Effective Analysis of Runtime Failures in Group Communication Systems

Alex Krits
IBM Haifa Research Lab
krits@il.ibm.com

Benny Mandler
IBM Haifa Research Lab
mandler@il.ibm.com

Yoel Krasny
New York University
jkrasny@stern.nyu.edu

Gabriel Kliot
Israel Institute of Technology
gabik@cs.technion.ac.il

Oren Rubin
IBM Haifa Research Lab
orubin@il.ibm.com

Roman Vitenberg
IBM Haifa Research Lab
romanv@il.ibm.com

Abstract

We describe the methodology employed to overcome the challenges we encounter while supporting world-wide installations of Distribution and Consistency Services (DCS), the distributed group communication component of WebSphere (WAS). Our solution relies on a log analyzer that performs cross-log verification for a set of traces collected from the appropriate processes. Trace generation can be dynamically and selectively enabled on the client premises. The comprehensive analysis relies on a carefully devised set of invariants that need to be preserved at every point in time. The analysis determines if a problem has occurred, and helps to pin-point the root cause of real problems. A separate mechanism is combined in order to provide runtime monitoring, health checking, as well as detecting and analyzing deadlocks.

The main contribution of this paper is in presenting the fruitful combination of the automated runtime monitoring with the post-mortem distributed log analyzer that together detect a significant portion of problems while minimizing manual involvement and facilitating root cause analysis.

1. Introduction

DCS is a view-oriented group communication component [1], which lays the foundation for highly available and fault-tolerant systems. It is a distributed, complex, scalable, and non-deterministic system, thus challenging to test and support while deployed in the field. Moreover, most failures of this component are catastrophic for the client, thus they demand fast resolution of encountered problems. An additional complication is that customers are often not even aware that a problem has occurred in the system. The complexity of the system further stems from the combination of two programming paradigms: asynchronous message-based communication and concurrent multithreaded programming (For more detailed information see [2]).

Our solution is composed of two components: exploiting DCS layered architecture we seamlessly insert a new layer which monitors the system at runtime, complemented by a distributed log analyzer, which forms a unified global picture of assorted distributed logs.

Enabling the system is a carefully composed set of invariants, some local and some distributed, that need to be preserved at every execution point. The invariants may be verified by the runtime monitor or by the log analyzer. Invariants examples:

- All view members agree on the same view members, id, and leader. Verification of this property provides an indication whether the occurred chain of view changes is legal or not.
- Messages were delivered according to requested guarantees.

2. Runtime Monitoring

In order to implement runtime monitoring, we chose to add a layer into DCS just below the application layer. This layer observes all messages and events through the stack and verifies the correctness of DCS operation. This layer can be made a part of the running DCS stack based on need.

2.1. Deadlocks detection

Deadlocks detection and elimination is not a trivial task in the multi-component environment of DCS. We utilize JDKs ability to detect deadlocks during core dumps, which can be triggered by sending a signal to the JVM when DCS suspects that a deadlock has occurred.

DCS suspects a deadlock by using a pair of timestamps and a separate thread that runs in parallel with the standard system threads. One timestamp is updated, under the main DCS lock, by the DCS periodic self-checking mechanism. A second timestamp is periodically updated by a separate thread, without taking any lock. If the second timestamp is not updated we suspect that DCS is suffering from CPU starvation. If, on the other hand, the second timestamp is updated but the first is not we suspect a deadlock involving the main lock, and thus initiate a core dump.

The biggest advantage of this mechanism is that it eliminates the need to recreate deadlocks. Once a problem occurs, all the information needed for the developer to solve the problem is logged.

3. Distributed log analyzer

The log analyzer performs cross-log verification of events that occur at different processes by utilizing both the real and logical time in which events took place. The analysis system verifies distributed invariants by inspecting logs sent by the customer. This post-mortem log analysis is done partly based on the information that the runtime monitoring layer recorded in the log files.

The log analyzer extracts all relevant log entries, reconstructs a global execution path of the system, builds helper data structures, and invokes various post-mortem checkers to verify that the distributed invariants were preserved.

By extracting only small amount of relevant information from all log files and checking the abidance by the invariants, the log analyzer solves the problem of large amounts of available distributed information that is hard to interpret. Automatic analysis of the gathered information relieves developers from sifting through and interpreting and manually analyzing the traces. When the post-mortem checking system finds a violation of the invariants, it supplies the developer with sufficient information to facilitate root cause analysis.

The main challenge of reconstructing consistent global history is to correlate multiple log events from different logs. The log analyzer handles the correlation problem by combining logical and real clocks [3]. The analysis consists of two phases: in the first phase, the analyzer correlates only events in different logs that are tagged with both logical and real time. The analyzer makes a single simultaneous scan over the logs of all processes in the execution and attempts to merge them. Once a correlation for a single global event is found, the analyzer proceeds to search correspondence for the next global event. More specifically, correspondence is established based on logical timestamps as long as the real time difference does not exceed the maximal assumed clock skew. In the rare case that there exist several different views with the same view identifier within the clock skew, we employ additional sanity checks for resolving the conflict.

Based on the gathered information, the log analyzer builds a view graph, which represents a global evolution of views in the system. Every node in this graph corresponds to a view whereas an edge between two views signifies that there is a process that established these two views in succession.

Once a correspondence for view-related events is established, the analyzer starts the second phase and attempts to verify the correctness of all other events. This is done by separately considering each group of events that are enclosed by the preceding and succeeding view-related events.

3.1. Additional Post-mortem checks

Once the view graph is built, various post-mortem checks can be executed on it. For example:

- Each process that is reported as a view member by other processes has a log record that corresponds to this view
- Messages delivery guarantees

When such a post-mortem check detects a problem, the developer will be supplied with all relevant information available in the logs: description of the problematic event, view number and names of the members involved, exact time of its occurrence at each participating member, lines in the log file from which the relevant log entries were extracted, etc. Afterwards, this information will be efficiently used in root cause analysis.

An important attribute of the post-mortem analyzer is that it is not tightly coupled with the specific DCS implementation and events. In fact, it may be used by any group communication system that logs sufficient view membership information and provides an appropriate log parser. The post-mortem analyzer already supports additional types of logs, such as the standard WAS logs, which include only a small part of DCS log events. This allows WAS system administrators to use it as an external tool at a client site.

4. Future Work

There are several outstanding issues which we encounter often in customer engagements for which a solution hasn't yet been devised.

- A prominent experience with customers is that configuring these systems is not trivial and is error prone. To this end a configuration verification tool was written, and will not be covered specifically here
- Customers are often not even aware that a problem has occurred. This can lead to a state in which the system is malfunctioning for a considerable amount of time, not providing the high availability features for which it was planned.
- Customers often do not realize the nature of the problems, and thus do not know on which processes to turn on verbose tracing, and which trace files they should submit for inspection.

References

[1] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[2] E. Farchi, G. Kliot, Y. Krasny, A. Krits, and R. Vitenberg. Effective testing and debugging techniques for a group communication system. In *Dependable Systems and Networks (DSN)*, pages 80–85, 2005.

[3] L. Lamport. Time, clocks and the ordering of event in a distributed system. *CACM*, 21(7):558–565, 1978.