Presented at SCI Europe ‘98, 28-30 Sept. 1998, Bordeaux, France

Design and implementation issues for an SCI cluster configuration system

Tore Anders Aamodt
Scali AS
Hvamstubben 17
N-2013 Skjetten, Norway
mailto:taa@scali.no

Abstract - This paper highlights the importance of and requirements for an SCI cluster configuration system. A number of cluster related design and implementation issues have been raised and resolved through the work on Scali's own SCI cluster configuration system: ***ScaConf*****. This has resulted in both the working application and some generally applicable methodologies which are presented.**

I. Introduction

Scali AS is in the business of selling SCI [8] cluster based supercomputing. Manual configuration of even small SCI clusters has proved too error prone and time consuming, and it rapidly gets worse with increasing cluster size. To a certain extent this can be alleviated by using script based tools, but especially in operational situations where high availability and utilisation is crucial, one needs a more flexible configuration system with possibilities for automatic reconfiguration as a method of error correction. System administrators at end-user sites must be able to maintain an SCI cluster without massive amounts of expert knowledge. It is therefore evident that an important factor for the mainstream acceptance of SCI cluster based computing is the availability of more user-friendly tools for configuration, monitoring and resource management. The configuration software described in this document is under development by Scali as part of the SISCI project (ESPRIT 23174) [5].

II. Requirements

The most important requirement for an SCI cluster configuration system is the ability to perform as many configuration tasks as possible from a single point of interaction through software. One should be able to view the SCI cluster as a single entity and perform actions on it as such. The need for repetitive remote logins to all nodes for system administration tasks should be minimised. More specifically this includes:

- Distributing and setting nodeIDs for all SCI adapters.
- Setting up optimal routing tables for a selected interconnect topology.
- Error detection, handling and reporting.
- Automatic re-configuration in case of node failure.
- Simple and flexible API to allow for integration with other tools.
- A choice of user interfaces, GUI or ASCII to run remotely as well as locally.
- Full functionality for remote configuration (customer support).
- Support for heterogeneous clusters and environments.
- Ability to cope with changes in SCI hardware, interconnect topology etc.
- Distributed system load and low resource usage.
- Support for extended hardware related tasks, like power and console switching.

III. Design

A. System Architecture

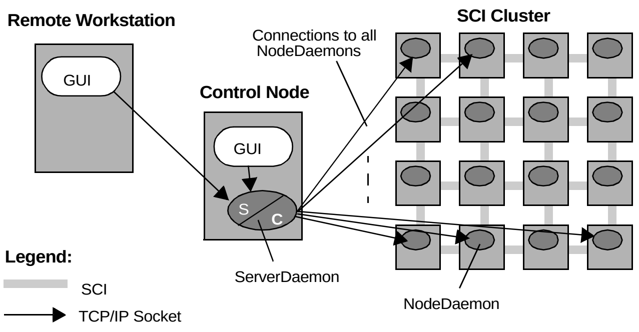


Figure 1: ScaConf System Architecture

Figure 1 shows the organisation of the configuration system. The rectangles represents physical computers, the ovals represents processes and the lines represents network connections. The SCI cluster shown indicates a 4x4 2D torus. In a sense the configuration system has a two-step client server architecture. The configuration ServerDaemon is a server for the user interfaces, and at the same time all the configuration NodeDaemons act as servers for the client side of the server daemon. Hence the ServerDaemon has been indicated to have a server “S” and a client “C” side.

The NodeDaemon is running on each of the nodes in the SCI cluster and serves as an interface between a defined message format and the native SCI driver API. Differences between APIs and SCI adapters will be handled transparently by the NodeDaemon so that the ServerDaemon can be designed to perform its tasks in a generic manner without worrying about differences in hardwarea at the other end of the socket connection.

Whereas the NodeDaemon may look as a simple command translator, the ServerDaemon is far more complicated. It must:

- Build and maintain a database of all nodes in the system with their current state.
- Contain algorithms for routing of all supported topologies.
- Propagate user interactions to nodes.
- Propagate node state changes to all connected interfaces.
- Analyse error situations and reconfigure the cluster according to set priorities.

As could be seen from figure 1, we have chosen to run the configuration server daemon on a separate node outside the SCI cluster. Although not strictly necessary the reason for this is that the server node may require a special configuration with a display adapter and additional hardware for console and power switching. Using a dedicated control node also has the nice side effect that all cluster nodes will remain equally configured and fully interchangeable. Since the ServerDaemon should accept multiple connections to its server side a security scheme is built in to avoid configuration conflicts.

The user interfaces only have to relate to the defined message format and could be implemented as graphical or text based applications. Using only socket communication between the ServerDaemon and the user interfaces, they should run equally well on the local server node as on a remote machine.

B. Object Models

We used OMT [2] as a design method. It was also clear that we would use a mix of C++ and Java for the implementation. Using OMT allowed the classes to come out of the analysis and design phase to be implemented in either language..

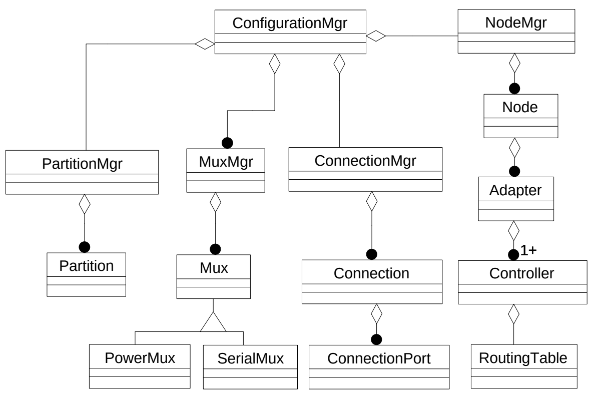


Figure 2: Confgiuration data object model

To illustrate this we have chosen the configuration database which is used by the ServerDaemon to keep the system status information. Here we managed quite successfully to use a physical object model of the system and translate it into a usable object style database. The object model in shown in figure 2 clearly resembles the physical system. Attributes and operators have been removed from the figure for clarity and space saving considerations, but all classes contains attributes for all values of interest, access methods for the attributes and convenience functions where appropriate. Classes with names ending in “Mgr” are managers which means they keep a collection of other objects. For the C++ implementation later we decided to base the managers on the standard container classes of Standard Template Library (STL) [9] to conform with the standards and gain portability.

The interface to the database is provided by the top level class is the configuration manager ConfigurationMgr. The ServerDaemon (not shown) therefore holds an instance of the ConfigurationMgr class. Simple “persistence” was gained by deciding that all classes of the configuration manager had to implement the methods *WriteGuts*(binstream) and *ReadGuts*(binstream) for full save and restore from a binary stream, and *Print*(stream) and *Parse*(stream) for reading and writing ASCII in a configuration file compatible format. The latter methods should also be available as insertion << and extraction >> operators.

An interesting detail about this part of the design is that the NodeMgr class including Node, Adapter and Controller classes is used both by the ServerDaemon as part of the ConfigurationMgr database and by the Java graphical user interface client because it needs a place to put a copy of the node status information. In the first case it is implemented in C++ and in the second it is implemented in Java. By sticking firmly to the design the two implementations will have a compatible binary format. This means that when the Java GUI requests nodemanager information the ServerDaemon (C++) can simply dump the contents of the nodemanager into a message using *WriteGuts*() while the GUI (Java) can restore the nodemanager from the message using *ReadGuts*(), regardless of the platform either one is running on. Refer to the next sections for more information about the link between binary streams and messages.

IV. Implementation

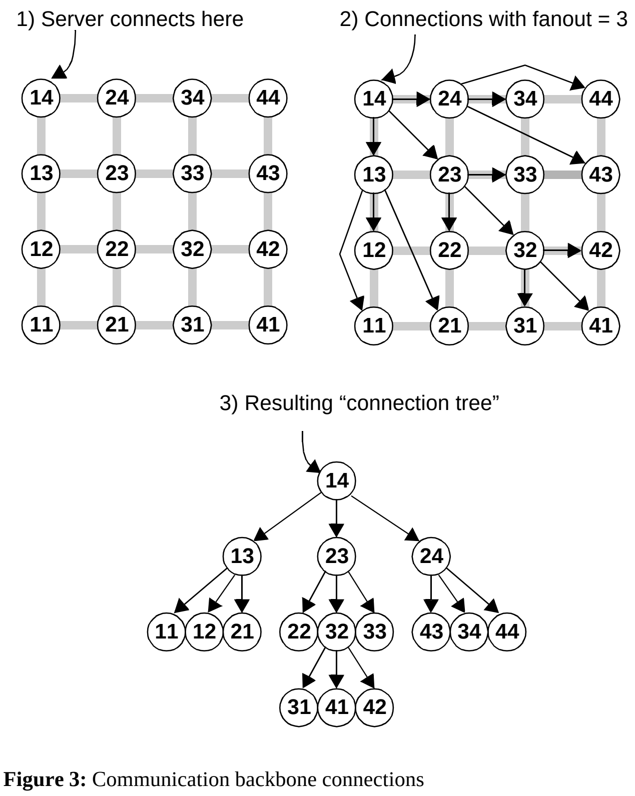
A. Communications backbone: ScaComd

It became evident that a system for setting up connections between a central process like the ServerDaemon, and applications or daemons running on every cluster node like the NodeDaemon, should be generalised as it would be useful for more applications than ScaConf. Several of the requirements for the configuration system could also be met by the proper implementation of such a system. This lead to the separate task of developing the Scali communications daemon: ScaComd.

Being a little concerned about the resource usage on the control node with big clusters (notably file descriptors) we wanted the daemon to have enough local intelligence to provide a fan-out/fan-in mechanism whereby the Server-Daemon could connect to any one node in the cluster with a request to set up communication channels to all the other nodes using only a single connection into the cluster.

The fanout mechanism works like this: The initiator, here the ServerDaemon connects to a randomly selected node in the cluster and sends a INIT_REQUEST message which contains a fanout factor and the nodes to which connections are required.

This newly appointed “top-node” will then randomly select a number of nodes depending on the fanout factor and the remaining number of nodes, connect to them and send out new INIT_REQUESTs, for the remaining nodes. This is repeated until connections to all nodes have been established. The result is a “connection tree” structure where the width of the tree is determined by the fanout factor. This is perhaps better explained by figure



1) Server connects here
2) Connections with fanout = 3
3) Resulting “connection tree”
14
24
34
44
13
23
33
43
12
22
32
42
11
21
31
41

Figure 3: Communication backbone connections

The communication load will of cause be higher closer to the root of the tree since all communication to the server will run through the parent node from its siblings. In the example in figure communication between the server and node 41 will pass through nodes 14,23 and 32 on the way, in fact communication to all nodes will pass through node 14. We do not anticipate this to be a problem because the system is merely designed for control type messages which are relatively small in size and low in frequency. For other types of applications with higher bandwidth requirements there are other solutions like our high performance MPI 1.1 implementation: *ScaMPI*. The communications backbone on the other hand should be lightweight and use only the standard TCP/IP protocol over the local Ethernet. This to be usable for control and monitoring of the more specialised interconnect technology: SCI.

If another application should need a connection to all nodes the random selection of the “top-node” will ensure that the communication load will be distributed evenly over the cluster. Say if the next application connects to node 44 as “top-node” this connection tree will spread out entirely different from the first with other nodes taking the greater part of the load.

Error detection and a degree of correction is also built into the system. A broken connections is taken as the indication that something is wrong. When an error is detected by the communication daemon on the node immediately above the failed one, two things happens: First a re-connection cycle is entered where the node tries to re-establish connections to all nodes which were below it in the connection tree. Then after a timeout period NODE_DOWN messages are generated for all node that could not be reconnected and sent up to the initiator. The NODE_DOWN messages will then be extracted by the initiator and appropriate action taken. Using figure 3 again as an example: If node 32 fails this will be discovered by node 23 which then tries to re-establish connections with 32,31,41, and 42. After the timeout node 32 could still not be connected and a NODE_DOWN message for node 32 is passed up to the server which has connected to node 14.

As always when faced with the possibility of a heterogeneous computing environment one needs to adress the byte order (big-endian vs. little-endian) problem. Scali is already using Sun’s SPARC and Intel IA-32 based clusters running the Solaris 2.6 operating system so for us the endian-problem represents a “clear and present danger”.

This implied a message format with a defined byte order and field size. Since Java was already in the picture boasting portability we decided to make a binary stream class on the C++ side which would be compatible with the Java DataInputStream and DataOutputStream classes (full names are java.io.DataInputStream and java.io.DataOutputStream). Apart from being well documented [6] this also solved the problems of interfacing to the Java GUI, cross platform compatibility and not the least defining a format of our own. The smallest usable subset of read and write operations from the DataStreams was found to be Char, Int and UTF (UTF is a string encoded using a modified UTF-8 format). The resulting binary stream class ScaBinStream forms the basis for the communication backbone message class: ScaComMsg. By only allowing access to the message through ScaBinStream’s read and write operations platform independent messaging has been obtained.

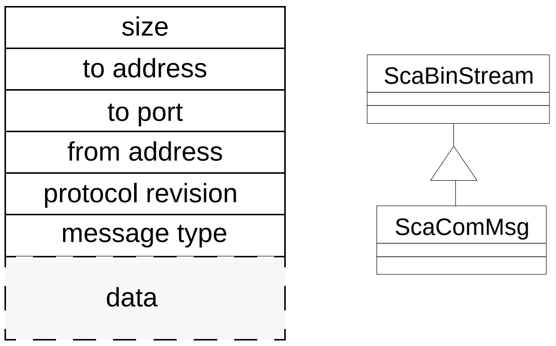


Figure 4: ScaComMsg message format and object model

The ScaComMsg has a 6 field header format as can be seen in figure 4. All 6 fields are full 4 byte integers and although this may seem as an overkill it makes the encoding and decoding simpler. The *size* field gives total message size including data. All addresses are full 32 bit IP addresses. The target for a message which is an application or daemon running on a node can be uniquely defined by the *to address* + *port*. It is assumed that all applications/ daemons using the ScaComd communication backbone has a unique well known port. The *from address* is used for replies and the *protocol version* to avoid errors from incompatible message formats. The *message type* is used by the application to interpret the data block following the header. Since the ScaComMsg now is-a ScaBinStream and all classes in the configuration database were required to provide binary save and restore to a ScaBinStream any part of the database can easily be dumped to and restored from a message. This means that we now have a way of transparently sending configuration data between daemons, servers and clients regardless of platform or location.

It is the idea that other and more specialised messages could be derived from the general ScaComMsg, adding more header fields after the message type as needed. The ScaConf message format SC_Message is one example. One may discuss the usability of heterogeneous clusters with non-compatible system architectures since this raises even bigger challenges in application compatibility and maintenance. But we decided that the configuration system should be able to handle this anyway. Later this has proved to be correct as we often run the ServerDaemon on a different architecture from the cluster.

B. The Configuration file

One of the design goals of ScaConf has been to avoid redundant and unnecessary information in the system. As much as possible should be determined run-time and deduced from existing information. Even so there are things which must be fed into the system. Notably which components the system is made up of and how they are physically connected. This information has been collected in a human editable ASCII file called the configuration file. The configuration file is deliberately kept as small and simple as possible to minimise the probability of errors.

The first thing the ServerDaemon does when it is started is to read the configuration file. Since the operation of ScaConf is based on the correctness of the contents of the configuration file it runs a few sanity checks on the data before it is accepted. For example the connections section which describes the SCI network topology there are checks for connection duplicates, endpoint duplicates (since the SCI adapters has uniquely defined endpoints), broken rings, loop-backs, dead-ends and unconnected rings. If the check fails the ServerDaemon refuses to start. This catches problems like common typing errors at an early stage.

C. Fault tolerance - reconfiguration

Scalability is advantageous in terms of capacity and performance, however a cluster configuration system must address the issue of increased error rate caused by scaling the system to a large number of nodes. Being a collection of autonomous nodes, a cluster survives failures by avoiding dependencies. All nodes remains operational except the failing ones.

In general there are two classes of network topologies, *indirect* and *direct*. In an indirect network all nodes are connected to a centralized switch and adding fault tolerance largely involves to ensure the switch handles a node failure properly. In a direct network as used by the Scali computers, each node is part of the network and adding fault tolerance means that action must be taken to reconfigure the network hardware of every node in the cluster in case of node failure. When a node is detected as faulty or erroneous its fail-state will fall into one of tree categories:

- Reachable (1)
- Unreachable with power on (2)
- Unreachable with power off (3)

From an SCI and routing point of view fail-states 1) and 2) will be treated similarly. The difference lies in how difficult it is to get the node back to operational state. Failed state 2) usually requires a power-off-reset with implications on the network. The Dolphin SCI adapters [3] we use are able to keep the ringlet operational as long as there is power on. The link controllers [4] even allows for configuration over SCI from other nodes on the ringlet which means that routing can be set up for nodes that is otherwise unreachable. This implies that a failed node still having power will not impose any constraints on the routing. If power is lost the ringlet is broken and useless. The difference in impact on a system is of cause significant and varies with the interconnect topology. ScaConf so far handles ring and 2D torus topologies.

In the case of a ring topology ScaConf will handle fail-states 1) and 2) by setting the routing to avoid the failed nodes. In case of fail-state 3) the entire ring will be “down” and there is nothing to do about it but wait till the faulty node has been repaired.

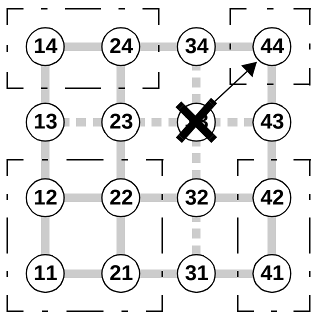


Figure 5: Failed node fractures cluster

With a 2D torus topology there are more options: Consider the situation in figure 5 The familiar 4x4 2D torus is routed using the X-Y routing scheme and node 33 fails. All nodes on the ringlets of node 33 are deemed unavailable. This leaves the usable parts of the machine fractured as four small partitions. Since we are dealing with a 2D torous structure we can freely select a starting point for the ringlets. In this situation ScaConf would logically remap the faulty node to the corner of the torus and generate new routing tables to enable the remaining nodes to be used as one big partition as illustrated in figure 6

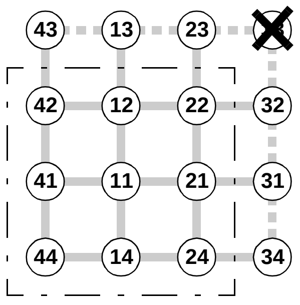


Figure 6: Failed node remapped to corner of torus

Note that the remapping is a purely logical one and performed only by changing the routing tables for the operational nodes. No end-point, that is SCI NodeIDs, are changed. This has the very nice side effect that a running application that was halted by the failure can continue as if nothing happened provided it was not using the failed node.

The previous method with a simple rerouting still leaves too many working nodes unused. The nodes on the horizontal and vertical ringlets of the failed node could still be reached via the other operational ringlets. Using even better routing algorithms like the turn model [1], it will be possible to utilise all but the failed node as can be seen in figure 7.

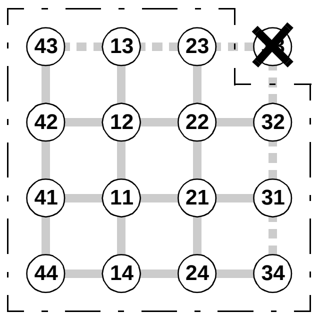


Figure 7: Remapped failed node with enhanced routing

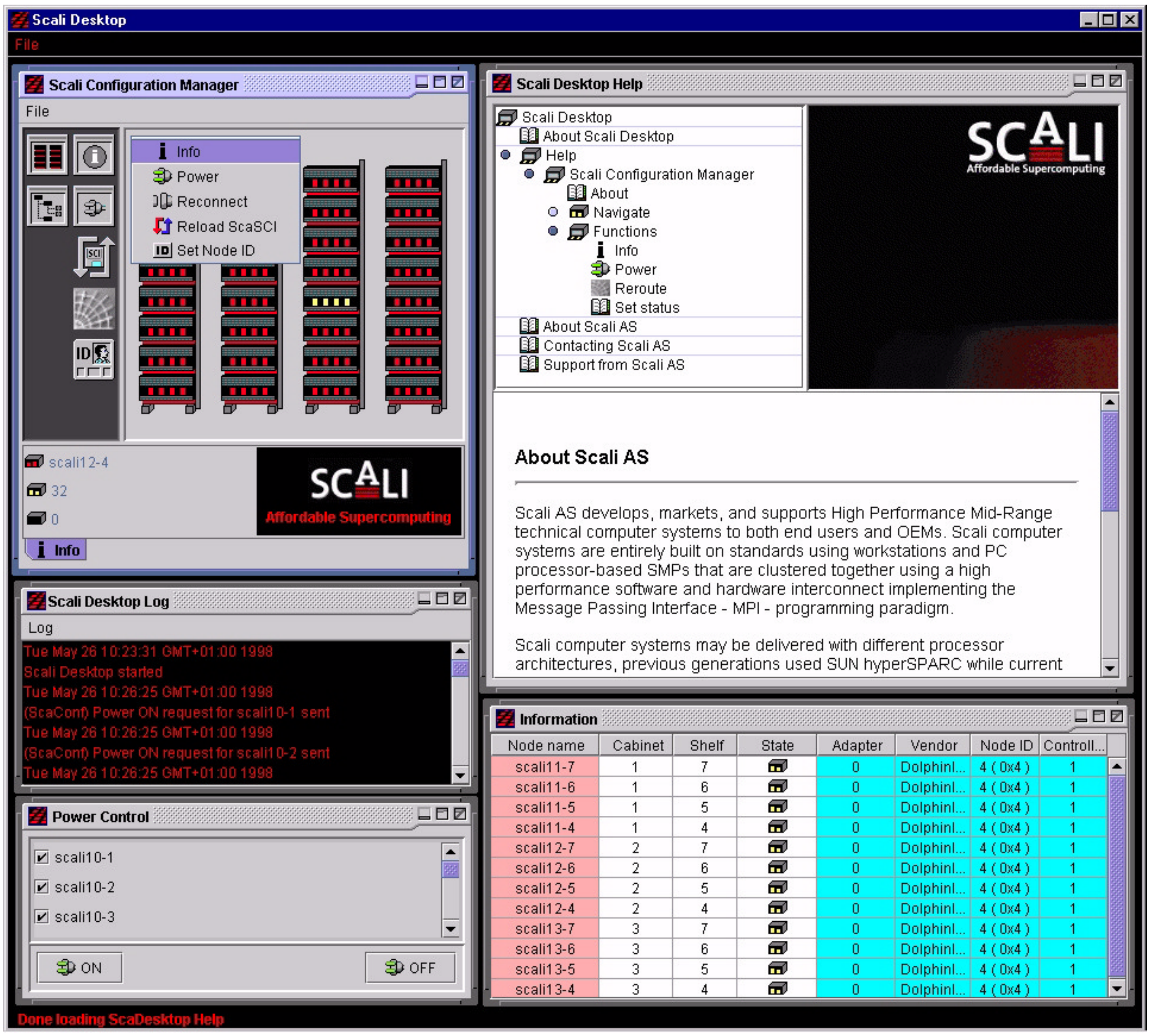
D. A Java user interface

With portability, cross platform and network capabilities as selection criteria Java was the natural choice for the ScaConf graphical user interface. Based on JDK 1.1.3 [6] and the Swing 1.0.2 GUI toolkit [7] from Sun Microsystems, Inc. the interface was developed under Windows NT using Borlands JBuilder. Without modification it now runs equally well on Windows NT and on Solaris 2.6 on both SPARC and Intel x86 platforms.

The Swing toolkit provides an option of having a virtual Java desktop inside the native window system. We used this to make a customised Scali Desktop on which ScaConf and other Scali applications lives. This concept offered many advantages, firstly the look and feel of the applications would be completely similar down to window operations across every platform. Secondly with the introduction of more applications in the Scali suite the Scali Desktop will be really neat way of organising things. Also as part of the Scali Desktop we add centralised functions like a logging service and a help file browser.

When the ScaConf GUI connects to the ServerDaemon the first message sent is a request for the NodeManager. When the NodeManager message is received a picture showing all nodes in the cluster and their states is drawn in the main window. Thereafter all available functions in ScaConf can be invoked in usual GUI style with multiple selection selections popup-menus etc. A screendump of ScaDesktop with the ScaConf GUI can be seen in figure 8.

.



```
Tue May 26 10:23:31 GMT+01:00 1998
Scali Desktop started
Tue May 26 10:26:25 GMT+01:00 1998
(ScaConf) Power ON request for scali10-1 sent
Tue May 26 10:26:25 GMT+01:00 1998
(ScaConf) Power ON request for scali10-2 sent
Tue May 26 10:26:25 GMT+01:00 1998
```

About Scali AS

Scali AS develops, markets, and supports High Performance Mid-Range technical computer systems to both end users and OEMs. Scali computer systems are entirely built on standards using workstations and PC processor-based SMPs that are clustered together using a high performance software and hardware interconnect implementing the Message Passing Interface - MPI - programming paradigm.

Scali computer systems may be delivered with different processor architectures, previous generations used SUN hyperSPARC while current

Node name	Cabinet	Shelf	State	Adapter	Vendor	Node ID	Controll...
scali11-7	1	7		0	Dolphinl...	4 (0x4)	1
scali11-6	1	6		0	Dolphinl...	4 (0x4)	1
scali11-5	1	5		0	Dolphinl...	4 (0x4)	1
scali11-4	1	4		0	Dolphinl...	4 (0x4)	1
scali12-7	2	7		0	Dolphinl...	4 (0x4)	1
scali12-6	2	6		0	Dolphinl...	4 (0x4)	1
scali12-5	2	5		0	Dolphinl...	4 (0x4)	1
scali12-4	2	4		0	Dolphinl...	4 (0x4)	1
scali13-7	3	7		0	Dolphinl...	4 (0x4)	1
scali13-6	3	6		0	Dolphinl...	4 (0x4)	1
scali13-5	3	5		0	Dolphinl...	4 (0x4)	1
scali13-4	3	4		0	Dolphinl...	4 (0x4)	1

Done loading ScaDesktop Help

Figure 8: The Scali Desktop with ScaConf GUI, help file browser and log system

E. Power and console switching

At first the ability to do power and console switching from software may seem as an unnecessary luxury but it has proved to be absolutely crucial for the usability of any but the smallest system.

Firstly if a node (or worse tens of nodes) should fail to boot you would want to have a look at the console to see what went wrong. Since it is not economical to fit display hardware in dedicated cluster nodes you'll probably have to manually plug a VT100 into the serial port of every node that went wrong. This is bad enough even if you work close to the physical location of the machine but it makes remote maintenance at this level impossible. In addition, the bigger the system, the more likely it lives in a special air-conditioned room further away from your own desk. The introduction of a software controlled serial port selection through serial port multiplexers efficiently provides direct access to console output from any node in the cluster.

The last resort for fixing a node which has become unreachable is to do a power off-on cycle. The same arguments applies here as it did for the console output. If you are physically close to the machine it is just inconvenient and time consuming. If you are further away it becomes intolerable or impossible.

So, by supporting both power and console switching ScaConf provides remote configuration and maintenance abilities for everything except the physical cabling - which is impossible anyway.

The added information needed by ScaConf to do the power and console switching is really minimal and kept in the configuration file. Every node must have a console and power switch name and port number for the switches it is connected to, and there must be entries for the switches themselves. The entire console/power switching logic is built into the ServerDaemon.

Since power switches and serial port multiplexers rarely have more than eight channels ScaConf also supports multiplexer hierarchies for systems larger than 8 nodes. The information needed to “route” through the multiplexer hierarchies can be readily extracted from the configuration file. With 8 channel multiplexers each new level introduced in the hierarchy effectively increases the maximum number of reachable nodes eightfold, i.e. three levels gives a maximum of 8 x 8 x 8 = 512 nodes. So capacity will not be a problem in this area.

V. Conclusion and further work

The design and implementation of ScaConf has provided Scali with reusable solutions to cluster related development issues. By initiating the development also of ScaComd, ScaConf contributed even more to the Scali clustering software platform than first anticipated. ScaConf is now making its way into Scali’s product line and in this sense it will follow a normal product life cycle with more features being added with each release. As with any software product of this complexity and size the possibility for improvements and new features seems endless. Therefore the biggest challenge will be to react to customer demands, identify the most important improvements and attend to those first. A few items which already have been identified are listed below

- More topologies, Currently ScaConf handles plain rings and 2D torus SCI topologies. In the very near future we foresee the need to handle 3D torus and counter-rotating ring topologies.
- Handling of more failed nodes
- Support for SCI switches
- Handlling of multiple clusters
- A (graphical) configuration file editor
- Automatic performance optimization of routing

Acknowledgment

The author wish to thank his co-workers at Scali for their contributions: Håkon Bugge and Rolf Welde Skeie came up with the original idea and are continously contributing to the design. Hans Westgaard Ry implemented ScaComd and Stian S. Johansen designed and implemented the Java GUI and ScaDesktop.

References

[1] Christopher J. Glass and Lionel M. Li, “The Turn Model for Adaptive Routing”, ACM pp. 278-287. 1992

[2] James Rumbaugh et al. “Object-Oritented Modelling and Design”, Prentice-Hall, 1991.

[3] Dolphin Interconnect Solutions AS, “PCI-SCI Cluster Adapter Specification”, Version 1.1, May 1996

[4] Dolphin Interconnect Solutions AS, “Link Controller LC-2 Specification”, Version 1.3, June 1997

[5] Stian S. Johansen, Rolf W. Skeie, Anders Liverud, “System Integration Requirements”, Intremal SISCI report: D511-20-190598 version 2.0, May 19 1998

[6] Sun Microsystems, Inc., “JDK 1.1.6 Documentation”, (online) http://java.sun.com/products/jdk/1.1/docs/index.html, 1998

[7] Sun Microsystems, Inc., “Swing Package Summary”, (online) http://java.sun.com/products/jfc/swingdoc-api/overview-summary.html, 1998

[8] IEEE, “IEEE Standard for Scalable Coherent Interface (SCI)” *ANSI/IEEE Std 1596-1992*

[9] Bjarne Strøstroup, “The C++ Programming Language, Third Edition”, Addison Wesley, September 1997