Programmable Elasticity for Actor-based Cloud Applications

Bo Sang  Purdue University
Srivatsan Ravi  University of Southern California
Gustavo Petri  IRIF – Université Paris Diderot
Mahsa Najafzadeh  Purdue University
Masoud Saeida Ardekani  Samsung Research America
Patrick Eugster  Purdue University, TU Darmstadt

Abstract
The actor model is a popular paradigm for programming scalable cloud applications. Building elastic and scalable cloud applications requires application developers to carefully adjust the application scale (the required resources) and the placement of actors at runtime. Unfortunately, there is no efficient solution which could manage application elasticity automatically during runtime without disrupting ongoing requests. This paper proposes the idea of programmable elasticity approach, which allows application developers to define a set of elasticity rules for different actors. The runtime service endeavors to apply the elasticity rules while relieving the application programmer from dealing with the management of distributed state and efficient utilization of cloud resources.

CCS Concepts  • Computing methodologies → Distributed programming languages;

Keywords  Cloud Application, Elasticity, Actor Model, Programming Model

1 Introduction
The actor model has gained popularity for building complex cloud-based distributed applications. Some latest actor-based programming language (e.g., AEON and Orleans) can help programmers implement cloud applications in a more efficient and simple way. Actors are used as service endpoints that process requests or tasks, and their placement can directly affect the application performance. Thus, it is important to decide how to distribute and place actors across different servers to optimize performance and resource efficiency. Due to changes in workloads, the static placement of actors would not necessarily lead to the best performance and resource efficiency. For example, consider an online messaging application where users, modeled as actors, who join the same chat room can talk to each other. To reduce the communication cost, the actors participating in the same chat room should be put on the same server. However, one or many chat rooms may become big over time, such that one single server cannot handle the whole traffic. Moreover, users may actively join and/or leave different chat rooms, so it is not straightforward to choose one server for placing the user’s actor. Thus, actor-based applications require an efficient elasticity management service in order to react to all workload changes by automatically 1) re-distributing actors among available servers, and 2) provisioning and de-provisioning resources while ensuring Service Level Agreements (SLAs) and preserving the application semantics.

Newell et al. recently showed that more than 90% of communications among actors are remote for certain applications. Thus, taking into account relationship between actors and also their workload can significantly increase the actual potential of platforms towards actor-based programming models for building low-latency and scalable applications. Moreover, and as we will argue in this paper, because resource usage and interaction of actors are directly related to application logic, without any application-level information, elasticity management services may fail to make efficient decisions. Therefore, it is crucial to have a fine-grained elasticity management service.

This paper proposes a model for programmable elasticity: it requires the minimal input from the programmers (e.g., specifying upper bounds on memory/CPU utilization, specifying which actors communicate frequently), while still providing efficient and fine-grained scale-adjustment without affecting the semantics of the distributed application or its availability. Generally speaking, this programming model raises the level of abstraction for elastic cloud programming by allowing developers to build efficient cloud applications with minimal programming efforts.
Heartbeat Messages

Message

Session

Session

Session

Session

Session

Observer

Router

Actor Reference

[0x0]V16
[0x0]14
[0x0]V4
[0x0]12
[0x0]14
[0x0]2
[0x0]V3
[0x0]5
[0x0]4
[0x0]V15
[0x0]V12
[0x0]15
[0x0]16
[0x0]V13
[0x0]V14
[0x0]13
[0x0]11
[0x0]V2
[0x0]V11

Figure 1. Halo 4 Presence Service \[4\].

Router actors execute CPU intensive tasks and session actors will make call on corresponding player actors frequently.

Roadmap. The remainder of this paper is structured as follows. §2 motivates our programmable elasticity model using examples of two popular cloud applications. §3 details our language support for programmable elasticity while §4 presents the details of our elasticity management service that implements the runtime system support for migrating actors. §5 presents preliminary evaluation for a cloud game engine, §6 overviews related work and §7 concludes the paper.

2 Motivation

In this section, we motivate the need for programmable elasticity through two examples: the presence service of a large scale elastic game (Microsoft Halo), and a large scale elastic data structure (B+ tree).

Halo 4 Presence Service. The presence service of Microsoft’s Halo 4 is responsible for tracking active game sessions, participating players, and their game status. This service is developed using a distributed actor model called Orleans \[4, 7\]. Orleans is an actor-based programming model extended from C#. Its runtime system keeps monitoring communications among actors, and collocate actors with frequent interactions on the same server.

As shown in Figure 4, the presence service has four types of actors: (1) Router actors receive heartbeats from Xbox consoles. These actors are responsible for extracting session IDs from the heartbeats, and forwarding messages to the right session actors. (2) A session actor represents a game session. It receives messages from router actors, updates its status, and calls player actors that are in the game session. (3) A player actor represents a game player. (4) An observer actor receives real-time notifications form session actors, and manages session actors if some errors happen.

Each of the above four actor types have different placement requirements during runtime. Since router actors are CPU demanding, a programmer may want to balance them out on various servers. Session actors, on the other hand, frequently communicate with their corresponding player actors. Therefore, a programmer may decide to colocate a session actor along with its corresponding player actors on a single server in order to reduce latency by eliminating remote messages. Moreover, since a player may join different sessions, the runtime system may need to continuously monitor them, and move a player actor to a server that is hosting its corresponding session actor. Consequently, these elasticity decisions are quite application-specific, and a program-agnostic approach can not easily capture execution logic of the application.

B+ tree. B+ tree is a popular data structure used for fast indexing. Figure 2 shows a typical implementation of distributed B+ tree. Since the keys of B+ tree are associated with the data in the storage, the size of B+ tree varies as workload changes i.e., adding or deleting keys. In order to handle large amount of data, B+ tree can be implemented as an elastic distributed application using actor-based frameworks by considering each B+ tree node as an actor.

The runtime system needs to decide how to automatically partition these actors among servers. One viable approach is to colocate nodes with frequent interactions on the same server, which is the strategy of Orleans. However, solely relying on communications among actors may lead to poor decisions. In a B+ tree, the size of leaf nodes is much larger than inner nodes, and have much higher migration overhead. The runtime system likely makes efforts to place leaf nodes with their parent inner nodes on the same server. Therefore, one server may not be able to hold all those leaf actors. In addition, deleting and adding keys (e.g., deleting key 5 in Figure 2) may result in B+ tree structure adjustment. Such adjustments can lead to frequent migration of leaf nodes.

The above examples show that the runtime system needs more application-specific information in order to have efficient elastic deployments. In this paper, we argue that programmers implementing elastic applications can augment their applications with little effort, and define fine-grained elasticity policies while the underlying runtime system endeavors to arbitrate among possible elasticity policies.
will choose to colocate actors that interact often in the same
server, hence reducing the overhead due to network latencies.

Two factors influence the choices of actor placement. Firstly,
the level of individual actor instances, we consider that a reason-
able compromise between programmability and fine-grained
control can be achieved by providing elasticity decisions at
the level of actor types. Hence, all actors of the same type
will follow the same policies, and the fine-grained tuning
between different instances of the same type is informed by
the statistics which are collected on a per-actor level.

Two factors influence the choices of actor placement. Firstly,
we consider the amount of resources necessary for an actor.
To this end, we allow programmers to inform the runtime
system of their expected CPU and memory utilization. The
second factor is the communication patterns, which greatly
impact the performance of the application depending on the
locality of the actors. In general terms, the runtime system
will choose to colocate actors that interact often in the same
server, hence reducing the overhead due to network latencies.

3.1 Elasticity Model

For actor systems, elasticity decisions boil down to how to
place actors between servers for the better adjustment to the
workload. While there are many variables that could affect
the performance of an actor, we restrict the runtime system
to measure the average cpu/memory/network usage, along with
average number of requests received within a period of time,
and the interactions between two particular actors.

While the finest elasticity decisions can only be made at the
level of individual actor instances, we consider that a reason-
able compromise between programmability and fine-grained
control can be achieved by providing elasticity decisions at
the level of actor types. Hence, all actors of the same type
will follow the same policies, and the fine-grained tuning
between different instances of the same type is informed by
the statistics which are collected on a per-actor level.

Two factors influence the choices of actor placement. Firstly,
we consider the amount of resources necessary for an actor.
To this end, we allow programmers to inform the runtime
system of their expected CPU and memory utilization. The
second factor is the communication patterns, which greatly
impact the performance of the application depending on the
locality of the actors. In general terms, the runtime system
will choose to colocate actors that interact often in the same
server, hence reducing the overhead due to network latencies.

3.2 Configuration Language

To express elasticity policies, our programming model is
integrated with an actor programming language [9]. We will
not discuss the details of the language, but we will assume that
actors are equipped with fields, and provide remote functions
implemented by means of message passing. In fact, elasticity

Table 1.

<table>
<thead>
<tr>
<th>(a) Primitive Resources</th>
<th>(b) Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( at, any )</td>
<td>colocate(( at_1, at_2, cd ))</td>
</tr>
<tr>
<td>( f\cdot count )</td>
<td>separate(( at_1, at_2, cd ))</td>
</tr>
<tr>
<td>( ref(f\cdot d) )</td>
<td>pin(( at ))</td>
</tr>
<tr>
<td>( f\cdot d )</td>
<td>isolate(( at, cd, re ))</td>
</tr>
<tr>
<td>( cpu )</td>
<td></td>
</tr>
<tr>
<td>( mem )</td>
<td></td>
</tr>
<tr>
<td>( net )</td>
<td></td>
</tr>
</tbody>
</table>

any indicates any actor type, \( cd \) presents the conditions to trigger
elasticity behaviors.

There are a number of primitive resources that the program-
mer can use to guide the policies as listed in Table[1]. When
describing the policy of a certain actor type, it is sometimes
useful to know how many times a certain function has been
called in the last epoch, i.e., a configurable period of time.
To this end, we provide \( f\cdot count \). One could be interested in
affecting all instances of a certain actor type \( at \) in a similar
fashion. That is the purpose of the syntax \( ref(at) \), which repre-
sents the set of all instances of type \( at \) pointed to by the
actor on which the policy is being defined.

On top of resources related to the logical structure of the
application, we provide also primitives to obtain information
about the percentage of CPU, memory, and network usage
incurred by an actor. Together these quantifiers can be used
in conditions – with standard conditionals and boolean opera-
tors – to indicate when to trigger a particular behavior. The
conditions could also be empty and indicate True to trigger
the behavior.

Elasticity behaviors, shown in Table[1], indicate to the
runtime system that a certain configuration of the actors is
desirable now. The first command, colocate(\( at_1, at_2, cd \))
indicates to the runtime system that whenever the condition \( cd \)
as described above is satisfied on actors of type \( at_1 \) and \( at_2 \)
the system should strive to preserve the concerned actors in
the same server. Notice that the condition \( cd \) can restrict the
relation between the actors of type \( at_1 \) and \( at_2 \). For example,
using the resource \( ref(at_2) \) one could restrict that the actor of
type \( at_2 \) be pointed by some field of the actor of type \( at_1 \).
This is typically the case when the programmer knows that two
actors related by \( cd \) should interact often, or are concerned
with a common service of the application. Conversely, the
syntax separate(\( at_1, at_2, cd \)) instructs the system to attempt
to keep the actors separate whenever resources are available if
\( cd \) is satisfied. This could be used for example if both actors
run computationally demanding activities, thus potentially ob-
structing each other’s operation if colocated. The rule pin(\( at \))
indicates that whenever possible actors of type \( at \) should not be
moved. This could be important for services that have to
be highly available, where migration could poorly impact
availability during the transition from one server to another.
Finally, isolate(\( at, cd, re \)) indicates to the runtime system the
desire to keep actors of type \( at \) in a server of their own when
resources are available and condition \( cd \) is met. The additional
condition \( re \) indicates conditions on the desired resources of
the receiving node after the migration. Thus conditions only
involving the resources cpu, mem or net are allowed. This

policies are described in a separate file, and can use elements
of the syntax presented in Table[1]. This specific syntax allows
a programmer to specify policies at the level of actor types
(ranged over with the variable \( at \) in the table). The reserved
keyword any is used to designate any actor type. Hence, our
configuration language integrates the elasticity concerns with
actors, fields and function names from the application codes.
is useful when the programmer knows ahead of time that resource demands will increase temporarily.

Many of the elasticity decisions are predicated over particular conditions of the runtime system state. To express these conditions, the programming model provides statistics about the resource usage of actors. In particular, these resources statistics are calculated over fixed time slots, and the queries always refer to the last completed time slot. To simplify, we consider that these statistics are collected based on the activity of the last minute.

As we explained in Section 2, there are two elasticity policies which may improve the performance of B+ tree application during the runtime:

\[
\text{separate}(\text{InnerNode}, \text{LeafNode},) \\
\text{colocate}(\text{InnerNode}, \text{InnerNode}, \text{ref}(\text{children}))
\]

The first policy tries to avoid to put any InnerNode actor and LeafNode actor on the same server. The second policy requires to colocate InnerNode actors with parent-child relationship. In B+ tree, if one InnerNode actor is the parent of another InnerNode actor, the first actor must has a reference to the second actor via its field \text{children}.

**Discussion.** We remark that there are many possible alternatives for our configuration language, including more precise runtime statistics, more precise elasticity migration decisions (which could be embedded into the program logic itself), and more specific choices for the migration of the actors. Ours is a first attempt to illustrate the need for such fine-grained elasticity policies. We will in future work consider finer constructs for our language.

### 4 Elasticity Management Service

In this section, we introduce the design of our elasticity management service which implements the runtime system for the configuration language introduced in the previous section. The actual duration of a time slot is configurable.

Figure 3 illustrates the architecture of the runtime system. It contains one Global Elasticity Monitor (GEM) and multiple Local Elasticity Monitors (LEMs). The GEM constantly monitors how much resources are being used by an application, and adjusts the total resources (e.g., the scale of cluster) that are being used by the application accordingly. In order to avoid server overload or underutilization, the GEM informs the overloaded servers about idle servers for scaling up, and also asks underutilized servers to migrate their actors to a single server for scaling down.

Each LEM is responsible for managing actors running on top of one single server. Once an LEM receives the list of idle servers from GEM, indicating its associated server is overloaded, the LEM firstly sorts all the local actors in descending order of resource usage. The LEM selects the most resource consuming actor along with all the other actors that communicate with it. The communication strength between two actors is determined by the behaviors and conditions in elasticity policies defined by programmers, and also by tracing actor communication patterns at the runtime. Then the LEM will trigger a set of migration events for the chosen actors. These events are inserted into the pending migration event queue. This procedure continues until enough number of actors have been chosen for migration, and server’s resource usage comes back to normal.

At the same time, every local actor periodically aggregates its resource usage and checks the elasticity policies. If all conditions of a policy are satisfied, the actor informs the LEM of the corresponding elasticity behavior. It is not straightforward how to implement elasticity behaviors in a correct and efficient way. A behavior may result in unacceptably high migration overheads which outweigh its benefits. Complicating matters further is that for a given behavior, there might be a wide range of migration choices. For instance, to colocate actor \(a\) and actor \(b\), one could migrate \(a\) to \(b\) or migrate \(b\) to \(a\) and even migrate both \(a\) and \(b\) to another server. Worse, for a given actor \(a\), there might be conflict behaviors like colocate\((a,b)\) and colocate\((a,c)\).

To tackle this problem, the LEM finds the relevant set of behaviors. Thus, we define a transitive relation between any pairs of behaviors. Two behaviors are relevant if they include at least one common actor. The LEM places all the relevant behaviors in the same set. Next the LEM generates all possible sequences of migration actions corresponding to all behaviors within a set. Some sequences may include conflicting migration actions, such as migrating actor \(a\) to server \(S_1\) and migrating actor \(a\) to server \(S_2\). These sequences have been deleted directly. For the rest of the action sequences, the LEM uses a measurement function to pick up the best action sequence. All the actions in the selected sequence are inserted into the pending migration queue.

Afterwards, the LEM starts to process actions in the pending migration queue. For every migration action, the local LEM negotiates with LEMs on the migration-target servers.
If all the remote LEMs accept the migration requests, the local LEM marks the action as accepted and performs the migration action. Otherwise, this action is simply removed from the queue.

Intuitively, the runtime system ensures that policy requests are totally ordered and ensure all-or-nothing semantics, i.e., a policy is executed in its entirety or not executed at all, which preserves the application semantics.

5 Preliminary Evaluation

In this section, we present our preliminary evaluation results. We evaluate the efficacy of our solution, and show that it can yield to substantial performance improvements.

**Experimental setup.** We implemented our elasticity management service in AEON [9] runtime system and implemented Halo 4 Presence Service using AEON. We deployed this application on two Amazon AWS m1.small instances. We created 10 router actors, 10 session actors and 1 observer actor in the presence service. Each session actor had at most 4 player actors. We deploy 10 clients on a m1.medium instance to simulate the game consoles, which repeatedly sends heartbeat messages to the presence service.

**Router actors.** Compared to other actors, router actors are more CPU-intensive because they require to extract data from heartbeat messages. Thus, the runtime system should guarantee the load of router actors are equally distributed among servers. However, without information of the router actors’ resource usage, the runtime system could only apply a simple placement strategy like assigning the equal number of actors to each server.

Since all actors in presence service are dynamically deleted and created, the number of actors on servers may become imbalanced overtime. Consider an example where most session and player actors, which are not CPU-intensive, have been scheduled on one of the two available servers. Upon creating new router actors, the runtime system can follow the simple strategy that balances the number of actors on each server. In this case, all created router actors may be placed on a same server. On the other hand, if the runtime system recognizes the resource usage pattern of router actors, it may decide to balance the number of router actors among two servers, which will certainly lead to imbalanced total number of actors at each server. Figure 4a shows that the workload-aware balance strategy outperforms the simple number of actors (#actors) balance strategy, even if it results in different number of actors at each server.

**Session actors and player actors.** Each session actor has references to player actors which are in the session. Since session actors communicate with player actors on every heartbeat, the latency could significantly be improved if the runtime system colocates these two actor types. To verify this, we perform an experiment in which each server has the same number of router actors, player actors and session actors. In order to minimize the effect of router actors, we reduce their computation workload. Using our programming model, programmers can suggest the runtime system to colocate session actors and their corresponding player actors. In a comparison setup, the runtime system ignores relationships between session actors and player actors and randomly distributes player actors. Figure 4b shows that ignoring references among session and player actors, and random placement leads to 50% additional latency and 33% less throughout.

**Observer actor and session actors.** Though the references between two actors give useful hints to the runtime system, we could not simply colocate all pairs of actors with relevant references. In Halo 4 Presence Service, observer actors also have reference to session actors. However, observer actors only make calls on those session actors when they observe errors, which occur infrequently. In our implementation, we only have one observer actor because there are only 10 session actors. If the runtime system colocate all actors with references, all session actors and player actors must be placed on one single server. The runtime system ignores reference between actors if programmers do not specify them in the policies. The experimental results in Figure 4c clearly indicates that the references could not be a universal rule to determine the placement of actors. The runtime system needs more specific information from programmer, which can help to arbitrate useful actor references.

6 Related Work

**Stateless applications.** Unlike our general programming model for actor-based applications which provides fine-grained elasticity at the level of actors, several works have studied elasticity techniques for “stateless” cloud applications for which migrating individual state does not affect the semantics of the application itself.

There are a few of works [6, 8] which apply machine learning algorithms to understand the workload and predict the required number of virtual machines during the next time period. Those methods manage computing resources at virtual machine level and target stateless applications like web applications. However, actor-based applications require more fine-grained elasticity management at the actor level. Even with the same number of VMs, different placement of actors in those applications may result in different performance.

AWS autoscaling [1] allows users to setup a dynamic cluster which can scale up and down according to user-defined conditions. This service relieves users from scaling adjustment and with the help of load balancer, it achieves automatic elasticity management for stateless applications. However, this service can not be used for stateful applications directly since the scaling adjustment of stateful applications requires state migration.

AWS Lambda [2] is one step further compared to autoscaling. Users only need to upload their codes has called Lambda
function and to specify sources which will trigger Lambda functions. Then the platform is responsible for capability, scalability and availability for their applications. However, Lambda is similar to auto-scaling which only works with stateless applications. According to the introduction of AWS Lambda, it is mostly used to implement back-end services.

Stateful applications. EventWave [5] is an event-driven programming model which supports strict serializability. Though its runtime supports live actor migration without efforts from programmers, programmers still need to specify the migration details (e.g., which actor and which server). In conclusion, it does not have automatic elasticity management.

Microsoft Orleans [7] is a programming language designed for Azure cloud platform. It is based on the actor model and extends the C# language. Its runtime already includes some optimization mechanisms based on actor locality. However, those optimizations are not a real elasticity solution as it ignores too many important factors in elasticity management like resource usage and migration overhead. A complete elasticity solution at least needs to process the scaling adjustment according to the workload. Furthermore, Orleans does not provide programmers any control over the runtime elasticity management of their applications. The runtime system could only obtain limited information by observing the behaviors of actors. This kind of black-box approach may fail to provide efficient elasticity solution in some cases.

AEON [9] is an actor programming model with automatic elasticity. However, its runtime system only supports very simple elasticity management consisting in distributing evenly on servers. The dynamic connection between actors and resources required by actors have evident impact on the overall performance. AEON’s simple elasticity management service could not capture these important features of the applications.

7 Conclusion

In this paper, we focus on the problem and solution space for fine-grained elasticity in actor-based cloud applications. Ideally, when building large-scale distributed applications like a game engine or complex data structures, the application programmer would like to program largely with sequential semantics in mind so that he would not need to worry about the subtle concurrency bugs that may violate the application semantics. Similarly, the application programmer would also like to specify fine-grained elasticity policies for scaling distributed actors with minimal programming overhead. The goal of this work is enabling application programmers to improve the resource efficiency of their cloud applications with little efforts.

There are different ways to implement the elasticity policies. How to effectively translate the elasticity policies into real implementations is still open. However, the runtime system could not follow programmer-defined policies directly. Some policies may conflict with each other, and some policies can promise more benefits. In the future, we are going to extend the LEMs with cost/benefit analysis. We will also investigate how to integrate service level agreements with our programming model to satisfy the quality of services.

Acknowledgments

This research is supported by NSF grants # 1117065, # 1618923, # 1421910, European Research Council grant # FP7-617805 “LiVeSoft – Lightweight Verification of Software” and German Research Foundation under grant # SFB-1053 “MAKI – Multi-mechanism Adaptation for Future Internet”.

---

**Figure 4.** Preliminary Experiment Results for Halo 4 Presence Service.

<table>
<thead>
<tr>
<th></th>
<th>Rand. Placement</th>
<th>Ref. Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput(req/s)</td>
<td>247</td>
<td>371</td>
</tr>
<tr>
<td>Latency(ms)</td>
<td>41.14</td>
<td>27.45</td>
</tr>
</tbody>
</table>

(a) #Actors Balance vs. Workload Balance

<table>
<thead>
<tr>
<th></th>
<th>Ref. Placement</th>
<th>Ignore Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput(req/s)</td>
<td>220</td>
<td>371</td>
</tr>
<tr>
<td>Latency(ms)</td>
<td>46.26</td>
<td>27.45</td>
</tr>
</tbody>
</table>

(b) Random Placement vs. Reference Placement

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateful vs. Stateless</td>
<td>Stateful</td>
<td>Stateful</td>
<td>Stateful</td>
<td>Stateful</td>
<td>Stateless</td>
<td>Stateless</td>
</tr>
<tr>
<td>Elasticity level</td>
<td>Actor</td>
<td>Actor</td>
<td>Actor</td>
<td>Actor</td>
<td>VM</td>
<td>Lambda function</td>
</tr>
<tr>
<td>Programmable elasticity</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
References

    https://aws.amazon.com/autoscaling/

    https://aws.amazon.com/lambda/


