RTES Demo System2004

Shikha Ahuja[1], Ted Bapty[1], Harry Cheung[2], Michael Haney[3], Zbigniew Kalbarczyk[3], Akhilesh Khanna[3], Jim Kowalkowski[2], Derek Messie[4], Daniel Mosse[5], Sandeep Neema[1], Steve Nordstrom[1], Jae Oh[4], Paul Sheldon[1], Shweta Shetty[1], Long Wang[3], Di Yao[1]

Vanderbilt University[1], Fermi National Accelerator Laboratory[2], University of Illinois at Urbana-Champaign[3], Syracuse University[4], University of Pittsburgh[5]

shikha@isis.vanderbilt.edu, bapty@isis.vanderbilt.edu, cheung@fnal.gov, m-haney@uiuc.edu, kalbar@crhc.uiuc.edu, akhanna@uiuc.edu, jbk@fnal.gov, dsmessie@syr.edu, mosse@cs.pitt.edu, sandeep@isis.vanderbilt.edu, steve.nordstrom@vanderbilt.edu, jcoh@ecs.syr.edu, paul.sheldon@vanderbilt.edu, shweta.shetty@vanderbilt.edu, longwang@crhc.uiuc.edu, dyao@isis.vanderbilt.edu

Abstract

The RTES Demo System 2004 is a prototype for reliable, fault-adaptive infrastructure applicable to commodity-based dedicated application computer farms, such as the Level 2/3 trigger for the proposed BTeV high energy physics project. This paper describes the prototype, and its demonstration at the 11th IEEE Real Time and Embedded Technology Applications Symposium, RTAS 2005.

1. Introduction

The Real Time Embedded Systems (RTES) project [1] was established to develop reliable and fault-adaptive middleware in support of large, dedicated, heterogeneous computational resources. The archetypical application for the RTES product was the proposed BTeV high energy physics experiment [2], with its embedded hard real-time Level 1 processing farm (originally to have been approximately 2500 DSPs), and commodity rate-based real time Level 2/3 computer farm (originally to have been a like number of commercial personal computers).

A prototype system was developed by the RTES project for the Level 1 farm, and was presented during the SC2003 Supercomputing Conference [3]. This paper describes the prototype developed by the RTES project in support of the Level 2/3 farm.

This work was supported in part by the National Science Foundation Information Technology Research Program (number #ACI-0121658)

2. Environment

A number of hardware and software aspects of the prototype were not developed by the RTES project, and are enumerated here to set the RTES development in context.

The hardware was a heterogeneous Linux processor farm [4] consisting of:

- 15 (500 MHz P3) dual-CPU worker computers
- 84 (1 GHz P3) dual-CPU worker computers
- three (500 MHz P3, 1 GHz P3, and 2.8 GHz P4) cluster manager nodes

These were recycled from other computing applications at Fermilab, for use as a prototype of the BTeV Level 2/3 trigger farm. The age and infirmity of (some of) these processors provided an effective setting for the testing of reliable software middleware.

In addition to conventional software found on most Linux systems, the RTES Demo System employed:

- Elvin [5] publish-subscribe messaging middleware, which was used as the primary communications channel between the RTES components
- Ganglia [6] cluster (computing) toolkit, to display long time-scale CPU utilization

Also, Matlab [7] was run on Windows laptop(s) to provide GUI widgets (buttons, text boxes, etc.) to control and display the System.

3. Demo System 2004

The following subsections describe the components developed for this project.

3.1. Domain Specific Modeling Languages

Model-based approaches for designing large scale systems can mitigate complexity associated with design management and component integration. Embedded systems should be modeled using domain-specific modeling languages (DSML) specialized for a particular domain [8]. For a large scale system, complexity can be further reduced by using multiple interdependent DSMLs to describe different aspects of the system. In Demo System 2004, we developed a modeling tool suite for specifying the relevant aspects of the prototype system. This tool is composed of a set of narrowly focused DSMLs integrated through a system level language implemented using Generic Modeling Environment (GME) tool [9]. Models constructed using DSMLs were then automatically translated to useful implementation artifacts using generation tools. This concept is illustrated in Figure 1. The following paragraphs provide an overview of these modeling languages along with their generated artifacts.

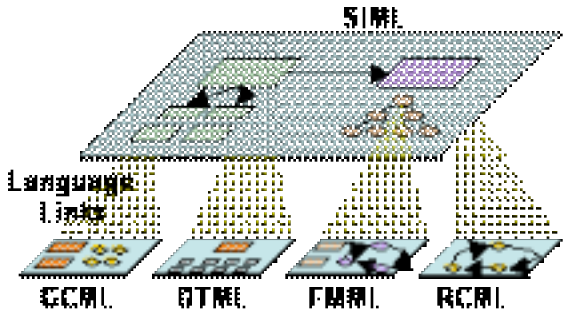


Figure 1. Relationships between the System Integration and GUI Configuration, Data Types, Fault Mitigation, and Run Control Modeling Languages

System Integration Modeling Language (SIML) is a language used for high level specification of the system. It allows the capture of system components, component hierarchy, component interactions within the system, and system configuration information. SIML also serves as the highest level language through which models of other languages are accessed using a Link type. The target modeling language is identified by the attributes of a Link. The generated artifacts are used to configure, deploy, and build the system.

Data Types Modeling Language (DTML) is a language used for specifying simple and composite data types for modeling message. It hides the implementation details of the underlying communication protocol by generating message marshalling and de-marshalling code that are used by all of the components of the system for communication.

Fault Mitigation Modeling Language (FMML) is a language used for specifying the behavior of fault management components in the system. Users can create custom fault-mitigation behaviors to suit their needs using a generalized notation with additional domain specific features. Complete generation of source code including object classes, middleware API, and communication API calls to implement the behavior of these fault management components, is performed from these models.

GUI Configuration Modeling Language (GCML) is a language used for layout and design of user interfaces for control, monitoring, diagnosis, and fault injection of the system. The generated Structural Specification file is used for implementing the structure of user interfaces in Matlab. Data flow interaction between user interfaces and other system components are specified in generated data-flow code.

Run Control Modeling Language (RCML) is a language used to describe the behavior of the control of the physics applications. These behaviors include loading application-related software, and the starting and stopping of the applications. The implementations of these behaviors are generated as script files.

3.2. Adaptive Reconfigurable Mobile Objects for Reliability

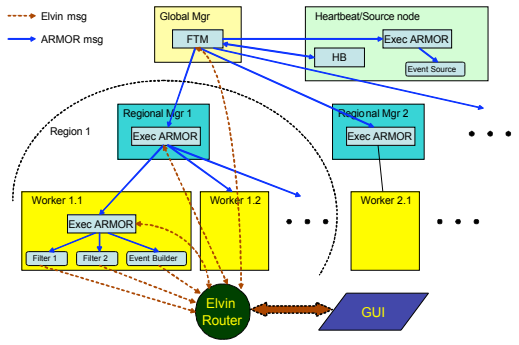
We have deployed adaptive reconfigurable mobile objects for reliability (ARMOR) middleware to provide a scalable high-availability test platform for physics applications running on the processor farm.

ARMOR infrastructure. ARMORs are multithreaded processes internally structured around objects, called elements, which provide elementary functions or services. Every ARMOR process contains a basic set of elements that provide core functionality, e.g., reliable point-to-point messaging between ARMORs and the ability to checkpoint ARMOR state. ARMOR processes communicate via message passing: the microkernel present in each distributes messages between elements within an ARMOR and between the ARMORs in a system. This modular, event-driven architecture permits developers to customize an ARMOR process's functionality and fault-tolerance services (detection and recovery) according to the application's needs. Several ARMOR processes constitute the self-checking runtime environment. Basic ARMOR types include: (i) *fault-tolerance manager* (*FTM*) initializes an ARMOR-based system configuration, maintains registration information on all ARMORs and applications, and initiates recovery from ARMOR and node failures, (ii) *heartbeat ARMOR* (*HB*) detects failures in the FTM by periodically polling for "liveness," (iii) *daemon*

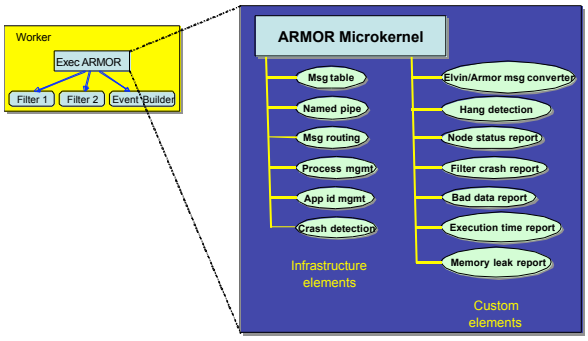
ARMOR, serves (on each node) as a gateway for ARMOR-to-ARMOR communication, and (iv) *execution ARMOR* launches and monitors application processes on a given node ([10], [11], [12], [13]).

RTES application configuration. Our initial focus has been to integrate the data dispatch and processing with the ARMOR infrastructure to provide fault-tolerant operations. As shown in Figure 2(a), there are four types of nodes in RTES: the global manager (where the FTM resides), the heartbeat/source node, the regional manager, and the worker node. Each regional manager node hosts an execution ARMOR which oversees all the worker nodes within a given region. Several applications reside on each worker node (a worker node is a dual processor machine) and an execution ARMOR, local to the worker node, monitors these applications. Figure 2(b) depicts an example configuration of the execution ARMOR. In this configuration the infrastructure elements support core ARMOR functionality, and the custom elements implement services specific to the RTES applications and enable utility functions, e.g., a node status reporting. In addition, message routing service (supported by an Elvin router), external to ARMORs, is established to support communication between the ARMOR-based infrastructure and the graphical user interface (GUI). This ARMOR-based system enables detection and automated recovery of application and ARMOR crashes, hangs, corrupted data, time requirement violations, and memory leaks.

Current efforts concentrate on the deployment of ARMORs in large networks. Multiple manager ARMORs with elements necessary to establish the ARMOR management hierarchy (with the FTM on top) are allocated, each with a distinct subset of computing nodes in the system to supervise. This approach will allow us to simplify the system monitoring and runtime fault management. An important goal is also to conduct fault/error injection based assessment of system behavior under realistic failure scenarios.



(a) ARMOR configuration in RTES



(b) Execution ARMOR configuration

Figure 2: Data dispatch and processing with ARMOR provided fault tolerance

3.3. Very Light Weight Agents

Multiple layers of very lightweight agents (VLAs) are responsible for providing the RTES/BTeV environment with a lightweight, adaptive layer of fault mitigation. The agents consist of a relatively few lines of code embedded within each node, which monitor hardware and software integrity. The VLA is both proactive and reactive. In Demo System 2004, the VLA was responsible for monitoring run times of the filter application, and alerting upstream processes of average and outlier processing times. It also tracked individual process memory and CPU utilization.

Given the number of components and intractable number of possible fault scenarios involved, it is infeasible to design an `expert system' that applies mitigative actions triggered from a centralized processing unit acting on *a priori* rules capturing every possible system state. Instead, a distributed multi-agent systems approach using self-organizing VLAs is being investigated to provide fault mitigation within the large-scale real-time RTES/BTeV environment. The latest phase of VLA development combines strategies from game theory, stigmergy, and other biologically inspired models to coordinate the actions of individual VLAs embedded within each node [14].

3.4 Physics Applications

A "user context" was needed for the RTES runtime infrastructure. The BTeV runtime environment had not yet been developed, and appropriate Level 2/3 trigger codes were not available. Consequently, an existing BTeV Level 1 muon trigger application was co-opted to serve as the Level 2 trigger. For the sake of the

Demo System, the muon trigger code served as the whole of the Level 2 computation. To provide an appropriate compute load, the trigger algorithm was repeated 1500 times on each data package, to achieve a 3.5 ms (mean) processing time, commensurate with expected Level 2 operation.

To supply data to the Level 2 trigger, and to create an event processing time distribution, a file reader ("data source") was developed. The file contained 210 GEANT generated, muon rich crossings, with typically 6 events per crossing. Since the purpose of the data source and trigger were simply to provide a non-uniform behavior for the RTES infrastructure, the small data set was sufficient to create a wide distribution of processing times.

To coordinate data handling between the event source and the trigger process(es), an event builder was developed. The event builder process maintained a queue of data packages (crossings), which it supplied to the trigger processes running on the same node.

One data source (node) provided 1 Kbyte data packages to the entire system of (variously) 12 to 64 worker nodes. Each worker node ran one event builder process, which served 2 trigger processes on that node. Each trigger process ran on its own CPU. Communication between the data source and the event builders was via TCP/IP sockets; communication between the event builders and their trigger processes was by named pipes. Mean processing time per data package was tuned to 3.5 ms; typical I/O time between packages (for a 3-worker micro-system) was a bimodal distribution with peaks at 1.5 ms and 6 ms.

3.5 Development Infrastructure

It is desirable to integrate design methods of Model Integrated Computing (MIC, [15]) (model-based design abstractions, model interpretation, and automated domain artifact generation) with an existing code management system. The RTES build system integrates RTES system models with the traditional code management system facilities to allow end-to-end automation from design to implementation.

The addition of models to the build and deploy process requires additional tasks of model interpretation to be performed before the source tree can be built. Typical executable compilers operate on source files to produce objects (for example, g++ operates on .cpp .hpp, etc. files to produce objects). Model translators can be thought of as compilers for models; in a similar fashion, model translators or interpreters also use an executable which operates on source files (models stored in .xml format) to produce objects. The objects produced from models may be any domain artifact (.cpp, .h, .rc, makefiles, .conf, etc) and may be placed in a variety of locations throughout the source tree. Figure 3 shows the propagation of artifacts from metamodel language specification to the creation of a run tree from which the final system can be deployed.

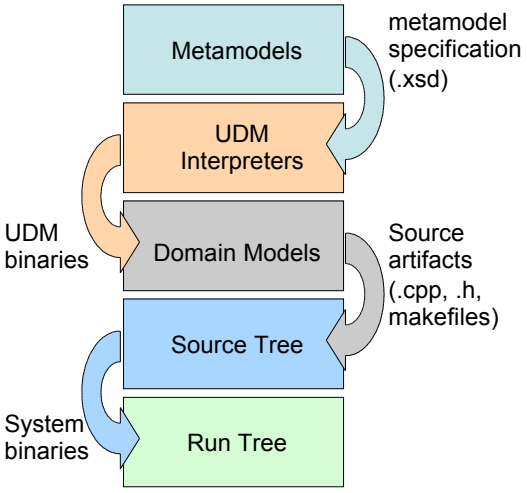


Figure 3. Propagation of model and source artifacts through the many layers of the RTES model integrated build system

Native support for command-line model interpretation on both Windows and Unix platforms is one feature of the Universal Data Model (UDM) [16] tools used by the build system. Using these tools, a set of models can be placed under version control in a traditional code management system and the process of extracting information from the models is automated within the build process.

Stages of the build system and details the actions taken at every step of the process are the following:

1) Language creation – Languages are created by interpreting metamodel paradigm specifications with the MetaGME interpreter. Each resulting language specification is placed in the build tree for use by other interpreters.
2) Domain specific language interpretation – Domain specific language interpreters are built from source code. This stage can be though of as "compiling the model compilers."
3) Domain model interpretation – For each model in the tree, the model-appropriate interpreter is identified, and executed using the model as input.
4) Coalescing of generated artifacts – Artifacts generated by the interpretation process are placed in their appropriate location the source tree.
5) Source Tree Compilation – Binaries are built for all modules in the source tree. Executables are placed into the run tree.
6) Coalescing of compiled artifacts – All artifacts generated by the compilation process are placed in their appropriate location throughout the run trees.

3.6 Control and Monitoring

Monitoring and control of a large scale system is essential to ensure its correct functioning. Graphical user interfaces provide an excellent way to visually monitor and control the system. In order to support monitoring and control of different aspects of the system at different times, the need for configurable user interfaces arises. Configurable user interfaces enable the users to dynamically view data and error conditions in ways that aid analysis as well as enable them to configure and control the state of the system.

The GUI Configuration Modeling Language (GCML) developed in GME facilitates the rapid layout and design of monitoring, control, diagnostics and fault injection user interfaces. The user interfaces use Data Type Modeling Language (DTML) to communicate with the system. DTML provides an abstraction over the Elvin publish-subscribe communication protocol used for message passing in the system. Once the models have been created in GCML, the user interface layout code as well as the dataflow code for the communication of the user interface with the system components is generated from the models.

The current run-time environment for the generated user interface is Matlab and the current run-time platform is Windows. In order to facilitate the communication of the user interface with the nodes on the Linux farm that are a part of a private network, Elvin Forwarder applications have been developed. These Forwarders are simple message repeaters written in Python that enable the exchange of messages between the nodes of the system and the user interface. This facilitates fast communication of the system with the user interfaces set up on any machine connected to the internet. While provisions have been made for multiple monitoring user interfaces to run simultaneously on different machines by running the one-way Elvin Forwarder application that forwards messages from the nodes to the monitoring GUI, only one instance of the control user interface can send control messages to the system.

4. Discussion

Two configurations were developed for presentation at the FALSE-II Workshop, held in conjunction with the 11th IEEE Real Time and Embedded Technology Application Symposium, March 7, 2005. The primary configuration employed 12 worker nodes with 4 additional nodes providing regional and global control; this configuration was the basis of the workshop demonstration. The second configuration employed 54 worker nodes, with 11 control nodes, and was an exercise in system scaling.

Several lessons were learned in the course of developing this demonstration.

4.1 Orthogonality

Efforts were made to provide a separation between the "physics code" and the RTES infrastructure, with the understanding that the physicist-authors of the trigger algorithms, etc., would at most have access to an API-level perspective of the RTES infrastructure. Similarly, the RTES developers would not be allowed to instrument physicist-authored code.

An example of this "Chinese wall" was the handling of fault initiation. Rather than sending a "run slower" message directly from the Matlab GUI a worker node (which would not be a normal command/channel), the fault instruction was routed through the data source, and embedded in the body of a data package. The trigger application, on receipt of this tainted data package, would then process more slowly, emulating a fault. This reduced unnatural command/channels in the Demo, compared to a real system implementation.

Decoupling fault management from the application also ensures low overhead during error-free operation, and simplifies future maintenance.

4.2 Scaling

The development of both "16 node" and "65 node" configurations illuminated several key characteristics of the toolset and project approach:

- debugging multiple configurations quickly identified scale-dependent and scale-independent aspects of the code. A healthy laziness, manifest as the desire to fix the "same problem" only once, promoted improved organization and automation in system generation. Model weaving tools may further improve multiple configuration support.
- scaling and remapping between configurations uncovered subtle design problems (e.g. timing and race conditions), not exposed by the testing of a single configuration. Hierarchical approaches can address scaling issues, and model analysis can identify sensitivities in advance, but exploring multiple configurations may be necessary to reveal serious system sensitivities.

4.3 Collaboration and Communication

Code was developed by 4 groups, with overlapping scopes. Well-defined APIs and development procedures, as well as collaborative communication resources (CVS, Wiki, videoconferencing), enabled members of the research team to contribute new functionalities to the system without precise knowledge of the complete runtime environment.

An excellent example of this was the development of custom ARMOR elements, to provide fault-specific behaviors. These custom elements (and the FMML graphical language for representing them) were developed by a team member who was not immediately involved in the development of the ARMOR middleware.

5. Conclusions

A prototype for reliable, fault-adaptive middleware for a commodity-based dedicated application computer farm has been developed and demonstrated. The prototype employs graphical representations capturing several dimensions of the system design, and both systemic and point solutions to fault detection and mitigation. Automated code generation simplified the support of multiple configurations. The prototype exhibits decoupling of fault management from the application, and benefited greatly from scaling, and the effective use of collaboration tools.

6. References

[1] Information on the RTES project is available from www-btev.fnal.gov/public/hep/detector/rtes/index.shtml

[2] Information on the BTeV Experiment is available from www-btev.fnal.gov/public/GeneralInformation.shtml

[3] Several SuperComputing2003 documents are available via www-btev.fnal.gov/public/hep/detector/rtes /Publications/index.html

[4] The farm is described in www-btev.fnal.gov/cgi-bin/DocDB/ShowDocument?docid=3939

[5] Elvin is a product of Mantara Software; www.mantara.com

[6] Ganglia is available from ganglia.sourceforge.net

[7] Matlab is a product of MathWorks, Inc; www.mathworks.com

[8] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, "Model-Integrated Development of Embedded Software", *Proceedings of the IEEE*, Vol. 91, Number 1, January, 2003, pp. 145-164.

[9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, P. Volgyesi, "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.

[10] Z. Kalbarczyk, R. K. Iyer, and L. Wang, "Application Fault Tolerance with Armor Middleware," *IEEE Internet Computing*, Special Issue on Recovery-Oriented Computing, March/April 2005, pp 28-37.

[11] K. Whisnant, R. Iyer, Z. Kalbarczyk, et al. "The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications," in *IEEE Transactions on Software Engineering*, 30(4), April, 2004, pp. 257-277.

[12] K. Whisnant, Z. Kalbarczyk, R. Iyer, "A System Model for Reconfigurable Software," in *IBM Systems Journal*, 42(1), 2003.

[13] Z. Kalbarczyk, et al. "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Trans. on Parallel and Distributed Systems*, 10(6), June, 1999, pp. 560-579.

[14] D. Messie, J. C. Oh,, "Polymorphic Self-* Agents for Stigmergic Fault Mitigation in Large-Scale Real-Time Embedded Systems", Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Utrecht, The Netherlands, July, 2005.

[15] J. Sztipanovits, G. Karsai, "Model-Integrated Computing", *IEEE Computer*, April, 1997, pp. 110-112.

[16] Universal Data Model tools are available from ISIS and the Escher Research Institute: escher.isis.vanderbilt.edu/tools/get_tool?UDM