction logs from previously detached peers, in that nearly all updates take the form of adding to or subtracting from some sort of counter.

3.2 Dynamic Application Mix

The current release system for OTP, with .boot and .script files, requires knowing at startup which OTP applications will run on the node. We would like to add and drop applications while a node is running, retaining ease of specifying supervision and restart strategy, yet without restarting the node, and without losing changes should the node or server reboot.

We are just beginning to look into the problem. Even with such a capability in place, there could be times when it is advantageous to run multiple OTP nodes on a single server.

4. CONCLUSIONS

OTP is a powerful set of tools for building reliable, distributed systems. The learning curve is shallow—one can build production-grade systems early in the process of learning how the Erlang language works and the components of OTP interoperate. Increasing familiarity with OTP brings steadily broadening horizons to its range of applicability. A downside of this learn-as-you-go process is that, more often than not, on completing a new subsystem, it becomes clear that there was a much better way to do it.

In deploying OTP on a server farm, several adaptations will likely be necessary. Some of these adaptations are: construction of a build system for development and production, specifying the nature of interaction with existing infrastructure including logging and monitoring, implementation of interfaces to non-OTP systems, and allowing for nodes to come and go smoothly in Mnesia replication.

Some of the code in such an installation will be reusable across multiple applications. For us, the generic components include management of configurable parameters, resource discovery, Unix-friendly logging, monitoring, Mnesia join-up, and a TCP interface to the C++ part of the platform.

As we continue to add applications, we would like to see the underlying OTP system and generic code grow in the following ways: migrate from static to dynamic supervision scheme on individual nodes, allow Mnesia nodes to resynchronize automatically on recovering from a network partition, provide command-line interfaces for operations staff, and keep historical queues of transaction counts for charting and threshold checks.

5. REFERENCES

[1] J. Armstrong. email-1.0.tgz. `http://www.erlang.org/user.html#email-1.0`.

[2] F. Cesarini and M. Rémond. The migration from Erlang to OTP: A Case Study of a Heavy Duty TCP/IP Client-server System written in Erlang. In *Proceedings of the Seventh International Erlang User Conference, Stockholm, Sweden*, September 2001. `http://www.erlang.se/euc/01/`.

[3] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Internet Engineering Task Force, Feb. 2000, `http://www.ietf.org/rfc/rfc2782.txt`.

[4] P. Högfeldt. Code Replacement Explained, 1999. `http://www.ericsson.com/cslab/~peter/code-replacement.html`.

[5] T. Oetiker. About RRDtool. `http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/`.

[6] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited Talk, 1996 USENIX Technical Conference, January 1996. `http://www.cc.gatech.edu/ccg/people/rob/software/threads/ousterhout_threads.html`.

[7] J. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, March 1998. `http://home.pacbell.net/ouster/scripting.html`.