Monitoring and Fault Tolerance for Real-Time Online Interactive Applications[*]

Vlad Nae, Radu Prodan, and Thomas Fahringer

Institute of Computer Science, University of Innsbruck,
Technikerstraße 21a, A-6020 Innsbruck, Austria
{vlad,radu,tf}@dps.uibk.ac.at

Abstract. The edutain@grid European project [1] is developing a support platform for deployment, management and execution of Real-Time Online Interactive Applications (ROIA) on Grid. In this paper we present a monitoring system we developed which collects data from all the resources in a distributed environment and from the ROIA managed by our platform. We also describe a fault tolerance service which addresses not only the faults commonly encountered in distributed systems, but also faults manifesting at service level, within the platform's management services. Finally, a use-case consisting of the platform running a massively multiplayer online game as a concrete ROIA, is presented in order to demonstrate the roles of the monitoring and fault tolerance services.

1 Introduction

The IST-034601 edutain@grid project [1] is focusing on enabling Grid support for general Real-time Online Interactive Applications (ROIA), with particular focus on online games and e-learning applications, including massively multi-user applications embracing large user communities. To achieve this goal, the project classifies ROIA as a new class of Grid applications with the following distinctive features that make them unique in comparison to traditional parameter study or scientific workflows, highly studied by previous Grid research [2]: (1) they often support a very large number of users connecting to a single application instance; (2) users sharing an application interact as a community, but they have different goals and may compete (or even try to cheat) as well as cooperate with each other; (3) users connect to applications in an ad-hoc manner, at times of their choosing, and often anonymously or with different pseudonyms; (4) the applications mediate and respond to real-time user interactions, and involve a very high level of user interactivity; (5) the applications are highly distributed and highly dynamic, able to change control and data flows to cope with changing loads and levels of user interaction; (6) the applications must deliver and maintain certain Quality of Service (QoS) parameters related to the user interactivity even in the presence of faults.

[*] This research is funded by the IST-034601 edutain@grid project.

Two of the main objectives of the edutain@grid project are unsupervised management of ROIA and load balancing of ROIA sessions by starting new servers or migrating users from overloaded servers to less loaded or newly started ones. In working to achieve these goals in distributed environments, fault tolerant services must be used and dynamic resource and ROIA-session monitoring information needs to be collected and processed. We designed, as part of the edutain@grid management layer, a monitoring service capable of collecting information about the current state and load of resources as well as low-level internal data from ROIA. We also designed and developed a fault-tolerance service which monitors the correct operation of the management services and the employed load distribution and load balancing actions taking the appropriate measures in case of faults.

We introduce the edutain@grid architecture and detail its management layer in Section 2. The monitoring service and the fault tolerance service are described in Section 3 and Section 4, respectively. Section 5 presents a use-case involving the monitoring and fault tolerance services and Section 7 concludes the paper and outlines future work.

2 Architecture

We designed a distributed service-oriented architecture depicted in Figure 1 to support transparent access[1] and scalability for an increased number of end-users (compared to current state-of-the-art) to existing ROIA. A scalable ROIA session is distributed across several ROIA server programs, called *ROIA servers* from here on, that run on distributed resources provided by multiple hosters. A real-time communication framework (RTF) [3] provides the fundamental protocols for parallelising and distributing the ROIA session across multiple ROIA servers. By distributing the load of a session on multiple resources, a larger number of end-users can be accommodated. The architecture is composed of three main actors described in the following subsections.

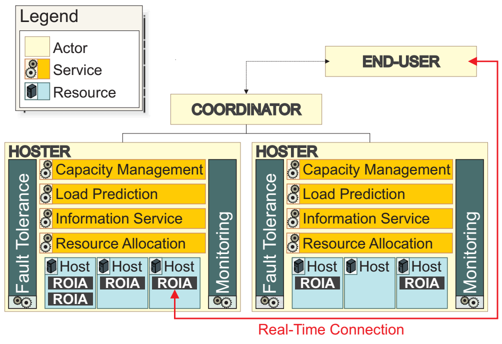

Fig. 1. The edutain@grid architecture. Detail of management services.

2.1 End-User

This actor seeks a connection to a suitable ROIA session, which provides the needed ROIA type and desired QoS. The client ROIA application negotiates

[1] The complex underlying hardware and software stacks are hidden from the end-users.

these terms with the Coordinator, transparently to the end-user. Once the appropriate ROIA session details are obtained from the Coordinator, the end-user connects to the respective ROIA server directly, connection called here "real-time connection", as depicted in Figure 1.

2.2 Coordinator

The coordinator receives from the end-user specific QoS requirements which can be performance-related (e.g. maximum latency, minimum bandwidth, minimum throughput) or ROIA-specific (e.g. ROIA type, number of participants). Its role is to distribute end-users to ROIA servers, which is accomplished as a distributed negotiation between the coordinator and hosters, each of them trying to optimise its own specific metrics expressing individual interests. The relation between the end-user and the coordinator is materialised as a client account and the coordinator-hoster negotiation is finalised as a contract. While the coordinator purely negotiates in terms of end-user-centric QoS parameters, the hosters try to optimise metrics related to their own and often contradicting interests such as maximising resource utilisation or income. Eventually an equilibrium that represents a balance between risks and rewards for all participating parties is reached. The result of the negotiation process is a performance contract that the coordinator offers to the end-user and which does not necessarily match the original QoS request. The end-user has the option to accept the contract and connect to the proposed session, or reject it.

2.3 Hoster

The hoster actor represents an organisation that provides the necessary computational and network infrastructure for running the ROIA sessions, similar to the "resource provider" from the scientific scene. The hoster also runs the management services, all within a *server container*, which monitor the provided resources, steer the hosted ROIA sessions and negotiate new connections with the coordinator.

Resource Allocation Service. Each hoster owns one resource allocation service, responsible for allocating local resources to a large number of connecting end-users. The coordinators make requests based on the load of the ROIA they operate (either statically or dynamically computed), and the hosters respond with offers based on their local time-space renting policy, their internal monitoring data collected by the monitoring service and their load prediction. The resource allocation is realised by a request-offer matchmaking mechanism based on three criteria that favour the hoster [4].

Capacity Management Service. There may occur factors during the execution of a ROIA session which affect the performance, such that the negotiated contracts are difficult or impossible to be further maintained. Typical perturbing factors include external load on unreliable Grid resources, or overloaded ROIA servers due to an unexpected concentration of end-users in certain “hot spots”. The capacity management service interacts at runtime with the monitoring service for preserving the negotiated QoS parameters for the entire duration of the ROIA session. Following an event-action paradigm, a violation of a QoS parameter triggers appropriate adaptive steering or load redistribution actions.

Load Prediction Service. The load of a ROIA session depends heavily on internal events such as the number of entities that interact altering each other’s state. Alongside internal events, there may also occur external events such as the connected end-user number fluctuation over the day or week with peak hours in the early evening [5]. Hence, it becomes crucial for a hoster to anticipate the future ROIA load.

In our load prediction service, for a fast system reaction time, we employ computationally inexpensive time series prediction models (like exponential smoothing and variants thereof) which generate predictions based on the data collected by the monitoring service. Although their predictive power is limited, they offer sufficiently high accuracy for this kind of trace data and, most importantly, have a very short prediction time which allows the capacity management service enough time to apply its ROIA session steering actions. For massively multiplayer online games, a particular subset of ROIA, we found in [6] a novel algorithm based on neural networks which offers a better accuracy than the aforementioned time series prediction models, while offering the predictions at comparable speeds.

Information Service. To store meta-information about the deployment, invocation, and execution of ROIA, we designed a generic information system with the database schema defined as a composition of independent, generic, type-specific schemas called beans, each bean consisting of one or more customised tables. The information service also stores data generated by the monitoring service and because the ROIA are very dynamic applications and generate large amounts of monitoring data in short time intervals, we optimised our information system’s performance with a special emphasis on the data storing speed on top of the MySQL database platform. We evaluated this service by carrying out a series of scalability experiments which are detailed in [7].

3 Monitoring Service

The monitoring service’s conceptual architecture and its interactions with the involved entities (i.e. monitored entities and clients) are presented in Figure 2.

There are three entities that can be monitored by this service, namely ROIA servers, ROIA sessions and hosts (i.e. the computational and networking infrastructure offered by hosters), for each one, the monitoring service including a profiled probe: ROIA probe, ROIA session probe and host probe, respectively. Each monitoring probe can be attached to a serialization probe whose function is to buffer the monitoring data generated by its attached probe and, eventually, serialize the collected data. Separate probes are constructed for each of the

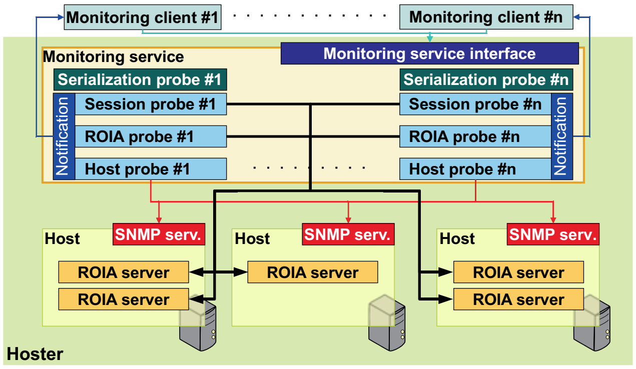

Fig. 2. The architecture of the monitoring service

monitoring service's clients, called *monitoring clients* from here on, and each one represents a separate monitoring session with specific metrics to monitor and monitoring time intervals. If the monitoring clients request notifications for their monitoring, they are registered with the notification dispatcher and each time new monitoring data is generated by their probes, they receive a notification. The monitoring clients are the other management services (all described in Section 2), the coordinator and the end-users through the client ROIA applications or through access portals. Monitoring clients have different privileges which limit the range of metrics they have access to, or the right to serialize data. The management services are granted all the monitoring privileges, whereas the end-users and portals only have access to restricted sets of public metrics.

Figure 3 presents the workflow the monitoring clients must follow in order to utilise the monitoring service:

Step 1. Choose the entity to monitor, select the needed metrics to monitor, the monitoring time interval and invoke the monitoring service with these parameters to start the monitoring process. At this point, the monitoring probes are created and started;
Step 2.[optional] Request the serialization of the monitoring data. Now, the serialization probes are created and started;
Step 3. [optional] Request notifications for new monitoring data. Here, the monitoring client is registered with the notification dispatcher;

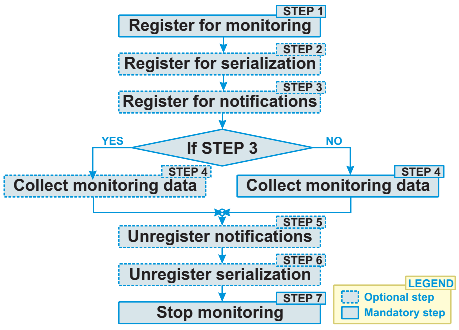

Fig. 3. The monitoring service utilisation workflow

Step 4. Collect the existing monitoring data. This step has to be performed regularly, but not necessarily at the same time interval as the monitoring is done, thanks to the buffering mechanism provided by the monitoring service;
Step 5. [optional] Stop the notifications for new monitoring data. At this point the monitoring client is unregistered from the notification dispatcher;
Step 6. [optional] Stop the serialization of the monitoring data. Here the monitoring client's serialization probes are destroyed;
Step 7. Stop the monitoring. Here, the monitoring client's probes are destroyed;

Clients can monitor sets of metrics of their choosing at time intervals greater or equal to one second. The monitoring service provides monitoring data buffers for all its probes, whose size can be adjusted prior to the service start. This facilitates the monitoring data collection process for the users because they are thus allowed to collect their data at intervals asynchronous to the monitoring interval. In addition, the monitoring service provides notification callbacks, in case the users need to collect the monitoring data synchronously with the monitoring interval.

4 Fault Tolerance Service

The fault tolerance service is designed to ensure a high level of tolerance to resource faults (to cope with the known problem of highly distributed and heterogeneous systems, like the Grid), as well as to internal faults (e.g. a subset of the management services failing).

4.1 Resource Level Fault Tolerance

In highly distributed and heterogeneous environments like the Grid, a multitude of unexpected events take place which can lead to resource failures, or the impossibility to access or use hosts. The fault tolerance service is designed to cope with faults of the following types:

A. *Host unavailability* The management services continuously check for available hosts and their status in the current setup and will only utilise the hosts which acknowledge and report a functional state;
B. *Host failure* If this event takes place while the host is being used (i.e. there are ROIA servers hosted on it) the management services will change the host's state to "unavailable" and will issue new connection details for the clients serviced by the host in question at the time of the failure. If the event takes place while the host is not involved in any ROIA sessions, then the host is simply reported as "unavailable", and will not be used until it recovers from the fault.
C. *Deployment issue* The management services do not hold exclusive rights to alter deployments. Human intervention, such as non-automated deployments and updates, is allowed which can generate faults. The fault tolerance service does not monitor the deployments (as is the case with hosts), instead they employ a *lazy fault detection* technique which is more efficient as it does not

generate overhead on the monitored hosts nor on the management services themselves. This technique involves the detection of faults in deployments at access time, when the needed ROIA files (e.g. invoking the ROIA executable) and handled accordingly, namely the deployment is marked as "unusable" and the ROIA server is started on other available hosts.

4.2 Service Level Fault Tolerance

As is the case in complex systems, the ROIA management services can be faced with unexpected events, which can lead to service faults. In order to remove or, if not possible, at least minimise the effects of such issues, all management services implement runtime diagnose methods and low-level control interfaces. This enhances the level of control over the management services at runtime, without having to restart the ROIA server container in case of faults or critical problems in any of the services. Moreover, the management services need to function without interruption since they monitor the volatile state of the ROIA sessions; a restart of the management service container is equivalent to a hoster site downtime and additionally the loss of all ongoing ROIA sessions. To this end, we have developed and implemented two monitoring, diagnose and control services designed to watch over all management services and take the necessary countermeasures to prevent and, if necessary, to handle service faults, namely *the service thread manager* and *the service internal state monitor.*

The Service Thread Manager. The management services are loosely coupled and implemented using independent threads, each thread managing a small subset of the functionality of a service. All the threads belonging to management services are automatically registered with the *service thread manager* which transparently monitors their states and activity. All threads can be in only one of the following states at any point during their lifetimes: *sleeping*, *blocked/waiting*, or *working.* The *working* state is not considered safe, thus the *service thread manager* is monitoring the actions of the threads in this state by means of checkpointing. In addition, the *service thread manager* continuously monitors the amount of work (i.e. consumed active processor time) for each thread and, at regular time intervals, assesses their sanity with one of the three defined qualifiers: *safe*, *overloaded* and *hung.* The *sleeping* and *blocked/waiting* states are not prone to faults, thus they are considered *safe* states and they do not require the service thread manager's intervention. Threads spending unusually long amounts of time in the *working* state are marked first as *overloaded,* and eventually, if the problem is not resolved in a predefined amount of time, are marked as *hung.* The *service thread manager* sends signals to all threads at regular intervals in case issues which could impede their normal functionality are detected. All the management service threads implement countermeasures for the *overloaded* and *hung* signals. Each service has its own customised mechanisms to cope with such problems, but the main actions which can be taken when *overloaded* or *hung* signals appear are:

A. For the *overloaded* signal:
   a) If the thread has a variable amount of workload, the sleeping/blocking time or distribution over time of the respective workload can be adjusted (e.g. for a monitoring serialization thread which writes data to disc in uneven chunks, a buffering solution is applied in an attempt to balance the workload between cycles);
   b) If the thread has a fixed workload each cycle, an attempt to share the load with a newly spawned thread can be made, or if by design the thread should not reach this state (e.g. light threads, monitoring threads), it will be flagged as a problematic thread and if the problem persists, its qualifier will be changed to *hung* and the corresponding signal will be sent;
B. For the *hung* signal:
   a.) If the thread receiving this signal does not keep internal data related to the ROIA sessions' states, the *service thread manager* restarts it and resets its associated internal monitoring data;
   b.) If the thread manages data related to the ROIA sessions' states, the *service thread manager* will try to first restart it attempting to reuse its full current internal state, then, on consecutive identical signals, will try restarting the thread and reusing gradually less of its present internal state (where possible) by discarding some of the internal data structures in the reverse order of their importance[2]. If this restart–*hung* cycle continues, the *service thread manager* will eventually restart the thread in question without reusing its internal state. Because this last resort measure can cause some disturbance in the normal activity of the affected service it is implemented only in threads which cannot compromise the global state of the management layer. A relevant example are the monitoring service's threads which are prone to such *hung* signals because of their reliance on network services. They can safely be restarted without preserving their internal states, in the worst case causing a loss of monitoring data on short intervals.

The Service Internal State Monitor. The *service internal state monitor*'s task is to monitor the management services' internal data. This service's purpose is twofold:

– Monitors the internal data generation and implicit memory consumption and prevents faults caused by insufficient memory by issuing *purge* commands. A *purge* command is sent to services as a signal to clear or reduce their data buffers in order to free memory for the new incoming data;
– Provides a safety measure against *memory leaks* (i.e. implementation oversights which cause data that should have been removed from memory to be accidentally kept in memory). The service observes the aberrant memory usage in the faulty threads and orders a cleanup of their internal data. This measure can be destructive (i.e. the service in question could lose important data during the forced cleanup), but it also has the potential to prevent more serious faults which could lead to the service or even service container shutdown.

[2] The importance of a data structure is directly proportional to the likelihood that a fatal failure can occur upon its corruption.

5 Massively Multiplayer Online Game Use Case

We present a use case for the monitoring and fault tolerance services, involving a *massively multiplayer online game* (MMOG), a popular ROIA type, which demonstrates how our ROIA management services cope with a resource-level fault. MMOG are based on a client/server architecture, in which the game server simulates a world via computing and database operations and receives and processes commands from the clients. Based on the actions submitted by the players, the game servers compute the global state of the game world represented by the position and interactions of the entities, and send appropriate real-time responses to the players containing the new relevant state information. The more populated the game world is and the more interactions between entities exist, the higher the load of the underlying game server will be.

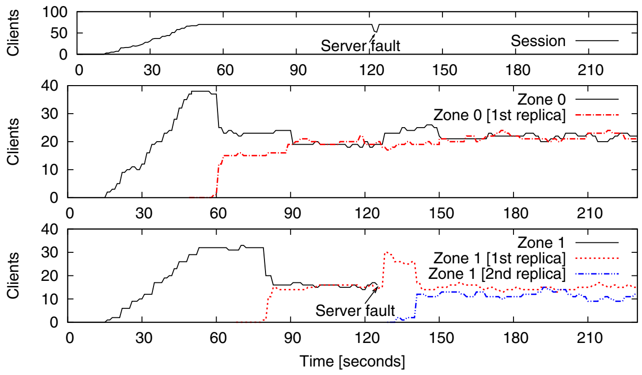

Fig. 4. Monitoring session traces. The clients connected to a game session and their distribution on the game servers within a game session.

We use a MMOG, which supports two methods of load distribution. One, called *zoning*, is based on the spatial partitioning of the game world in zones to be handled independently by separate machines. Clients can freely move between zones by means of transfer portals. The other, called *replication*, distributes load by replicating the same game world (or zone) on several machines [8]. Clients may be transparently migrated between replicated game servers, this representing the actual load sharing mechanism behind this load distribution method. These methods are dynamically employed by our ROIA management services described in Section 2.

Figure 4 shows the total number of clients connected to the distributed MMOG session (top side) and their distribution on the game servers (middle and bottom). The data here shown is collected using the serialization probes of the monitoring service. The session consists of two zones ("zone 0" and "zone 1"), initially running on two machines. At second 15 we initiate a wave of 70 clients connecting to the session, distributed among the two zones. When the initial game servers are getting close to an overload, the management services start replication game servers; for zone 0 at second 60 and for zone 1 at second 78. We simulate a failure of the host running zone 1 at second 125 which results in the disconnection of all the clients connected to it. The fault tolerance service detects the host failure and treats it as described in Section 4.1, by immediately reconnecting the clients to another game server hosting zone 1 (*zone* 1 [1st *replica*]) in order to hide the fault for the involved clients. In a few seconds, because of the additional load generated by this new wave of clients on the *zone* 1 [1st *replica*] server, the capacity management service starts a new replication process and migrates a subset of the zone 1 clients onto it.

6 Related Work

In the area of distributed systems monitoring a significant amount of work has been carried out [9] [10]. While the existing monitoring solutions address distributed systems, some of them having a strong emphasis on performance, they either only support machine and infrastructure level monitoring [9], or only application instrumentation [10]. In contrast, our approach combines the machine level monitoring using the SNMP protocol [11] with the application instrumentation, ensured by the RTF [3], into a single unified system.

Regarding fault tolerance, while comprehensive platforms for fault diagnosis and recovery exist [12] [13] [14], they mostly focus on the correctness of the services' output and root cause of the failures rather than on their overall availability. Our approach lacks the capability to detect the cause of failure in real-time, but, in turn, it has a light design which ensures a fast reaction time in case of failures, thus guaranteeing an almost continuous availability of the managed services which is of utmost importance in ROIA management.

7 Conclusions

We focus our research towards the development of a service oriented ROIA management platform. Here, we presented the architecture and the component services of such a platform. A monitoring system which collects data from resources in the distributed environment as well as from the managed ROIA was described. We also detailed the functionality of our fault tolerance service which addresses not only the resource and deployment related faults, commonly encountered in distributed systems, but also faults manifesting at service level, within the platform's other management services. Finally, a use-case consisting of the platform running a massively multiplayer online game as a concrete ROIA, was presented in order to demonstrate the roles of the monitoring and fault tolerance services.

References

1. Fahringer, T., et al.: The edutain@grid project. In: Veit, D.J., Altmann, J. (eds.) GECON 2007. LNCS, vol. 4685, pp. 182–187. Springer, Heidelberg (2007)
2. Taylor, I., Deelman, E., Gannon, D., Shields, M. (eds.): Workflows for e-Science: Scientific Workflows for Grids, p. 530. Springer, Heidelberg (2007)

3. Glinka, F., Ploß, A., Müller-lden, J., Gorlatch, S.: Rtf: a real-time framework for developing scalable multiplayer online games. In: NetGames '07, pp. 81–86. ACM, New York (2007)
4. Nae, V., Iosup, A., Podliping, S., Prodan, R., Epema, D., Fahringer, T.: Efficient management of data center resources for massively multiplayer online games. In: Proceedings of the ACM/IEEE conference on Supercomputing (2008)
5. Feng, W.c., Brandt, D., Saha, D.: A long-term study of a popular mmorpg. In: NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games, pp. 19–24. ACM, New York (2007)
6. Nae, V., Prodan, R., Fahringer, T: Neural network-based load prediction for highly dynamic distributed online games. In: Proceedings of 14th International Euro-Par Conference, pp. 202–211 (2008) ISBN 978-3-540-85450-0
7. Nae, V., Herbert, J., Prodan, R., Fahringer, T.: An information system for real-time online interactive applications. In: Euro-Par 2008 Workshops, pp. 352–361. Springer, Heidelberg (2009)
8. Müller, J., Gorlatch, S.: GSM: a game scalability model for multiplayer real-time games. In: Infocom. IEEE Computer Society Press, Los Alamitos (2005)
9. Newman, H.B., Legrand, I., Galvez, P., Voicu, R., Cirstoiu, C.: Monalisa: A distributed monitoring service architecture. CoRR cs.DC/0306096 (2003)
10. Gunters, D., et al.: Dynamic monitoring of high-performance distributed applications. High-Performance Distributed Computing 0, 163 (2002)
11. Case, J., Fedor, M., Schoffstall, M., Davin, J.: A simple network management protocol (snmp). rfc 1157. Technical report, Network Working Group (1990)
12. Abd-El-Malek, et al: Fault-scalable byzantine fault-tolerant services. In: SOSP '05, pp. 59–74. ACM, New York (2005)
13. Hofer, J., Fahringer, T.: Grid application fault diagnosis using wrapper services and machine learning. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 233–244. Springer, Heidelberg (2007)
14. Dialani, V., et al.: Transparent fault tolerance for web services based architectures book. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 107–201. Springer, Heidelberg (2002)