GSI OPERATING SOFTWARE MIGRATION OpenVMS TO Linux

R. Huhmann, G. Fröhlich, S. Jülicher, V. RW Schaa, GSI, Darmstadt, Germany

Abstract

The current operating software at GSI has been developed over a period of more than two decades using OpenVMS now on Alpha-Workstations. Parts of this complex software will have to be integrated within the control system of the new FAIR accelerator. To ease future maintenance a migration to Linux is considered a precondition. For porting to Linux a set of libraries and tools have been developed covering the necessary OpenVMS (referred to as VMS in the paper) system functionality. The interoperability with FAIR controls applications is achieved by adding a simple but generic middle-ware interface to access the ported software in a service-like manner from modern Java applications.

GENERAL ASPECTS

Presently the operating software runs on a cluster of DEC-Alpha machines, Alpha is today a brand of Hewlett-Packard. The computers' OS is OpenVMS. The user interface is realized by X-Window and Motif based clients, beside some hardware display and control units like LED, knobs, keys and so on. As a first step of migration the formerly used relational database system has been replaced by a ORACLE 10g SQL system on a Linux cluster. The operating software is mainly written in F77 with DEC Fortran extensions, lots of VMS specific system calls are used, the X11 event scheme is embedded in the VMS system event scheme. The interprocess communication is based on VMS system API and on GSI proprietary APIs realized with VMS system functionality. The whole system is embedded in a complex VMS environment.

On Linux, modern techniques can be used for future developments. An increase of maintainability and a unique platform for operating shall be achieved. Additionally, using Linux makes it much easier to connect to systems which will be used for the new FAIR accelerator facility. Migration shall take place on application level, i.e. applications are ported to Linux but the VMS run time environment is not to be simulated, especially no DCL (DEC Command Language) or related stuff shall be part of the migrated system. Due to the huge amount of existing applications software porting shall be achieved with minimal source modification. Additionally run time data from the operating software shall become accessible via interprocess communication for modern Java applications, again with minimal source modification.

TECHNICAL ASPECTS

This section covers main aspects of porting VMS specific Fortran software to Linux. Due to the limited size of this paper it is impossible to mention all aspects or to go in much detail.

DEC Fortran Extensions

DEC Fortran was established as a quasi standard which exceeds and improves the Fortran77 specifications. It covers a lot of compiler directives and language extensions to increase possibilities for Fortran programmers. An example is the type `STRUCTURE` to create composed data objects like `struct` in `C`. Other important examples are the compiler directives `%REF, %LOC, %VAL` which enable the Fortran programmer to use pointers. Luckily most of the extensions are supported by main Fortran compiler manufacturers. We tested successfully the Intel Fortran compiler for Linux. Some missing features, e.g. testing for existing arguments in the parameter list of a function, have to be replaced by suitable Fortran90 constructs.

VMS System Libraries

In order to retain the source code structure essential parts of the VMS Runtime Libraries had to be implemented on Linux. Although some commercial products offer a very complete and well documented implementation of the VMS runtime library for Unix based systems the evaluation of this products failed due to some incompatibility in technical detail which would have implied major source code modification.

To establish the needed VMS functionality a generic plain-C API (called *ix*) was developed using POSIX and SYSV system calls. Above *ix* a limited VMS System API is set up (called *vx*), supplying a Fortran Interface and VMS status return codes. This library is still under construction and offers currently the most important parts of the VMS System Service calls used at GSI.

Events The support for event driven architectures is a key feature of the VMS System Services (beside asynchronous system callbacks). Virtually any asynchronous system task may notify completion via event propagation. In this paradigm the application waits in a main receive loop for any event which has been set up and supplied to an asynchronous execution branch. When the event is triggered the waiting application is interrupted and supplied with the information which event was triggered to process any appropriate action.

To rebuild such a paradigm on Linux the *ix*-API utilizes the *pthread_cond* functions for signaling and waiting for events. Each *ix*-event object gets a list of listeners. A listener corresponds to a set of *ix*-events which can be waited for (`SYS$WFLOR`), implemented with *pthread_cond_timedwait()*. Triggering an event (`SYS$SETEF`) leads to signaling all its listeners via *pthread_cond_signal()*.

Timer VMS allows to couple timers to events. An elapsed timer triggers the event which can be received in the application.

On Linux the *ix*-API realizes timers with a Unix system call *select()* in one separate thread. The *select()* function covers synchronous I/O multiplexing, i.e. listens for I/O conditions to emerge (e.g. socket ready to read). Additionally a timeout can be given when the function shall return to caller. The VMS timers (`SYS$SETIMR`) are inserted in a time-sorted list and *select()* is called with a appropriate timeout to meet the first timer in the list. When elapsed, the corresponding event will be triggered, i.e. the waiting thread is signaled.

In parallel the same *select()*-thread is used in the *ix*-API to cover the asynchronous I/O of the VMS System Library and the event based interprocess communication.

Global Section Of course, single node global sections (`SYS$MGBLSC`, `SYS$CRMPSC`) are easily to realize with Unix shared memory.

But there are some specialities to take care of. It is common use on VMS to map an initialized memory region (F77 COMMON BLOCK) to a named global section. If it is the first mapping, the value in memory is that of the initialization and of course may be altered afterward by the application. Any further mapping from other applications even to initialized memory will yield the current values of the global section.

We use POSIX file-based shared memory, but for counting mappings we additionally have a dummy SYSV shared memory, its *id* is stored in the file header (see figure 1). This is to ensure that for the first mapping the contents of mapped memory is saved to shared-file. For any successive mapping the memory is overwritten by shared-file contents.

Message Compiler On VMS a tool called Message Compiler processes special-format text files with mappings from system-wide error condition numbers and names to message texts. It generates code which is linked to applications. The error numbers are used in a signal generating routine (`LIB$SIGNAL`) which prints the corresponding message (optionally processing supplied arguments) and causes a program exit in case of a severe exception condition.

On Linux the Message Compiler was implemented to be compatible with the original message text files. It creates a library for resolving the message texts and builds include files for Fortran and C with the defined error condition constants and their values. Additionally it generates a Linker Script to retain the possibility to use the error condition numbers as symbolic constants declared as `EXTERNAL` and which values can be retrieved in Fortran code by the `%LOC` directive resolved at linking time.

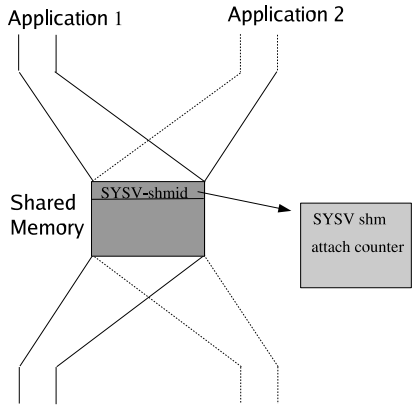

Figure 1: Global section

X11 Integration

For the migration of Fortran X11-Clients one needs a wrapper library to map the so called Non-C-Binding X functions to the Linux X library. This is straight forward and reduces mainly to differences in the call strategy for Fortran and C. Only a few functions differ slightly in parameters and functionality. For Motif it is even simpler since VMS uses the same C-API. After compilation (which produces either lower or upper case function symbol names with appended underscore) one has to map with the tool `objcopy` the symbol names to the correct names found in the Motif library.

Further procedure depends upon the usage of X11 and Motif event processing. Currently at GSI two paradigms of X11 programming are used on VMS:

The first one adds a timer with lowest acceptable interval to the set of utilized VMS events. All triggered VMS events are processed in a central event receive loop. In the timer elapsed routine all pending X11 events are processed in a *do-while* loop using *XtAppPending()*, *XtAppNextEvent()* and *XtDispatchEvent()*. On Linux this has been replaced by a thread which consumes and processes pending X events. Processing of these events is mutex synchronized with the processing of the *ix*-events.

The other paradigm utilizes the Motif main event loop *XtAppMainLoop()* for X11/Motif and VMS system event processing. That implies the usage of *XtAppAddInput()* which links a data source to a user specified callback. Parameters are operating system dependent. On Unix based systems the source parameter is a file descriptor used as source of data, on VMS it is an event id. The problem is solved by a own implementation of *XtAppAddInput()*. It supplies the native Motif *XtAppAddInput()* with a single in-

ternal pipe as data source and creates a mapping from the supplied event id to the supplied callback. A general hook function for the *ix*-events is set which writes the id of triggered events to the internal pipe. When data is available in the pipe the supplied native *Xt*-Callback reads the event id from the pipe and dispatches the user callback found in the mapping table.

GSI Specific APIs

The implementation of some functionality hidden by GSI-specific APIs is completely replaced on Linux.

Interprocess Communication The GSI-API for Interprocess Communication on VMS is written in Modula-2 and transfers binary data packets between applications. It utilizes VMS system mailboxes, events, communication servers and raw Ethernet network communication.

To provide its functionality on Linux the implementation is replaced by peer-to-peer network communication through TCP/IP sockets and utilizes the *ix*-event architecture. To implement the addressing scheme of the network packets (i.e. a unique name set up at application runtime) a server registers a mapping of the name to TCP/IP host and port information.

Alarm System The implementation of the GSI-API for device and process alarming will be replaced.

Device Access The GSI-API for device access on VMS is written in Modula-2 and realized by communication servers and raw Ethernet network communication.

On Linux the API implementation is replaced by a wrapper to the new Corba based object oriented device access interface.

Interface to Java Applications

This section describes an approach to connect the operating software after migration to Linux to the currently designed Java environment of the new FAIR control system. Keeping that in mind the following constraints should be regarded:

- minimal source code modification for existing Fortran applications
- flexible data structures on application level
- simple and generic interface to avoid API adaptations
- C/C++/Fortran Server-API and a Java Client-API

To fit these constraints a so called universal value architecture (*uv*) was developed which implements a kind of directory service. Using *uv* each Fortran application is able to represent specific value structures as a tree analogous to a hierarchical file-system (see figure 2). A tree-node is called file. Beside other properties each file has a name and a value, which is either of primitive data-type (boolean, byte, integer, long, double, string), or an array of primitive data-type, or a list of files which is called folder, or a special value-type (event-queue, mount-point). Each node in the tree is named in a unique way by its absolute path name starting with the root-node. The *uv* architecture implements internally a publisher-subscriber pattern to inform clients (subscriber) about changes of the server (publisher) which may be initiated by another client or by the server locally.

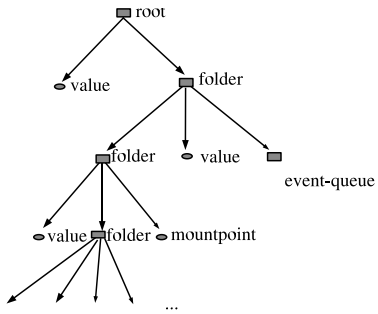

Figure 2: Example of a *uv* structure

Publisher The Server-API for the Fortran application allows to set up and change the value tree-structure (like adding files and folders to a file-system) and to set and change locally the numerical values of the leave-nodes. Services like tree transfer of a read result, value changes by clients, sending value change notifications to clients, processing structure changes and subscriptions are hidden in the implementation of the API and transparent for the application. Value changes initiated by a *uv*-client are notified to the application via callback.

Subscriber The Client-API for the Java application is to read the value tree structure or subtree structures. It subscribes to folders for receiving structure changes or to leave-nodes for receiving numerical value changes or to event-queues for receiving event objects. It allows to set values of leave-nodes of an *uv*-server.

CONCLUSIONS

The status of the project has shown the technical feasibility of migration respecting the constraints of minimal source code modification and interoperability with Java applications. The developed libraries and tools build a solid basis for migration. Further efforts will be made to integrate the current Operating Software into FAIR controls.

REFERENCES

[1] OpenVMS RTL Library (LIB$) Manual, OpenVMS System Services Reference (SYS$) Manual

[2] Linux manual pages and for POSIX: IEEE Std 1003.1, 2003 Edition, Standard for Information Technology – Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 6, Copyright (C) 2001-2003 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group