IMPLEMENTATION OF THE DISTRIBUTED ALARM SYSTEM FOR THE PARTICLE ACCELERATOR FAIR USING AN ACTOR CONCURRENT PROGRAMMING MODEL AND THE CONCEPT OF AN AGENT.

D. Kumar, G. Gašperšič, M. Plesko, Cosylab, Ljubljana, Slovenia
R. Huhmann, S. Krepp, GSI, Darmstadt, Germany

Abstract

The Alarm System is a software system that enables operators to identify and locate conditions which indicate hardware and software components malfunctioning or nearby malfunctioning. The FAIR Alarm System is being constructed as a Slovenian in-kind contribution to the FAIR project. The purpose of this paper is to show how to simplify the development of a highly available distributed alarm system for the particle accelerator FAIR using a concurrent programming model based on actors and on the concept of an agent. The agents separate the distribution of the alarm status signals to the clients from the processing of the alarm signals. The logical communication between an alarm client and an agent is between an actor in the alarm client and an actor in the agent. These two remote actors exchange messages through Java MOM. The following will be addressed: the tree-like hierarchy of actors that are used for the fault tolerance communication between an agent and an alarm client; a custom message protocol used by the actors; the message system and corresponding technical implications; and details of software components that were developed using the Akka programming library.

INTRODUCTION

The FAIR Alarm System is composed of three major layers: a generation layer, a processing layer and a client layer. The connecting glue between the layers is the messaging system which allows the layers to communicate by passing messages into each other's queues and topics.

The Generation Layer

The alarm generators are the components that raise/lower alarm signals which are transported to the processing layer through a Java Message Oriented Middleware. The main purpose of the generators is to produce the alarm signals containing an alarm identification and state of the alarm that can be active or inactive. They are also responsible for handling the fast alarm oscillations. The alarm generators produce life-cycle messages notifying the processing layer about their health. The alarm generators must be registered with the processing layer before the alarm signals can be sent. This gives the processing layer a chance to prepare the environment for alarm generator monitoring and alarm signal receiving. During the registration process the processing layer also checks that the alarm identifications are known to the system. If they are not known, the processing layer creates a default configuration for the unknown alarms.

The Processing Layer

The core of the alarm system is the alarm processor which is responsible for alarm signal processing and dispatching of the processed alarm signals to the client layer via an agent. The alarm processor also monitors the alarm sources. The alarm signal processing includes: matching the alarm signal with its configuration, updating the alarm state, alarm masking, and alarm archiving. The alarm processors are stateless and session-less, working in groups to share the load of the alarm processing. Processed alarm signals are not dispatched directly to the client layer but are sent to the agents which have active client sessions. The client layer accesses the alarm system only through an agent by opening a client session. All client requests are handled by the agents. The agent is responsible for handling alarm reduction, subscribing and filtering alarm state changes, acknowledgement of the alarms, sending filtered alarm state changes to the subscribed clients, and searching the alarm state and alarm archive.

The Client Layer

There are many types of alarm clients: the alarm monitoring viewer showing the state of the alarms, the alarm archive browser displaying the alarm history from a selected time range, the alarm configuration editor which issues CRUD operations on the alarm configuration. Common to all alarm clients is that they access the alarm system through an Alarm Client API. The alarm clients open many concurrent and independent sessions through which they issue requests to the alarm system and receive replies and alarm state changes. When a session is opened in the client layer, another session is also created on the agent. A hierarchy of actors [1] is created on both layers establishing a logical communication channel between a client session actor and an agent session actor. The physical communication is done through the actor hierarchy where individual actors take different roles: session management, session supervision, service worker, JMS message producer, and JMS message consumer.

ISBN 978-3-95450-146-5

The Technology

The alarm generator is written in Java SE 7 and C++. The native version of the generator works under Linux and Windows using a Boost portable C++ source library [2] and ZeroMQ [3] for a networking and concurrency framework. The underlying messaging system is ActiveMQ [4] in a "shared file system master slave" for a fail-over configuration. A custom ActiveMQ plugin was implemented bridging ZeroMQ and JMS messages. The alarm processor and the agent are written in Java SE 7 using the Spring framework [5], Bitronix standalone transaction manager for the distributed transactions [6], and JPA for the management of relational data. No application server is needed to run the alarm system.

The alarm processors work in a cluster using Hazelcast [7], an In-Memory Data Grid. The Alarm Client API is written in Java SE 7 and uses the Akka library [8] for the actor abstraction implementation in the client and processing layer. All layers of the alarm system use custom protocol messages encoded with Google Protocol Buffers [9].

ACTORS AND AGENTS

As already mentioned, there are two subsystems in the processing layer. The core task of the alarm signal processing and generator monitoring is assigned to one or more alarm processors running in a cluster. The task of managing the client layer is done by the agents. The window into the alarm system from the perspective of a client is a session in the Alarm Client API subsystem. While the session is being established on the client layer, another linking session is also opened in the processing layer on the agent. These two sessions communicate with each other through a logical communication channel exchanging regular and life-cycle messages. If the client session detects that the agent session is not responding, it will reregister with another agent and transfer its state to it. The applications using the Alarm Client API will not notice that the session was re-established on a different agent. The application using the Alarm Client API can open many sessions. These sessions are fully independent from each other, running concurrently with their own alarm subscriptions, their own event listeners and alarm reduction rules settings.

To ease the development of concurrent and fault-tolerant alarm clients and agents, we replaced the traditional model of shared state concurrency with the Actor Model, thus avoiding the pitfalls of controlling and manipulating the shared state with locks and threads.

In the Actor Model, all objects are modelled as independent, computational entities that only respond to the messages received. There is no shared state between actors. Actors change their state only when they receive a stimulus in the form of a message [10].

Error detection is an essential component of fault tolerance. That is, if you know an error has occurred, you might be able to tolerate it by replacing the offending component, using an alternative means of computation, or raising an exception [11].

Each actor that performs a task is associated with a supervisor actor which monitors its actors for faults. If an error occurs in the supervised actor, the supervisor will initiate some error recovery procedure. This error recovery can restart or resume the subordinate actor, terminate it, or escalate the failure to its own supervisor which has the exact same options regarding the error handling. The supervisors and worker actors thus form a supervisor hierarchy. In our case, a session in the client layer is implemented as an actor that belongs to a hierarchy of supervisors and supporting actors that enable the session actor to communicate with its counterpart session actor on the processing layer. The same holds true for the session actor on the processing layer. It too belongs to a supervisor hierarchy with the supporting actors that enable communications with the client layer and service actors that execute requests on behalf of the alarm clients.

Figure 1 shows the outline of the alarm supervisor hierarchy in the client and processing layer and, more importantly, the roles that actors play to establish different types of communication channels between the client and the agent.

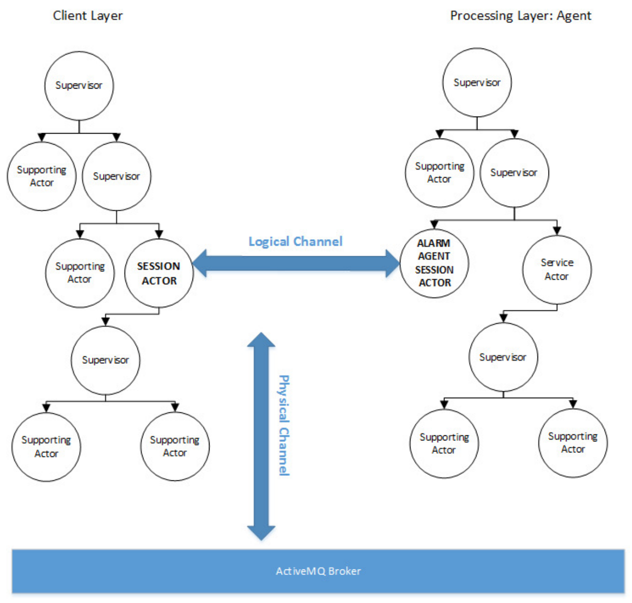

Figure 1: The Alarm Supervisor Hierarchy.

THE AGENT SUPERVISOR HIERARHY

The agent supervisor hierarchy is shown in Fig. 2 and has a root actor, AlarmAgentActor, which is responsible for bootstrapping the hierarchy. In the tree hierarchy we have three main branches of actors. The branch holding the AlarmAgentConsumerSupervisor and its supporting actors is responsible for receiving the register and unregister client protocol messages and alarm state changes from the alarm processors.

ISBN 978-3-95450-146-5

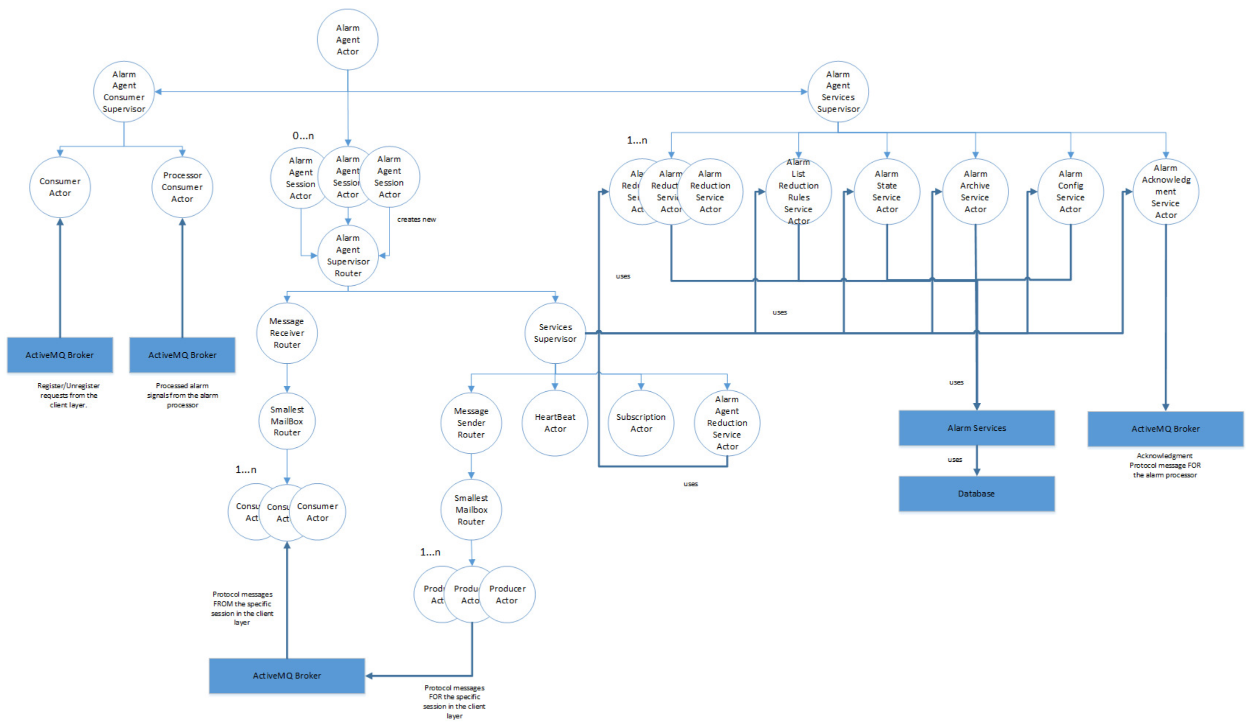
Alarm Agent Actor
Alarm Agent Consumer Supervisor
Alarm Agent Services Supervisor
Consumer Actor
Processor Consumer Actor
0...n
Alarm Agent Session Actor
Alarm Agent Session Actor
Alarm Agent Session Actor
creates new
Alarm Agent Supervisor Router
ActiveMQ Broker
Register/Unregister requests from the client layer.
ActiveMQ Broker
Processed alarm signals from the alarm processor
Message Receiver Router
Services Supervisor
Smallest MailBox Router
1...n
Consumer Actor
Message Sender Router
HeartBeat Actor
Subscription Actor
Alarm Agent Reduction Service Actor
Smallest Mailbox Router
1...n
Producer Actor
Protocol messages FROM the specific session in the client layer
ActiveMQ Broker
Protocol messages FOR the specific session in the client layer
1...n
Alarm Reduction Service Actor
Alarm List Reduction Rules Service Actor
Alarm State Service Actor
Alarm Archive Service Actor
Alarm Config Service Actor
Alarm Acknowledgment Service Actor
uses
uses
uses
uses
uses
Alarm Services
Database
ActiveMQ Broker
Acknowledgment Protocol message FOR the alarm processor

Figure 2: The Agent Supervisor Hierarchy.

The register protocol message will create a new agent session actor in the middle branch of the tree hierarchy. The newly created AlarmAgentSessionActor bootstraps its own sub-hierarchy of supporting actors. The session actor pushes all of its protocol messages, self-generated or received through the AlarmAgentConsumerSupervisor, to the AlarmAgentSupervisorRouter. This actor then routes the messages to the ServiceSupervisor where the client requests are handled by its task actors. The ConsumerActor in the session actor sub-hierarchy is responsible for receiving the client protocol messages for that session. These messages are routed to the session service supervisor by the AlarmAgentSupervisorRouter. The session service sub-hierarchy has one supervisor (ServiceSupervisor) and many task actors. The task actor HeartBeatActor is responsible for the session life-cycle management. The SubscriptionActor will filter the alarm state changes, the AlarmAgentReductionServiceActor reduces the alarms. All messages that are produced by the task actors are sent to the client layer through the ProducerActors. Lastly, we have the service layer of the agent represented by the AlarmAgentServiceSupervisor where we have all the services that are used by the session actors.

CONCLUSION

Implementing a distributed and a fault tolerant system is never an easy task. To simplify the development of a distributed and a fault tolerant alarm client layer and processing layer we avoided using the shared state concurrency model and went with the actor model. The system was made fault tolerant by organizing the actors into a supervisor hierarchy containing the actor tasks and the supervisor actors responsible for fault monitoring and error recovery [11]. The alarm supervisor hierarchies were also built and modelled in the simulation package AnyLogic 6 [12] where we used agent based modelling. The results we obtained from the simulation were used to prove that the non-functional requirements of the FAIR Alarm System were satisfied.

REFERENCES

[1] Wikipedia website: http://en.wikipedia.org/wiki/Actor_model
[2] Boost website: http://www.boost.org
[3] ZeroMQ website: http://zeromq.org
[4] ActiveMQ website: http://activemq.apache.org
[5] Spring framework website: http://projects.spring.io/spring-framework
[6] Bitronix website: http://docs.codehaus.org/display/BTM/Home
[7] Hazelcast website: http://hazelcast.org
[8] Akka website: http://akka.io
[9] Google Protocol Buffers website: https://developers.google.com/protocol-buffers
[10] M. K. Gupta, Akka Essentials, ISBN: 978-1-84951-828-4, Packt Publishing, Birmingham, UK (2012), pp. 11.
[11] J. Armstrong, "Making reliable distributed systems in the presence of software errors", Doctoral Dissertation, The Royal Institute of Technology, 2003, pp. 115-127.
[12] AnyLogic website: http://www.anylogic.com