Supporting Resource Bounded Multitenancy in Akka

Ahmed Abdel Moamen and Nadeem Jamali

Agents Laboratory
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada
ahmed.moamen@usask.ca, jamali@cs.usask.ca

Abstract

There are several advantages of multitenancy: the serving of multiple tenants, each with its own privileges, from the same instance of a software system. Although the naming convention in actor systems – actor names cannot be guessed – naturally supports multitenancy, there is no explicit way of managing the resource competition between tenants. There are models for coordinating resource use in actor systems; however, they are difficult to implement for efficient implementations of Actors. This paper presents our efforts in implementing resource coordination support for actor systems implemented using the Akka library.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming

Keywords Resource Coordination, Actors, Akka, Multitenancy

1. Introduction

Service providers with business clients often have to launch separate instances of their service for each client. This is expensive in terms of resource use, often requiring dedicated hardware to support the runtime system for each client. An alternative to this is multi-tenancy, where the same instance of the service can serve the needs of multiple clients simultaneously. A requirement for such a system is the ability to manage the resource competition between the tenants. One way to support the functional needs of multitenancy is by implementing systems using Actors [1]. Actors are autonomous concurrently executing active objects, each of which encapsulates a thread of control and behavior. The model mandates globally unique names for actors, and these names cannot be guessed, making it possible for multiple tenants to coexist in the same namespace without interfering with each other. Managing the resource competition between tenants requires additional support for resource coordination, as provided by the CyberOrgs model [3].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2989219

There is a growing number of implementations of Actors, including production languages such as Scala,[1] which supports actors through its Akka library.[2] Although resource coordination has been implemented for efficient prototype implementations of Actors [4], none of the production languages currently support it. The specific mechanisms making up these solutions do not easily transfer between languages because of significant differences in their message-handling and scheduling infrastructures. In other words, the opportunities afforded for the required fine-grained scheduling are very specific to each language.

This paper presents our prototype implementation of such support as an extension[3] for Scala/Akka.

2. Design and Implementation

To support multi-tenancy in an actor system, we need to control the amount of resource used by each tenant's actor computation in the system. The most direct way to do so is by controlling the scheduling of the actors' threads [2]. However, efficient production languages such as Scala/Akka do not use one thread per actor; it is orders of magnitude more efficient to have a pool of threads, where each thread executes multiple actors. One possibility could be to have all actors of each tenant be executed by a thread, and then schedule the threads as required. The performance would then depend on the number of tenants hosted in the system.

A different approach to controlling resources for actors is described in [4], where opportunities for control are found by manipulating the order in which actor messages are delivered for processing. Although the level of control afforded by this approach is not as precise as what could be possible in a one-thread-per-actor implementation, it offers sufficient control for important classes of applications [4]. In this paper, we apply a similar approach to actor systems implemented in Akka. The main challenge we faced was the very different messaging and scheduling infrastructure in Akka's runtime system, requiring new algorithms to be developed.

The only computational resource we consider is CPU time, counted in 1-millisecond *ticks*. Ticks are consumed by a tenant's actors to executes computations triggered by the arrival of messages. Allocations are made to tenants within

[1] The Scala Programming Language. http://www.scala-lang.org

[2] Akka Actor Library. http://www.akka.io

[3] Available online: https://github.com/ama883/Akka-Resource-Control-Lib

recurring *time intervals*. If a tick available to a tenant in an interval is not consumed, it expires.

A system administrator specifies the number of ticks per interval allocated to each tenant. To control the ticks consumed by a tenant in an interval, the run-time system controls the flow of messages to the tenant's actors in that interval. Particularly, it relies on an estimate of the number of ticks required for processing the message, provided by the programmer. We enable this by defining a new type of actor messages which encapsulates expected execution time for a message along with the Akka message.

In Akka, a message dispatcher is considered the *core engine* for the runtime system because it controls the processor cycles given to actors, as well as their throughput, which is the number of messages delivered to them at a time. The dispatcher has access to the global message queue, actors' mailboxes, and the pool of threads which execute the actors. When a message is sent to an actor, the dispatcher first places it in the global message queue. When it is that actor's turn to be executed, the dispatcher moves the right number of messages for that actor from the global queue to the actor's mailbox (as specified in the throughput), and finally, grabs an idle thread from the thread pool, and tells it to execute the actor for those messages.

For our extension, we control the number of messages delivered to actors' mailboxes using mechanisms which work within the constraints of Akka's message dispatcher. In particular, we add two new components to Akka's infrastructure. We add a *reasoning component* into the message dispatcher to either deliver or postpone the delivery of messages for an actor according to the number of ticks remaining in its tenant's allocation for the interval. We also add a *monitoring component* in an actor's mailbox to observe the number of ticks consumed by it. Additionally, we added book-keeping to keep track of tenants, their actors, and their resource allocations. To allow a tenant to maximally utilize its allocated ticks for an interval, we replaced Akka's default FIFO actor mailbox queues with priority queues, queueing messages with smaller execution times ahead of those with larger execution times. Although this changes the messages' order or delivery, Actor semantics [1] explicitly allow it.

Figure 1 illustrates how the extension modifies the life cycle of a message. Particularly, note the monitor in the actor mailbox, and the reasoner just below that; the rest of the figure essentially shows Akka's default message dispatching. Once a thread has been given messages for an actor to execute, for each message (beginning with the top message in the priority queue), the reasoner examines the ticks required for executing it, compares it with the ticks remaining in the actor's tenant's allocation for the interval, and executes the actor for the message only if permissible. The monitor reports back the actual number of ticks consumed in processing the message, which is then deducted from the tenant's allocation for the interval. If the ticks required for the message exceed the tenant's remaining allocations for the interval, the thread is returned to the pool, and the actor mailbox (representing the actor's future computations) is placed on a queue of mailboxes waiting for the next interval.

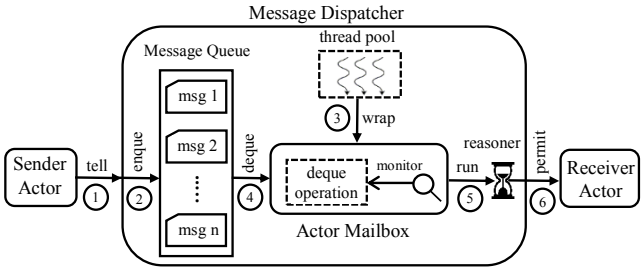

Figure 1: Modified Life Cycle of Message Dispatching

3. Conclusions

This paper presents a mechanism that supports multi-tenant actor systems. This mechanism provides a set of multi-tenancy capabilities including a provision of differential quantities and qualities of service to clients. We prototype such a mechanism as an extension for Akka.

One limitation of this approach is that instead of measuring the actual execution time of the computations triggered by messages, the system relies entirely on the information provided by the programmer, which could be estimated analytically or experimentally. We also implemented an alternative solution where the system measured the actual time taken by actors to process messages; however, because Akka does not support the suspension of an already executing actor, the entire computation is left vulnerable to a single actor taking excessive resources.

In on-going work, we are also experimentally evaluating the scalability, performance, and effectiveness of control offered by the approaches.

Acknowledgments

Support from Natural Science and Engineering Research Council of Canada (NSERC) for collaborative work with Updex Solutions Inc. is gratefully acknowledged. Al Marashi and Sina Ghaffari at Updex, and Dezhong Wang at the Agents Lab, offered valuable feedback on this research.

References

[1] G. Agha. Actors: A model of concurrent computation in distributed systems. *MIT Press, Cambridge, MA, USA*, 1986.

[2] N. Jamali and X. Zhao. A scalable approach to multi-agent resource acquisition and control. In *Proceedings of Autonomous Agents and Multiagent Systems*, pages 868–875, 2005.

[3] N. Jamali and X. Zhao. Hierarchical resource usage coordination for large-scale multi-agent systems. In *Massively Multi-Agent Systems I*, LNCS, pages 40–54. Springer, 2005.

[4] X. Zhao and N. Jamali. Supporting deadline constrained distributed computations on grids. In *Proceedings of Grid Computing*, pages 165–172, 2011.