An Akka Mailbox Implementation Facing SDN

Shou-shuo FANG, Hang-yu FAN, Yong LI and De-peng JIN*  
Department of Electronic Engineering, Tsinghua University, Beijing, China 100000  
*Corresponding author

Keywords: Akka, Mailbox, Red-blacktree, Priority, Distributed system.

Abstract. Akka is a highly-extensible lightweight software toolbox. It solves the classic problem of sensing node state in distributed system by abstracting things into Actor models. Because the limited-capacity Mailbox is generally used, in fact, when there is no available space, some high-priority messages will not be able to enter the container, which will lead to some important messages not being processed; in addition, when the message waits to be processed, if more than expected waiting time it will generate feedback to tell the sender, the sender will cause behavior like repeat-operation. The receiver does not know and will process the message as normal, so it will handle this message twice. Based on the red-black tree data structure, this paper improves the original Mailbox architecture so that it has the function of entering or quitting the team according to priority when the team is full. It can also discard the message that the waiting time exceeds the set-value when the message quits, so as to avoid wasting resources.

Introduction

With the expansion of network at the present[1], the traditional network equipment has a large number of complex protocols built in, which leads to a large increase in the cost of network operation and maintenance. The SDN (software-defined Networking) [2] emerges as the times require. SDN separates the data plane from the control plane to flatten the management. The core of the controller in SDN is the control layer which need to enrich the North-South interface to control the network resources and provide services to the application layer. The controller most widely used is Opendaylight (ODL), which supports Openflow and LISP protocols, among a plurality of controllers can be applied to the cluster model. Because of the high reliability and high performance requirements of the system and the consistency of data and logic, infinispan data grid platform, distributed[3] Akka [4] framework and zooker are usually used as the technical solutions to build the controller cluster.

Akka is a highly-extensible lightweight software toolkit, developed by the Scala language, while providing API support for Scala and Java. Its main goal is to achieve improvement of system performance, reliability and scalability as much as possible in concurrent programming. Actor model solves these problems in distributed systems by abstracting things [5].

The Actor model highly abstracts the communication process between models, which makes the development of distributed system greatly simplified. Based on the analysis of mailbox, this paper comprehensively describes the deficiency of mailbox in the Akka actor model, and improves it according to the requirement. Finally the simulation verifies the implementation of this paper are carried out.

Introduction of Akka

Akka Mailbox

Actors using the default Dispatcher only process one message at a time. In order to ensure that other messages arriving during message processing are not discarded, Akka provides the Mailbox structure as message queue of Actor.
Built-in Mailbox Defects

Akka's built-in Mailbox has the following problems in extreme situations.

**Defect 1: When the priority queue is used, the queueing of high-priority messages is insecure and important messages will be lost.**

From the analysis above, we can see that Akka's built-in limited-length Mailboxes inherit the BoundedQueueBasedBlockingQueue class, regardless of the message priority queue. Because the unique criteria of judging if the message is on place is that whether the queue of the current message in the message queue \( l(t) \) is less than the maximum length of the queue \( L \). It has nothing to do with the degree of importance and the priority of the message. Therefore, when an Actor is processing a message that takes a long time and causes the message queue to overflow, it will be unable to correctly receive more important control messages, such as heartbeat and other control signaling. In some scenarios, this problem is fatal. Because the computational tasks exist for a long time, the number of control messages is much smaller than the number of computing messages. At this time, if the target Actor's message queue is in a long-term fully occupied state, as shown in Figure 2, the control message will not be sent to the target message queue for a long time, resulting in loss of control according to Akka's random competition mechanism.

**Defect 2: The reliability and timeliness of messages cannot be properly guaranteed, which will result in waste of resources.**

In the actual environment, the rate of consumption of messages \( r_c \) is a time-varying process. That means, the processing time of the message is not fixed. This has brought certain difficulties to ensure the reliability and timeliness of news. Specifically, the selection of \( T_o \) in reliability is usually a fixed value (as shown in Figure 3). In this case, although the message \( m_c \) to be processed has successfully entered the target Actor queue, the processing time of the message \( m_a, m_b \) located in front of the \( m_c \) in the queue exceeds the expected reply time, resulting in the message producer received no reply within the stipulated time, so that the production end considers the message to be discarded. According to Akka's current framework, the message at the consumer side was unconditionally processed inside the Mailbox, even if the message has been timed out at the originating side. That
means, the message m_c is considered to be discarded, but the consumer still executes m_c contained tasks. Obviously, this operation caused a waste of resources.

Figure 3. Sketch of timeliness defects.

The reliability and timeliness of messages are widely considered in the actual environment, and usually Actor should distinguish the different degree of urgency message. Therefore, the improvement of the above two defects has important research significance.

**Functional Expansion**

Based on the defects of the Akka built-in Mailbox, this section proposes a highly reliable Mailbox implementation scheme based on the Akka framework for the application scenarios of timeliness, reliability, and difference service.

Figure 4. Message container and team behavior.

In order to achieve this type of Mailbox, making it meet the above objectives, this section will introduce specific improvements. The overall improvement is divided into the improvement plan of the message access team behavior control and the improvement plan of the message container. In Mailbox, the inbound behavior and message container are shown in Figure 4. The inbound and outbound activities refer to Akka’s access to the Mailbox message queue. This behavior allows the message queue to be blocked and queued, captain limited the container itself does not have the standards behavior, but the behavior will not directly operate on the container's data structure. Instead, after limited the behavior of the inbound and outbound team operations, operates on an internal and actual message container. The improvement of the message container is the solution to this specific container. Both the improvement of the behavior and the improvement of the message container
Behavioral Control of Message Enqueued

The behavior of the inbound operation is controlled by the offer method of the queue class with capacity constraints when the message queue space is full. This method determines whether to add new messages to the container by comparing the number of messages in the message container and the capacity limit. Therefore, the implementation proposed in this paper needs to modify or expand the function to make inbound high-priority messages capable when the message container space is insufficient. To achieve this goal, the scheme adopted in this paper is that: when a high-priority message attempts to enter the queue, if the message queue is full and there is at least one message in the queue that is lower than the priority of the message to be enqueued, the lowest priority and the last-arrived message in the queue is discarded, and the high-priority message to be inbound is enqueued.

The specific implementation of the control of the enlistment behavior is as follows. This work derives from Akka's built-in BoundedBlockingQueue which controls the incoming and outgoing queues of a finite-length queue, and overrides its offer method. The Offer method is implemented as shown in Algorithm 1. It first judges whether the backing is full. If it is not full, the message is encapsulated and then it is enqueued. If the backing is full, an element e_t is dequeued from the tail of the backing and both priorities are compared. If the message to be enqueued has a higher priority. If it is high, the message after the enqueuing is encapsulated, e_t is discarded. Otherwise, e_t enters the team, and the message to be enqueued is discarded.

```
Algorithm 1: Offer.
Input: Element e.
\[ts \leftarrow \text{get\_current\_time()}
\]
\[\text{wrapped} \leftarrow \text{wrap\_element(e, ts)}\]
\[\text{if backing\_size()} = E \text{ then}\]
\[\quad e_f \leftarrow \text{backing\_peekLast()}\]
\[\quad \text{if priority}(e_f) < \text{priority}(\text{wrapped}) \text{ then}\]
\[\quad \text{backing\_pollLast()}\]
\[\quad \text{backing\_offer}(\text{wrapped})\]
\[\text{else}\]
\[\quad \text{backing\_offer}(\text{wrapped})\]
```

Message Dequeue Behavior Control

The behavior of enqueuing operations is controlled by the poll method of the BoundedBlockingQueue when the message queue space is full. This method sends the elements to the Actor from the end of the queue. In order to ensure that all messages received by the Actor from the Mailbox are valid within the timeliness requirements, the solution adopted in this paper is to add a time-sensitive discrimination process to the poll method, so that the messages meeting the timeliness requirements are dequeued to the Actor and for those are dissatisfied with the time-sensitive information is discarded after it is dequeued.

The specific implementation of the control of the enqueued behavior is as follows. This work derives from Akka's built-in BoundedBlockingQueue which controls the enqueuing and dequeuing of the finite-length queue, and overloads its poll, isEmpty method, and defines a new private method called prune.

Prune is used to eliminate the expired messages within container. isEmpty refers whether the current container is empty, so that Akka can correctly call the poll method. Because there is a possibility that all messages in the container are out of date, the isEmpty method needs to be overridden to make the prune operation first.

In order to make the message queue sensitive to timeliness, the message queue needs to be assigned a timeout period T_o. Users can specify this time according to actual needs with different granularity.
That is the full queue granularity, priority granularity, and even single message granularity. This scheme uses full queue granularity, that is, the entire queue uses the same $T_o$.

The algorithms such as prune, poll, and isEmpty are shown as Algorithm 2, Algorithm 3, and Algorithm 4 respectively. Algorithm 2 performs peek operation on the backing to determine whether the first element of the team is expired. That is, to check if the time of the team entry is longer than $T_o$ from the current time. If it exceeds, dequeue the backing and repeat the above process until an unexpired leader element is found, or backing is empty. Algorithm 3 first performs the prune operation, then dequeues the backing and returns the decapsulated message. Algorithm 4 first performs a prune operation and then returns to check whether the backing is empty.

---

**Algorithm 2: Prune.**

```plaintext
while backing ≠ ∅ And get.current.time() -
    get.timestamp(backing.peek()) > $T_o$ do
    backing.poll()
```

**Algorithm 3: Poll.**

```plaintext
prune()
return backing.poll()
```

**Algorithm 4: IsEmpty.**

```plaintext
prune()
return backing = ∅
```

---

### Message Container Improvements

The message container is the structure that actually holds the message. Akka's built-in limited queue leader's priority Mailbox message container is the PriorityQueue structure. The structure is implemented by a binary heap, with (1) no-priority quantity limitation, (2) complexity of the enqueue and forward dequeuing time $O(\log N)$, (3) complexity of the backward dequeuing time $O(N)$, (4) Coexistence of different messages with the same priority, (5) The order of dequeuing messages with same priority is not stable according to the enqueuing order.

Due to the complexity ($O(N)$)of the backward dequeuing of the binary heap, the efficiency of the enqueuing operation will be degraded when the full-team process is triggered frequently. In addition, the binary heap cannot guarantee that the nature of the same priority elements dequeued by FIFO does not meet the design requirements.

Therefore, this paper uses the red-black tree[6] structure as a message container after investigation. The container has following properties: (1) no-priority quantity limitation, (2) complexity of the enqueuing and forward dequeuing time $O(\log N)$, (3) complexity of the backward dequeuing time $O(\log N)$, (4) Only allowed the same priority exists in one element and so on.

The nature of the container (3) is better than that of the binary stack and does not create a bottleneck for performance. However, the property (4) needs to be improved to meet the design goals. In this work, this improved scheme attaches a unique timestamp identifier to the priority, at the expense of a small amount of space, so that the actual priority identifiers of different messages are unique, and messages with the same user-defined priority can be made timely.

This section proposes a container based on red-black trees. The structure of the message container is shown in Figure 5. It consists of a leader element cache $C_c$ of length 1 and a red-black tree assembly structure $C_RB$.

In this structure, the red-black tree structure serves as the main message storage container, which can store up to $L-1$ encapsulated message elements. The first element cache stores one encapsulated message element. This element is the next element to be dequeued. The function is to improve the efficiency of the peek operation, so as to improve the efficiency of judging whether the queue is empty in the access control. It is worth mentioning that the peek time complexity in the binary stack structure is $O(1)$, and the peek time complexity of the red-black tree structure is $O(\log N)$. In this scenario, we need to optimize this operation due to the large number of peek operations that need to
be performed. By setting a head cache $C_c$ and guaranteeing that the highest priority and the first arrived element in the entire container is always stored in the cache by the algorithm proposed below, the peek method can be simplified to directly return the elements in $C_c$. Therefore, the peek time complexity of PriorityDeque is reduced to $O(1)$.

![Figure 5. Message container diagram.](image)

**Algorithm 5: Offer.**

```
Input: Element $e$.
if $C_c = \emptyset$ then
  $C_c.offer(e)$
else
  if $\text{priority}(e) > \text{priority}(C_c(1))$ then
    $C_{RB}.offer(C_c.poll())$
    $C_c.offer(e)$
  else
    $C_{RB}.offer(e)$
```

**Algorithm 6: Poll.**

```
if $C_c = \emptyset$ then
  return Null
else
  ret $\leftarrow$ $C_c.poll()$
  if $C_{RB} \neq \emptyset$ then
    $C_c.offer(C_{RB}.poll())$
    return ret
```

**Algorithm 7: Poll Last.**

```
if $C_{RB} \neq \emptyset$ then
  return $C_{RB}.pollLast()$
else
  if $C_c \neq \emptyset$ then
    return $C_c.poll()$
  else
    return Null
```

**Algorithm 8: Peek.**

```
if $C_c \neq \emptyset$ then
  return $C_c(1)$
else
  return Null
```

The element flow in and out of the container is shown in Algorithm 5, where Algorithm 5 shows the elements enqueue operation. Algorithm 6 shows the elements in the forward dequeue poll operation. Algorithm 7 shows the elements in the back dequeue poll. Algorithm 8 shows the peek operation. Among them, algorithm 5 compares the element to be enqueued with the element priority
in Cc. If the element priority of Cc is lower than the element to be enqueued, the elements in the buffer are placed into C_RB and the elements in Cc are replaced with elements to be enqueued; if the Cc element has a higher or equal priority than the element to be enqueued, the element to be enqueued is placed in C_RB. Algorithm 6 returns the current element in Cc and places the C_RB forward dequeuing elements into Cc. The algorithm 7 determines whether the C_RB is empty. If the C_RB is empty, the elements in the Cc are dequeued, otherwise the elements in the C_RB are dequeued backward and returned. Algorithm 8 returns directly Cc elements and does not perform any modified operations on Cc.

Simulation and Evaluation

Simulation Introduction

In this section, we compare the BoundedStablePriorityMailbox built in Akka, to the performance of multiple indicators, and present the test results and conclusions. We use PSQ to represent Akka's built-in BoundedStablePriorityMailbox and use PDQ to represent the solution described in this article.

Simulation Environment. We built two test environments. The first one is a single Akka node, one producer, one consumer; the second is three Akka nodes, three producers, and one consumer.

The test environment server configuration is: CPU: 2.7 GHz, 4 cores; memory: 16 GB; network interface rate: 1 Gbps; network delay: about 0.1 ms.

Simulation Scenario.

1) Support capability performance simulation scenario

In this scenario, we construct two types of messages that simulate high-priority control messages and low-priority computing messages, respectively. After the consumer Actor receives the control message, it immediately responds to the producer, which means successful receiving. After receiving the calculation message, it randomly blocks for a certain period of time Tb and returns a calculation completion message. The production end will set the waiting time for reply and record the time interval ΔTsr of the sending message, the total number of No of the replied message, the number of the sending control message Nh, and the total number of reply of received control message N_hack. Production side summaries data after each test. We allocate a large amount of queue space for the production end and assume that the queue will not overflow, so that Nh = N_hack. The number of unexpired dequeued messages (No) is the number of messages that meet the ΔTsr < T_o condition. Finally, we calculate the evaluation indicators Ap, Aa through the above summary data.

This experimental environment, as described in 4.1.1, includes single-node and three-node environments. The relevant parameters of the test are shown in the table 1. We use the following parameters to simulate a scenario where calculations and controls coexist. Here, Actor spends 0.1-0.5 seconds of CPU time on receiving computational messages, and hardly consumes CPU time when receiving control messages. In addition, we assume that the control message is relatively small, about 1%. In this scenario, since the timeout period is set to 1 second, 2-10 computing messages are expected to cause timeouts in the remaining messages in the Mailbox. Therefore, we set the queue length to 10. However, in order to verify that the work is still available when the user incorrectly sets the length of the queue, we have added a queue length of 100, 1000 as a reference.
Table 1. Function Simulation Parameters Table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_o$</td>
<td>Overtime time</td>
<td>1 sec.</td>
</tr>
<tr>
<td>$T_b$</td>
<td>Blocking time</td>
<td>0.1~0.5 sec</td>
</tr>
<tr>
<td>$L$</td>
<td>Queue Length</td>
<td>10,100,1000</td>
</tr>
<tr>
<td>$P_{hp}$</td>
<td>The proportion of high-priority messages</td>
<td>1%</td>
</tr>
<tr>
<td>$r_p$</td>
<td>Message production rate</td>
<td>10 message/second</td>
</tr>
<tr>
<td>$T_t$</td>
<td>Testing time</td>
<td>30 minutes</td>
</tr>
</tbody>
</table>

2) Enqueuing and dequeuing performance simulation scenario

In this scenario, we construct $n_p$ type messages, giving them different priorities. In order to eliminate the interference factors, we performed the enqueuing and dequeuing performance test in a single-node environment, and provided that the consumer-side Actor completes the processing of the message immediately after receiving other than the first message (operation, blocking time is 0). The production end generates a Ni number of messages best-effort in the shortest time, performs an enqueuing operation, and counts the time from the start to the end of the enlistment $T_e$. In order to fill in the message queue as much as possible to more accurately measure the efficiency of the team, the production end will first send a message with the highest priority. When the consumer receives the first message, it first blocks a certain time for the production end to fill the message queue. After the end of the blocking, the consumer starts timing and stops timing when receiving the Ni message to obtain the time interval Td. Finally, we calculate the evaluation indicators Ri, Ro through the above summary data.

The test parameters for this experiment are shown in the Table 2. This experiment simulates the maximum throughput rate of non-blocking messages to get the best performance of the incoming team. In order to avoid the additional impact of operating system thread scheduling, we set the queue length to a larger value and ensure that the simulation results contain fullness by using the first-in-team and then-out-of-team solutions.

Table 2. Performance Simulation Parameters Table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Queue length</td>
<td>5000,50000,500000</td>
</tr>
<tr>
<td>$n_p$</td>
<td>Priority number</td>
<td>500</td>
</tr>
<tr>
<td>$N_e$</td>
<td>Number of incoming messages</td>
<td>L-10</td>
</tr>
</tbody>
</table>

Simulation Results

Supportability Performance Test Results

![Graph A](image1.png)

![Graph B](image2.png)
Figure 6 shows the results of the guarantee capability test.

The important message guarantee capability $A_p$ is defined as the important message dequeuing rate. This definition is shown in Equation 1.

$$A_p = \frac{N_h}{N_i}$$  \hspace{1cm} (1)

The higher the $A_p$, the stronger the ability to protect important messages. In the formula, $N_h$ refers to the number of high-priority message dequeues, and $N_i$ refers to the number of high-priority messages attempting to enqueue.

The time effectiveness guarantee capability $A_a$ is defined as an unexpired message rate. This definition is shown in Formula 2.

$$A_p = \frac{N_u}{N_o}$$  \hspace{1cm} (2)

The higher the $A_a$, the stronger the time-saving guarantee ability. $N_u$ in the formula refers to the number of unexpired messages that are dequeued, and $N_o$ refers to the total number of messages that are dequeued.

From the results shown in the above figure, we can see that the scenario proposed in this paper is 100% guaranteed for the important message guarantee capability indicators, which is much higher than the 40% guarantee capability of existing plans. The proposed scheme is more than 75%, which is much higher than the existing scheme’s capacity of about 10%. In the case of three nodes, the results are similar. In this case, the important message guarantee capability of the existing solution is further reduced, and the capability of this solution remains unchanged. In addition, the time-saving guarantee ability of the scheme is improved in the case of small queue length.

The above results show that the performance indicators of the various guarantee capabilities of the scheme far surpass the existing ones, and the reliability has greatly increased.
Figure 7 shows the results of the enqueuing and dequeuing performance simulation test.

The definition of enqueuing performance $R_i$ and dequeuing performance $R_o$ are the number of attempted enqueued message per unit time and the number of dequeued messages per unit time, as shown in formula 3 and formula 4, respectively.

$$R_i = \frac{N_e}{T_e} \quad (3)$$

$$R_o = \frac{N_d}{T_d} \quad (4)$$

The higher $R_i$ and $R_o$, the higher the enqueue and dequeue performance. In the formula (3) and (4), $N_e$ refers to the total number of attempted enqueuing messages, $T_e$ refers to the time taken to enqueued messages, $N_d$ refers to the total number of dequeued messages, and $T_d$ refers to the time taken on dequeued messages.

From the results shown in the above figure, the performance of the enqueuing operations has a certain degree of decrease, due to the addition of prune, search for priority, etc. but it is in the same order of magnitude as the original planned. The performance of dequeuing operations is about 0.8 orders of magnitude better than the existing solution. The above results show that in terms of the performance of the enqueue and dequeue performance, this plan is basically the same as the original plan and has availability.

**Concluding Remarks:** This article mainly focuses on the two problems: The high-priority messages in the original solution are discarded because the message queue is full, causing the loss of important messages and the reliability and timeliness of the messages cannot be properly guaranteed to cause waste of resources. An improved Akka Mailbox program was proposed. The scheme avoids the situation where important messages are dropped when the message queue space is insufficient, and has availability in terms of the reliability and timeliness. The Mailbox of this solution has several advantages: The unlimited number of priorities; The messages with the same priority are queued in chronological order; High-priority messages can replace low-priority messages when the queue is full; Messages are dequeued with logarithmic time complexity in the forward and backward directions, etc. This article further provides the performance test of the Mailbox. The test results show that the Mailbox mentioned in this article meets the expected performance.

**Acknowledgement**

This research was financially supported by the sub topic of National 973 Basic Science Foundation: Basic Research on Theory of Smart and Cooperative Network (2013CB329105).
References


