Joint NCC & IIS Bull., Comp. Science, 20 (2004), 87–95

Integrated enterprise-level security solution "Vostok"

V. S. Ryzhov, L. R. Rabinovich

Abstract. This work considers a model of a multilevel information system that integrates heterogeneous classes of objects of different configuration and complexity range. The architectural solution is suggested that allows dynamical extension of a working system by connecting objects both of existing classes and new classes as well. The use of the distributed architecture significantly improves reliability of the system. The work also considers the experience of practical implementation of the model in the form of a distributed application and exploitation of the system that includes subsystems of access control, power supply, fire safety, video observation and other subsystems of life support.

1. Introduction

This work considers a model of a multilevel information system that integrates heterogeneous classes of objects of different configuration and complexity range. The architectural solution here suggested allows a working system to be dynamically extended by connecting it with objects both of existing classes and new classes as well, and the use of the distributed architecture significantly improves reliability of the system. The work analyses and summarizes the experience of practical implementation of the model in the form of a distributed application that includes subsystems of fire/access/video control, power supply control and others. The following aspects of constructing the enterprise security systems are discussed:

1. Fusion of heterogeneous subsystems into a uniform system.
2. Adaptation of a system to objects of different configuration and complexity.
3. Information system protection at all levels in order to provide enterprise security.

Integration of heterogeneous subsystems is required, since the system of enterprise security consists of various fire/access/video control subsystems and other components that include objects of different types that should operate in close interaction within a unified information space.

The system should be easily adaptable, since the objects whose vitality it should provide can be essentially different. Customization of this system is necessary, because, at first, it is desirable to unify behavior of the enterprise

security system and, at second, elaboration costs for such systems are rather high. As a consequence, it is required to support possible changes in the scenario of the system behavior during its functioning.

The aspect of reliability of the enterprise security system itself is of particular importance. Here reliability means the system protection at different levels, for example, unauthorized access and net bugging, as well as physical data damage, and the system's capability to recover in case of all sorts of damage.

2. Proposals for architecture of information system of enterprise security and life-cycle support

In order to make a correct choice of the system's architecture and methods of its implementation, the system requirements should be completely determined.

2.1. Stability

Each part of the system should be stable and mostly independent in its work from the others.

The architecture should provide solutions to the following problems:

- Fault-tolerance of hardware (fault monitoring and automatic recovery).
- Message delivery confirmation in the network interaction or guaranteed delivery of such messages (data flow buffering).
- Fault-tolerance of processing the user scripts (processing time checking and exception handling).

2.2. Scheduling of intertask communication

The problem of heterogeneous subsystems integration is solved in the following way: subsystems are functioning independently, when fulfilling a series of formulated tasks. Thus a notion of a **workflow** appears. It is necessary to implement a management system for such work flows to provide work accordance to specifications and to schedule intertask dependencies, i.e., to ensure subsystems time and data synchronization.

Except for management itself, the system should be able to restore workflows after failures inevitable in practice.

The system of control flow management consists of the scheduler and task agents. A task agent controls the work execution by the processing object; there is one agent for each task (that is, for each physical device). A scheduler is a program that processes the workflow by starting various tasks, processing various events (an event is a fact of receiving some signal from

a subsystem) and estimating the conditions related to the intertask dependences. The scheduler can submit a task for execution (to the task agent) or demand that the tasks earlier started to be stopped. (For example, if a transaction affects several databases, then tasks are subtransactions, while processing objects are local RDBMS). According to the workflow specification, the scheduler makes dependence scheduling and is responsible for all tasks being completed with valid states.

Another means of control can be added to the system: distributing control of a specification fulfillment between a scheduler and some task agent (it is called weak transaction management).

There are three architectural approaches to constructing a workflow control system:

- Centralized approach assumes that one scheduler organizes all workflows running concurrently.
- Partially distributed approach assumes that there is one (copy of a) scheduler for each workflow. If concurrent processing can be separated from the scheduling function, this approach is more natural.
- Completely distributed approach assumes that a scheduler does not exist; task agents coordinate their execution by interacting with each other in order to resolve the intertask dependencies and other requirements of work flow execution.

2.2.1. Work flow scheduling

The main purpose of the scheduler is to meet the following requirements.

- **Validity of scheduling.** The scheduling process should not break any dependence presented in the workflow specification. Besides, the scheduler is bound by constraints imposed by global control of concurrent processing, because uncontrolled interaction of tasks belonging to different workflows may lead to incorrect results. A particularly difficult task is to determine whether scheduling time dependences can be satisfied or not [1]. The scheduler should take into account that, given the time dependences, logical values of the scheduling predicates can change dynamically without any system impact. At the same time, these dependences can restrict possible actions of the scheduler (for example, by indicating that the task cannot be started before 10 a.m.).
- **Reliability.** The scheduler should guarantee that the workflow will be completed in one of the specified valid final states. Before making an attempt to process the workflow, the scheduler should address to its specification in order to check if this flow is completed in a valid

state. If the scheduler cannot guarantee that the workflow is completed this way, it should reject this specification not trying to process this workflow. Even if the workflow specification is safe, i.e. a valid final state is always reachable, there still remain some problems in workflow processing, since various deadlocks may appear. To guarantee that the workflow has no deadlocks, we can use formal methods of specification and analysis. The task of the workflow scheduler is to work out the processing strategy, which guarantees that the workflow will not be completed in an invalid state.

- **Optimal scheduling policy.** The scheduler should reach valid final states in an "optimal" way. Nevertheless, the notion of "optimal" can differ for different applications. One of its possible definitions is minimization of the processing time. Another definition uses binding of an estimate function with every task process. In this case the purpose of the scheduler would be completion of the whole workflow at minimal costs. If probabilities of successful execution are known beforehand, the scheduler can use them when choosing the execution strategy that leads to a successful global completion with maximum probability.
- **Failure handling.** The scheduler should be able to reach a valid final state even in case of failure. There are some different scenarios for failure handling. In the first one, the scheduler could ignore a failure and continue processing as if "nothing had happened", providing the so-called "forward recoverability" [3]. In the second one, the scheduler could stop processing of the whole workflow (i.e. reach one of the global states of abnormal termination). Both approaches require status information to be saved in case of failure, since, even in the second case, fixing or renewal of executing some tasks (for example, writing log files) may be required. Therefore, the scheduler should keep in a reliable storage device the whole status information, which can be used for recovering and further processing.

2.3. Workflow recovering

The main goal of work flow recovering after a failure is to provide that the operations of work flow are atomic, i.e., the results of these operations depend on the initial data only, not on some random factors caused by the failure. Recovering procedures should guarantee that, in case of a failure in any workflow-processing component (including the scheduler), the flow would reach some valid state, anyway (possibly, with the use of compensation, i.e. looking for lost data through log analysis, etc). We can assume that the task agents have their own local systems of recovering and can process their local failures by themselves. Correspondingly, the scheduler will manage mostly its own failures.

In order to recover the context of execution environment, the procedure of post-failure recovering should recover the status information at the moment of failure, including status information for each task and information about scheduling dependencies. So, the corresponding status information should be kept in a reliable storage device [2].

We should also pay attention to the contents of the request queue. If the mechanism used in the system supports the functions of stable program channels (for example, in the VMS system by DEC, queues can be implemented with the use of mailboxes — stable message queues accessible as input/output virtual devices), messages are kept in a permanent storage device and cannot be lost in case of failure. If the queues are kept in some unreliable storage device (for example, in OS UNIX environment, queues can be implemented with the use of sockets), then, instead of recovery of queues' contents, it is acceptable that schedulers resend their requests.

Many commercial products use e-mail interfaces in order to distribute tasks among process stations responsible for their execution. This simple solution has definite advantages resulting from the fact that existing e-mail systems are relatively reliable and offer a possibility of making queues and message resending.

2.4. System performance

Generation of a statistical report should take an appropriate time. The system should provide the mechanisms of making a report on a bulk of information at regular intervals (for example, once a day) and in the background mode.

The system reaction time should be comparable to the hardware response time.

2.5. System configuration

The system should provide convenient tools for configuring to be used by the system installer and maintainer. These tools should provide the possibility of configuring with minimal efforts of a large number of uniform devices and uniform business-logic scenarios.

2.6. Scalability

The architecture proposed allows us to build systems of different level – from a minimal one, implemented on one computer with external devices connected directly, to a system distributed over a large number of different computers joined into a network.

2.7. Development

The system should provide the possibility of its expansion with new devices and with new business-logic components and variants of the user interface, as well.

Physically, the system is a set of external devices (electro-mechanical devices of alarm system) connected to one or several computers joined into the Ethernet network.

From the viewpoint of software, the system is a set of components that use message exchange for interaction. The system components are functioning in the virtual operating system (virtual machine) environment, which depends on neither a specific OS, nor the computer hardware implementation.

The system is built on the blocks principle from an expandable set of components.

2.8. Virtual OS kernel

The virtual OS kernel implements the message routing and processing functions thus providing interaction between system components.

Each system component uses its system's kernel API for sending a message. In the case of a local destination address (within the same node of the virtual OS), the message is put into the process queue of the corresponding component. Otherwise the message is put into the network outgoing queue. It is assumed that the dispatch mechanism does not provide blocking of the main control flow, i.e. the message is queued, while the main process goes on. The delivery algorithm should guarantee message delivery to the addressee. The message is removed from the queue only after confirmation of its delivery to the next unit in the rout (in case of network delivery) or after it was queued for processing by a local addressee. If there is no confirmation, the system will retry to deliver the message until the delivery time limit is up.

The mechanism of message queue processing may differ. The base one is FIFO mechanism without parallel processing of messages: while a current message is in process, the others are waiting. It may be necessary for some components that other queue processing mechanisms (for example, for exceptional transactions processing) and/or message parallel processing mechanisms be implemented.

Thus, the virtual system kernel is a binding element, which combines the system components into a virtual network that supports message exchange.

2.9. System components

At the component initialization, the system registers its name (address) that is then used in message exchange. For each system component there is a specified list of messages (commands) that it can process and a list of events, which it can generate. In addition, the system component can send messages to other components.

It is assumed that there exists a hierarchy of components according to the dependence principle (for example, vista – multiplexer – com-port), which is clear for the system installer and maintainer. This hierarchy should also be reflected at the GUI (Graphic User Interface) level.

Each system component is determined by the algorithm of incoming message processing, its configuration parameters and internal state. Interaction with the system component can be divided into three categories: configuring, monitoring and control. It is assumed that, in addition to specific commands processed by a certain component, any component can execute some basic commands (for example, a request for a list of commands or configuration parameters).

2.9.1. Component configuration

For each type of components, it is supposed that there is a component that implements the configuration GUI intended for the system installer. Such a component can be considered as an application that implements the user interface and sends specific commands with configuration parameters to the target component.

The list of configuration parameters and corresponding commands (messages) is specified for each type of components.

2.9.2. Component monitoring

For each component, a list of events that it can generate is specified. Any system component can send a special command to this component and register itself as an event listener. When an event occurs, all the components concerned receive a notification message. This scheme is aimed to build the terminals and applications that process and visualize the components' state. For each type of components, there is a simplest variant of GUI for component monitoring and control available for system installer and maintainer.

2.9.3. Component control

A component may be intended for external device control or just have a controllable internal state. A list of control commands and their parameters is specified for the component. These commands are sent as a result of processing the events occurred in the system components. For each type

of components, there is a simplest variant of GUI for sending the control commands available for system installer and maintainer.

2.10. System managers

System managers are an abstract system component joining other components according to their functional models. System managers are responsible for component initialization and configuration at the system start (power-on), and for restoring and recovering of system's configuration.

The manager configuration contains a list and a configuration of the system components and a list of some parameters unified for the manager components. In a sense, the OS unit kernel is also a manager (and a component) but on the very top of the components hierarchy.

2.10.1. Manager of external devices

This manager is responsible for loading (connection) of drivers of external devices. A driver processes messages received from the system and generates the corresponding directives for a device. Responding to a change in the device state (let us call such a change "an event") the driver generates messages and sends them to all the concerned (registered) recipients.

The device driver performs the functions of device operability control, as well as initialization and recovering after hardware failure.

2.10.2. Manager of applications

API of applications is similar to that of drivers. Usually an application is registered at a certain device driver as an event listener, i.e., an application is intended to perform some business-logic on the event occurrence.

One of the possible applications is a component that allows for interpretation of scripts written by a system's user.

Another variant of an application is a component that implements GUI for interaction with the other system components.

2.10.3. Net interaction manager

This manager delivers messages to other units of the virtual OS by encrypting the IP report with common IP-routing with the use of SSL or IPSec technologies. It also receives messages from the network for a given virtual machine and put them into the process queue. In addition, the manager exerts burglar control using the firewall-like mechanism.

2.10.4. Data manager

This manager interacts with RDBMS or another information storage system. Requests for data are made through the components similar to the device

drivers that provide a proper functionality for different data types. Thus, at the data manager level, the data processing logic is separated from the method of its storage (the data base structure). Based on this manager, it is possible to create separate components that provide functionality for report generation.

2.10.5. Synchronization and transaction control manager

Synchronization and transaction control manager allows one to create and use special objects in order to synchronize the work of applications and other components. The corresponding "drivers-factories" are intended for special objects types: semaphores, mutex, locks, etc.

The manager provides a mechanism for transaction creation and control. The system components use this manager for collateral execution of a sequence of actions within one transaction.

3. Deployment results

The first release of the system here described is implemented in the form of a software system for the "Vostok" complex. The operation experience has shown that the use of the distributed architecture allows us to greatly increase fault-tolerance of the system that is one of the major requirements to the enterprise security systems.

The complex is put in operation at several enterprises in Siberia. "Vostok" was awarded twice with the Big Golden Medal of "Siberian Fair" exhibition. The second release of the system is now under development.

References

[1] Georgakopoulos D., Rusinkiewicz M., Litwin W. Chronological Scheduling of Transactions with Temporal Dependencies. — Houston, February 1991. — (Tech. Rep. / Dept. of Computer Science, University of Houston; UH-CS-91-03).

[2] Jin W., Ness L., Rusinkiewicz M., Sheth A. Concurrency Control and Recovery Multidatabase Work Flows in Telecommunication Applications // Proc. of the SIGMOD Conf. — May 1993. — P. 456–459.

[3] Reuter Reuter A. ConTracts: A Means for Extending Control Beyond Transaction Boundaries // Proc. of the 3rd Intern. Workshop on High Performance Transaction Systems. — September 1989.