

Ronja—A Java application platform

Magnus Karlson

Ronja is a Java development framework based on the TSP that enables the rapid development of, and reduced maintenance cost for, servers in the new telecommunications infrastructure. The main design goal for the framework has been to use standard, off-the-shelf products, in order to facilitate the sourcing of external components and to take advantage of economies of scale.

The Ronja framework includes a high-level, high-availability abstraction library, which means that programmers need not get bogged down with the low-level problems of creating these programs—another benefit that translates into shorter development time.

The author describes requirements for scalable, high-availability applications, the hardware architecture, Ronja's view of hardware and execution environments, the structure of a Ronja application, and framework services.

A component-based architecture

Ronja is one of the components in the newly released platform for the development of central transactions and control servers in the telecommunications infrastructure. This platform, called TSP, contains a large set of components, including operation-and-maintenance (O&M) components, operating systems, middleware/frameworks, signal stacks, and hardware. Because they have well-defined, standardized interfaces, these components can be combined in many different ways to form suitable platforms for different servers in telecom networks. The idea of defining a component-based architecture with standardized interfaces is to ensure that the architecture is future-proof by making the components easily exchangeable should a better solution be found (Figure 1).

The first product to use Ronja is the new radio network server (RNS).¹

Design goals

Ronja is a middleware/framework that has been developed to exhibit standard telecom characteristics, such as:

- high availability—the middleware must be able to handle loosely coupled multi-processor systems in which fault tolerance and reliability are based on $n + 1$ redundancy of processing elements—that is, the software must be able to handle handover, failover, and so on (by failover, we mean the functionality for restarting a failed application unit);
- scalability—the middleware must be able to support the distribution of an application over multiple processors; and
- hot-code replacement—the software

must allow all parts of the system, both hardware and software (including the middleware and operating systems), to be updated while the system is running, and with minimum degradation of performance.

To minimize development costs as well as the cost of owning applications, Ronja was designed with very distinct design goals, such as

- making it easy to reuse technology;
- using standard processors and operating systems
 - to allow for a high level of external sourcing;
 - to target and develop systems that have the same environment;
- making high-availability and clustering functionality as transparent as possible for the applications;
- supporting several programming languages;
- facilitating the porting of software; and
- allowing the use of mainstream software development and testing tools and environments.

Trying to achieve these goals is the basic reason for choosing Java as the main implementation language. The design of high-level, transparent high-availability/cluster functionality requires the high abstraction and protection layers provided by Java. Ease of porting is inherent in Java, and new development and test tools are also available very early for a language such as Java.

The current implementation of Ronja is based on the high-availability and clustering software used in Ericsson's general packet radio service (GPRS) product portfolio. However, the idea is to ensure that any such framework can be used without affecting future application programs. Note: the choice of high-availability and cluster environment defines the level of supervision that can be applied to parts that are not directly developed for the platform. Although Ronja defines methods of communicating with programs or parts of programs written in other languages, the underlying framework must provide support for starting, stopping, upgrading and supervision if the parts are to exhibit full telecommunications characteristics for the complete application.

Requirements for scalable high-availability applications

Unfortunately, there is no magic way for any middleware to create a scalable high-

BOX A, TERMS AND ABBREVIATIONS

3GPP	Third-generation Partnership Project
API	Application program interface
CI	Capsule instance
GEM	Generic Ericsson magazine
GPRS	General packet radio service
IRP	Integration reference point
JVM	Java virtual machine
O&M	Operation and maintenance
PI	Processor instance
RNS	Radio network server
Thread	Very light-weight process

availability application from an ordinary program written in traditional languages (C, C++, Java, and so on) unless the program is designed from the start to comply with some fairly obvious design principles:

- the algorithms used must be distributed;
- several instances of the part of the application that carries out the work must be able to coexist, or work in parallel, or both; and
- if the program parts are to be fail-safe and upgradeable while in service, it must also be possible to replicate or save necessary status information so that a new version can pick up the information and continue after a handover or switchover is effected.

These functions can be more or less hidden in the platform. After all, Ronja's purpose is to make this as simple as possible for the programmer.

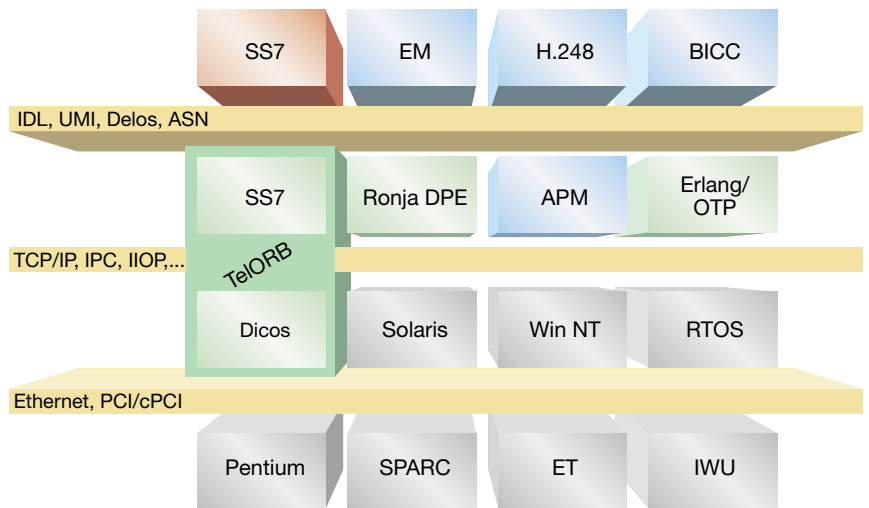


Figure 1
Component architecture of the TSP.

Hardware architecture

Our approach to solving the problems of high availability (scalability, reliability, and in-service upgrading) is to build multiprocessor systems that do not have any single point of failure. In practice, this means that all components, processors, networks, network interfaces, power supplies, and cooling fans are at least duplicated or able to exploit $n+1$ redundancy. In TSP, we use a new hardware architecture called the generic Ericsson magazine (GEM). In a system built up using Ronja, this calls for a magazine (subrack) that is filled with SPARC-based processor boards, Ethernet switches and bridges or routers together with any special hardware that a specific server node might need (Figure 2).

Ronja's view of the hardware and execution environments

A cluster contains several processor instances (PI—the processors available in the cluster) that can be used by an application. Any number of capsule instances (which is an aggregate term for a processing environment) can be started on these processor instances. In the Ronja framework, a capsule normally corresponds to a Java virtual machine (JVM) for Java execution. For other languages, a capsule is usually the equivalent of a program (C, C++), or where appropriate, a virtual machine (such as Perl or

Figure 2
Typical hardware subrack for Ronja.



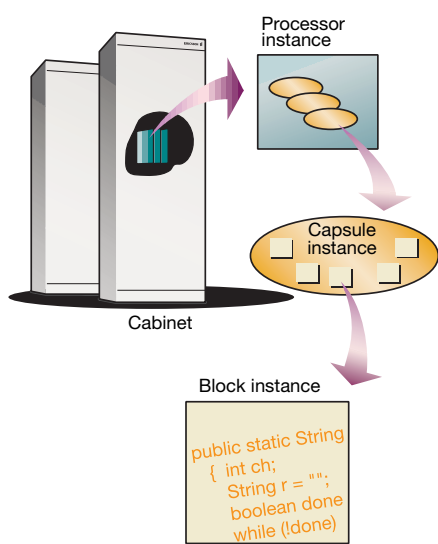


Figure 3 Execution environment.

Erlang). Applications are then mapped onto these capsule instances (Figure 3).

Structure of a Ronja application

A Ronja application is split into smaller portions called blocks, which are the smallest entities that can be maintained by the framework. By a “maintained entity” we mean one that can be loaded, started, stopped, supervised, upgraded, and version-handled. A block can be written in any language, which means that an application can consist of blocks written in different languages. One block (the root block) is mainly responsible for communicating with the framework and for defining or supervising the distribution structure and functions of the application. Because Ronja is a Java platform, the root block must be written in Java to take full advantage of the features that Ronja can provide (Figure 4).

The role of the root block

Ronja’s high-level, high-availability cluster library and templates become very useful in the root block. Most application-management functions are more or less transparently handled in the template by means of standardized profiles that the application programmer uses to define how the applications will behave. When the root block is being defined, the programmer can choose between profiles that are supplied by Ronja or developed by the project to manage the distribution, upgrading, migration and restarting of the application or parts of the application. When an application is started in Ronja, it is actually the root block that starts. The root block then starts the rest of the application by invoking the appropriate profiles so that the application is mapped onto the available processor instances and into different capsules instances. After that, the root block mostly plays the role of a supervisor. It is informed of any changes that affect the application while it

is running—such as the addition or removal of new hardware, or the failure of some block that is part of the application, or requests to update the software—and takes the appropriate action by using the matching profiles to remedy the situation or comply with requests. The templates and profiles defined by Ronja are sufficient for nearly all applications; only in very rare or special cases should it be necessary for the programmer to write unique profiles.

The application block

Ronja puts only minor requirements on the implementation of “normal” application blocks. Basically, the block needs only three interfaces to start, stop, and to receive information or data. A template block in Ronja helps the programmer to design the program.

Sourcing external components

When Ronja was designed, ease of sourcing was a major goal. There are several reasons for this:

- Ericsson already has many software components that can be reused;
- specialist companies produce considerable amounts of useful software; and
- a lot of the *de facto* standard software used in, say, Internet applications is not produced in-house at Ericsson.

To take advantage of sourced software, there must be as few restrictions and rules in the framework as possible. For “stand-alone” programs, such as a Web server, object-request broker, and so on, the high-availability framework’s concept of applications can often be used to raise the level of supervision (restart if a failure occurs). To include more sophisticated error detection or failover functionality, small supervision programs need to be written that can start the sourced application. However, we often want to source technology or software at a more embedded level, such as parts of a program

(communication stacks, conversion libraries, simple network-management-protocol agents). In these cases, the framework must not put any restrictions on the kind of system calls or language construction that might be used. Ronja imposes no such restrictions. If the software lacks some of the functions (starting and stopping, for example), a wrapper template that emulates some of these functions can easily solve the problem. In some extreme cases—for example, if the sourced component cannot be stopped or is using threads in an uncontrolled way—the sourced component can be run in a separate capsule (a capsule can always be stopped forcibly), so that it can be stopped or upgraded in a controlled manner.

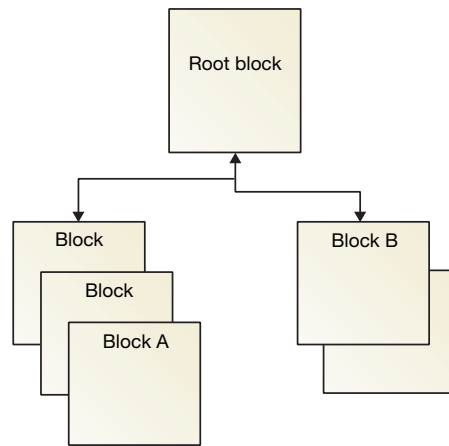


Figure 4
Application structure.

Framework services

The framework also contains much of the generic functionality that is needed by telecommunications applications. Typical examples of this kind of functionality are timers, naming, communication, status information (of the system), event channels, log files, and configuration data. Whenever functionality is defined in some usable standard, Ronja generally complies with the standard—in this context, we can mention the IRP-compliant alarm-and-notification service agent, which is available in the platform. IRP, or integration reference point, is a 3GPP specification.

It is also possible for a development project to add its own special services to the framework by using a defined framework-extension application program interface (API). As a rule, this is only necessary for very special functionality that cannot be programmed using more commonplace programming constructs, such as library functions or modules and interprocess communication. For normal functions, this is not the recommended way of solving problems. This is because framework services cannot be upgraded on the fly without restarting the capsule. By contrast, ordinary blocks can be upgraded while a capsule is running.

Conclusion

The Ronja framework with high-availability/cluster software delivers standard telecommunications characteristics to applications that use commercial operating systems and hardware. Plug-and-play functionality, scalability, uninterrupted execution and failover can be transparently implemented in the applications using the high-level, high-availability library and templates provided in the framework. An ordinary set of telecom and high-availability/cluster library functions and services along with the templates allows for very high productivity when applications are being designed. The use of standard operating systems and languages enables developers and testers to employ the latest mainstream tools in their work, thus ensuring high productivity in later phases of development projects as well as during maintenance. Since the framework can be run on standard workstations, no special hardware is needed during most phases of development and testing. This reduces the need for special (costly and difficult-to-maintain) hardware installations in the course of projects.

TRADEMARKS

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

REFERENCES

Musikka, N. and Rinnbäck, L.: Ericsson's IP-based BSS and radio network server. *Ericsson Review* Vol. 77(2000):4, pp. 224-233.