УДК 004.272

А. В. Гурин, А. А. Пазников
*Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)*

# Реализация в модели акторов децентрализованной атомарной рассылки без процедуры выбора лидера

*Атомарная рассылка (atomic broadcast) – фундаментальный примитив синхронизации процессов, необходимый для реализации разделяемого состояния в распределенных вычислительных системах (ВС). Для выполнения атомарной рассылки перспективен децентрализованный подход, при котором отсутствует процедура выбора лидера (leaderless). Такой подход позволяет равномерно распределить нагрузку между узлами, обеспечить отказоустойчивость и эффективен в большемасштабных ВС. В настоящее время не существует алгоритмов реализации атомарных рассылок в модели акторов (actor model), которая активно развивается и находит достаточно широкое применение. В отличие от модели передачи сообщений, представленной стандартом MPI, модель акторов реализует динамическое отображение активных объектов (акторов) на потоки и процессы операционной системы. В данной работе в модели акторов реализован децентрализованный алгоритм атомарной рассылки. Высокоуровневая модель акторов позволяет применять созданный инструментарий для построения распределенных приложений широкого спектра. Проведено натурное моделирование на вычислительном кластере. Полученное значение латентности выполнения атомарной рассылки составляет менее 10 мс для подсистем из 20 узлов при отправке 20 запросов в секунду размером 100 байт. В статье приводится описание алгоритма и механизма реагирования на отказы в системе. Алгоритм реализован программно в виде библиотеки для построения распределенных приложений. Созданный программный инструментарий может быть использован для организации отказоустойчивых вычислений в распределенных вычислительных системах, например для реализации репликации данных.*

**Распределенные системы, консенсус, протокол консенсуса, атомарная рассылка, модель акторов, репликация**

Распределенная вычислительная система (ВС) – это множество элементарных машин (ЭМ), взаимодействующих через сеть связи с целью решения сложных задач. Задачи для распределенных ВС представлены в виде параллельных программ, включающих параллельные процессы (потоки).

Под консенсусом (consensus) понимается соглашение между процессами относительно выбираемого значения. Процессы в распределенной ВС используют протокол достижения консенсуса для того, чтобы выбрать одно из значений, которые предложил каждый из процессов. Любой инструментарий для распределенных ВС, включающий поддержку общего состояния для нескольких процессов или необходимость совместного принятия решения о выполнении следующей операции, включает процедуру решения консенсуса.

Алгоритм консенсуса должен учитывать, что в распределенной системе возможны отказы (failures, errors). Процесс или сегмент системы может стать недоступным для остальных процессов в результате программных или аппаратных ошибок (сбоев сетевого оборудования, операционной системы или программы). Алгоритмы консенсуса предлагают различные гарантии относительно устойчивости к той или иной разновидности сбоев. Чтобы определить отказ какого-либо из процессов, используется детектор ошибок (failure detector), который реализует периодическую рассылку сообщений (heartbeat request) процессам, на которые они должны ответить подтверждением (heartbeat response). Если подтверждение не было получено в течение определенного времени, процесс считается отказавшим.

Понятие консенсуса носит фундаментальное значение для параллельного программирования, поскольку консенсус обладает свойством «универсальности» [1]: на базе объекта, реализующего свободный от ожиданий (wait-free) протокол консенсуса для произвольного числа процессов (потоков), могут быть реализованы любые более сложные свободные от ожиданий разделяемые структуры данных [1]–[6].

На практике алгоритмы консенсуса используются для реализации репликации – процесса синхронизации нескольких копий (реплик) заданного объекта. Репликация позволяет повысить отказоустойчивость системы, поскольку если один из процессов отказывает, объект остается доступным для пользователей [7]–[10]. Для репликации необходимо, чтобы копии имели одинаковое состояние и все изменения, внесенные в одну из них, отражались в остальных. Поэтому реплицируемый объект должен соответствовать модели детерминированного конечного автомата [11]: очередное выходное значение детерминированного автомата определяется его исходным состоянием и последовательностью ранее выполненных команд. Несколько копий могут поддерживать одно и то же состояние, если получаемые от клиентов команды выполняются в одном и том же порядке. Именно за обеспечение согласованного порядка получения команд репликами отвечает алгоритм консенсуса. Данная операция может быть реализована с помощью атомарной рассылки (atomic broadcast), которая обладает следующими свойствами [12]:

1. Если сообщение рассылается каким-либо процессом, то все процессы должны получить его.

2. Сообщение не доставляется более одного раза, то есть отсутствуют дубликаты.

3. Если сообщение $m$ было получено одним из корректных (в которых не произошел сбой) процессов, то его также гарантировано получат остальные корректные процессы.

4. Все сообщения упорядочены, т. е. если процесс $p_1$ получил сообщение $m_1$, а затем сообщение $m_2$, то и все остальные процессы должны получить сообщение $m_1$ перед $m_2$.

Задачи обеспечения консенсуса и выполнения атомарной рассылки эквивалентны [12].

Отметим, что атомарная рассылка может быть также использована в практике параллельного программирования (например, в стандарте MPI) в качестве коллективного информационного обмена для линеаризуемого (linearizable) [1] обновления состояния распределенных структур данных и обеспечения отказоустойчивости [13]–[15]. Таким образом, данная операция – один из основных компонентов инструментария параллельного программирования распределенных вычислительных систем [16].

**Децентрализованный алгоритм атомарной рассылки.** В данной работе построен алгоритм на основе подхода, предложенного в [17]. Основная особенность алгоритма состоит в том, что он не базируется на выборе лидера (leader election) среди процессов [18]–[22]. При таком подходе лидер – это узкое место алгоритма, поскольку нагружен больше, чем остальные процессы. Кроме того, отказ лидера приводит к неспособности системы достижения консенсуса. Используемый децентрализованный подход позволяет распределить нагрузку между серверами равномерно. Кроме того, децентрализация позволяет обеспечить функционирование системы при отказах отдельных процессов и организовать атомарную рассылку в большемасштабных системах (каждый процесс взаимодействует с ограниченным числом других диспетчеров, образующих его локальную окрестность) [23], [24].

Применяется стандартный механизм распознавания ошибок при помощи детектора отказов вместе с механизмом раннего завершения. В данной работе не предполагается появления в системе византийских ошибок (byzantine faults): отказ процесса означает то, что он перестает посылать сообщения и отвечать на запросы.

Для передачи сообщений в рамках используемого подхода применяется оверлейная сеть (overlay network). Параметры графа, ее описывающего, влияют как на надежность алгоритма, так и на его производительность. Значение вершинной связности графа $k$ определяет, сколько процессов должны отказать, чтобы система была разбита на несвязанные между собой подсети. Реберная связность $\lambda$ отражает возможные ошибки в коммутационном оборудовании. Как вершинная, так и реберная связности ограничены степенью $d$ графа $G$: $k \le \lambda \le d(G)$ [25]. Если $k = \lambda = d(G)$, то граф $G$ считается оптимально связным.

В рамках используемого подхода каждое сообщение рассылается всем процессам сети, соответственно диаметр графа $D(G)$ влияет на время реализации пересылок. В свою очередь, диаметр ошибки $D_f(G, f)$ определяет, насколько сильно отказы серверов влияют на продолжительность рассылки [26].

Для организации рассылки применяется биномиальный граф (Binomial Graph, BMG) [27] и граф $G_s(n, d)$ [28]. Данные графы – субоптимально связные и имеют достаточно небольшой диаметр, однако диаметр $G_s(n, d)$ незначительно изменяется при наличии отказов отдельных процессов.

BMG (рис. 1) – это граф, состоящий из $n$ вершин, в котором два процесса $p_i$ соединены, если хотя бы для одного из $0 \le l \le \lfloor \log_2 n \rfloor$ справедливо любое из следующих условий:

$$i = j + 2^l \pmod n;$$
$$i = n + j - 2^l \pmod n.$$

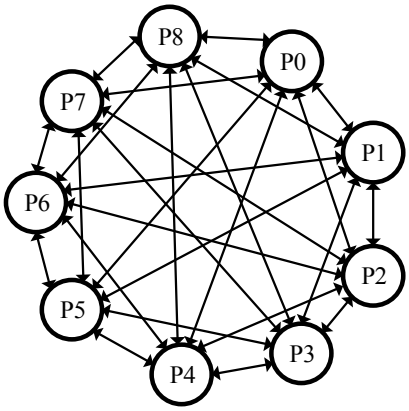



*Рис. 1*

Связность графа BMG, а значит, и надежность построенной на его основе системы определяется количеством его вершин, при этом предлагаемый уровень надежности чаще всего либо чрезмерен, либо недостаточен [17]. В реальной системе рекомендуется использовать $G_s(n, d)$ – граф со степенью $d$ и количеством вершин $n$, который позволяет подбирать необходимый уровень надежности (изменяя параметр $d$); кроме того, при выполнении условия $n \le d^3 + d$, граф имеет квазиминимальный диаметр т. е. диаметр максимум на единицу больше, чем нижняя граница $[\log_d (n(d-1)+d)] - 1$ [28]. Поскольку BMG построить проще, при дальнейшем разборе алгоритма будем использовать именно его, однако разработанный инструментарий предоставляет возможность использования обоих графов.

Для отслеживания отказавших процессов и исключения их из списка ожидания сообщений в алгоритме используется механизм раннего завершения. Для завершения одной итерации алгоритма процессы должны разослать свое сообщение и получить сообщения от всех остальных процессов. Каждый сервер обрабатывает полученные уведомления от детекторов ошибок и определяет, от каких процессов можно не дожидаться сообщений. Отказавший процесс мог успеть послать свое сообщение соседним процессам (согласно топологии) – в таком случае это сообщение все еще можно атомарно разослать.

Для отслеживания возможных владельцев сообщения каждый сервер хранит массив графов $g$, по одному на каждый процесс системы. Как только детектор отказов определяет процесс как отказавший, коммуникация с ним прекращается, а всем соседям отправляется уведомление. Получив уведомление об отказе, другой процесс может быть уверен в том, что у отправителя отсутствует сообщение отказавшего, иначе он отправил бы его до уведомления (порядок получения сообщений для двух конкретных процессов соответствует порядку отправки).

Рис. 2 содержит алгоритм атомарной рассылки, представляющий собой набор процедур, вызываемых в ответ на получение сообщений. В любой момент времени процесс обрабатывает только одно сообщение. Процедура RECEIVE<*ClientMsg*> описывает данные процесса, которые обрабатываются в последующих процедурах. Счетчик текущего раунда хранится в переменной *round* и инкрементируется, как только процессы завершают атомарную рассылку очередного набора сообщений.

Процедуры обработки получаемых сообщений обозначены как RECEIVE<*Tun_сообщения*> (*Данные_сообщения*). Полученное от клиента системы сообщение передается в функцию RECEIVE<*ClientMsg*>. Сообщение помещается в очередь и, если в текущем раунде процесс не отправлял свое сообщение, рассылается остальным при помощи SENDMYMSG.

Функция SENDMYMSG отправляет сообщение $m$ из очереди процесса. Если очередь пуста, то будет отправлено сообщение без данных. Процесс посылает $m$ всем своим соседям (строки 4–6). Вместе с $m$ посылается номер текущего раунда *round*. Отправленное сообщение помещается во множество сообщений $M$, полученных в текущем раунде.

Функция RECEIVE<*Msg*> вызывается в ответ на получение сообщения $m_s$ от другого процесса. Существует вероятность, что процесс получит либо устаревшее сообщение $m_s$ из прошлого раунда, либо $m_s$ от процесса, который уже успел начать следующий раунд. В первом случае сообщение отбрасывается, а во втором его обработка откладывается (строка 3: функция RECEIVELATER помещает сообщение в буфер, из которого оно может быть впоследствии извлечено и снова отправлено на обработку функции RECEIVE<*Msg*>). Если $m_s$ соответствует текущему раунду, то, если оно уже не было получено ранее, оно ретранслируется всем соседям (при этом процесс-отправитель остается прежним – строка 7) и добавляется в $M$. Граф отслеживания, соответствующий данному сообщению, очищается (строка 10).

**Данные процесса**

$n$ – количество серверов; $E$ – множество вершин графа топологии; $F$ – множестово уведомлений об отказах;
*round* – номер текущего раунда; $M$ – множестово сообщений текущего раунда;
*sent* – флаг отправки сообщения в текущем раунде; $G[n] = \{g_0, ..., g_{n-1}\}$ – графы отслеживания;
$P[n] = \{p_0, ..., p_{n-1}\}$ – массив процессов системы; *this* – индекс процесса, на котором запущен алгоритм;
$Q$ – очередь сообщений на отправку

**Обработка сообщения от клиента**

$m$ – полученное от клиента сообщение

```
1  Receive<ClientMsg> (m)
2   Q.ENQ(m)
3   if !sent then SENDMYMSG() end if
```

**Отправка сообщения**

```
1  Function SENDMYMSG()
2   if Q.ISEMPTY() then Q.ENQ(EMPTY_MSG) end if
3   m = Q.DEQ()
4   foreach p_i in E where (p_this, p_i) do
5    send Msg(m, round) to p_i
6   end foreach
7   M.ADD(m)
8   sent = true
```

**Обработка сообщения от процесса**

$m_s$ – сообщение;
$r$ – раунд, в который было послано сообщение;
$p_s$ – процесс, приславший сообщение

```
1  Function RECEIVE<Msg> (m_s, r) from p_s
2   if r < round then return end if
3   if r > round then RECLATER(m_s, r, p_s) end if
4   if !sent then SENDMYMSG() end if
5   if !M.CONTAINS(m_s) then
6    foreach p_i in E where (p_this, p_i) {
7     resend MSG(m_s, r) to p_i
8    end foreach
9    M.ADD(m_s)
10   g_s.CLEAR()
11  end if
12  TESTTERM()
```

**Проверка завершения раунда**

```
1  Function TESTTERM()
2   if G.ISEMPTY() then
3    SORT(M)
4    foreach m_i in M do
5     deliver (m_i)
6    end foreach
7    foreach p_i where !M.Contains(m_i) do
8     G.Remove(g_i)
9    end foreach
10   foreach (p_cr, p_s) in F where G.CONTAINS(g_cr) do
11    RESEND Failure(p_crash, p_s)
12   end foreach
13   foreach g_i in G do
14    g_i.RESET()
15   end foreach
16   round = round + 1
17   sent = false
18   M.CLEAR()
19   RECEIVEALLPOSTPONED()
20  end if
```

**Обработка сообщения об отказе**

$p_{cr}$ – отказавший процесс;
$p_s$ – процесс, приславший сообщение

```
1  Function RECEIVE<Failure> (p_cr) from p_s
2   foreach p_i in E where (p_this, p_i) do
3    resend Failure(p_cr) to p_i
4   end foreach
5   F.ADD((p_cr, p_s))
6   foreach g_i in G do
7    if !g_i.CONVER(p_cr) then continue end if
8    if !g_i.HASCH(p_cr) then
9     Qr.ENQ(p_cr)
10    foreach (p_cr, p_k) in E do
11     if p_k ≠ p_s and p_k ≠ p_i then
12      Qr.ENQ((p_cr, p_k))
13     end if
14    end foreach
15    while (!Qr.ISEMPTY()) do
16     (p_x, p_y) = Qr.DEQ()
17     g_i.ADDVER(p_y)
18     g_i.ADDEDG(p_x, p_y)
19     foreach {p_y,*} in F do
20      foreach p_z in E where (p_y, p_z) do
21       if p_z ≠ p_this and !F.CONTAINS((p_y, p_z)) then
22        Qr.ENQ((p_y, p_z))
23       end if
24      end foreach
25     end foreach
26    end while
27   else if g_i.HASCH(p_cr) then
28    g_i.REMEDG((p_cr, p_s))
29    if !g_i.HASPAR(p_s) then g_i.REMVER(p_s) end if
30    if !g_i.HASCH(p_cr) then
31     Qt; Qt.ENQ(p_cr)
32     while (!Qt.ISEMPTY()) then
33      v = Qt.DEQ()
34      foreach v_k in g_i do
35       g_i.REMEDG((v_k, v))
36       if (!g_i.HASCH(v_k)) then Qt.ENQ(v_k)
37      end foreach
38      g_i.REMVER(v)
39     end while
40    end if
41    foreach v_k in g_i where ! g_i.PATHEXISTS(v_k, p_i) do
42     g_i.REMVER(v_k)
43     g_i.REMEDG((v_k, *))
44    end foreach
45   end if
46   if g_i.CONTAINSONLYVERTEXES(F) then
47    g_i.CLEAR()
48   end if
49  end foreach
50  TESTTERM()
```

*Рис. 2*

Функция TESTTERM реализует проверку, завершился ли текущий раунд атомарной рассылки сообщений. Раунд завершен, если все графы отслеживания пусты (строка 2), т. е. получены сообщения от всех корректных процессов. В этом случае $M$ сортируется, после чего обрабатываются необходимым образом (строка 5). Далее из $g$ удаляются графы процессов, от которых сообщение не было получено в текущем раунде (строки 7–9), и заново рассылаются уведомления об ошибках, если указанный в них отказавший процесс все еще присутствует в массиве $g$ (строки 10–12). Затем графы отслеживания заново инициализируются, увеличивается номер раунда, очищается $M$, а отложенные ранее (строка 16) сообщения снова отправляются на обработку.

Процедура RECEIVE<*Failure*> вызывается в ответ на получение уведомления об отказе процесса $p_{\text{cr}}$, которое прислал $p_s$. Все полученные уведомления ретранслируются соседям и помещаются во множество $F$. После этого в каждый из графов отслеживания вносятся изменения, отражающие возникший отказ. Существует три сценария при внесении изменений в очередной граф $g_i$.

Первый сценарий – граф $g_i$ не содержит вершины, соответствующей отказавшему процессу. В этом случае не требуется каких-либо действий (строка 7).

Второй сценарий – граф $g_i$ содержит вершину отказавшего процесса $p_{\text{cr}}$, у которой нет потомков, т. е. она еще не была развернута (строки 8–27). В этом случае данная вершина разворачивается. Для этого создается очередь $Q_r$, в которую помещаются дуги соседей соответствующего процесса (за исключением процесса, который прислал уведомление, и процесса, сообщение которого отслеживается в данном графе $g_i$ – строка 11). Затем выполняется цикл, который завершается, как только очередь станет пустой. На каждой итерации извлекается очередная дуга $(p_x, p_y)$ из очереди и добавляется вместе с вершиной $p_y$, если ее еще там нет, в граф. При этом если процесс $p_y$ ранее упоминался в уведомлениях об ошибках (строка 19, $*$ означает любой процесс), он также разворачивается, добавляется в очередь (за исключением дуг, которые присутствуют в $F$ – строка 21).

Третий сценарий – граф содержит вершину отказавшего процесса, у которой есть потомки, т. е. она уже была развернута (строки 27–45). В этом случае удаляется дуга $(p_{\text{cr}}, p_s)$ и вершина $p_s$ (если у нее не осталось родителей). Если после удаления вершина $p_{\text{cr}}$ осталась без потомков, то ее необходимо также удалить, как и ее родителей, если у них также перед удалением осталось по одному потомку. Данная операция реализуется в цикле, в котором такие родители помещаются в очередь, а затем удаляются (строки 32–39). В конце удаляются висячие вершины, у которых не существует пути до вершины $p_i$, т. е. процесса, чье сообщение отслеживается (строки 41–44). Если граф содержит только вершины, соответствующие процессам, об отказе которых уже известно, то он очищается.

**Алгоритм атомарной рассылки в модели акторов.** *Модель акторов в библиотеке Akka.NET.* Модель акторов – модель параллельных вычислений, строящаяся вокруг понятия «актор», обозначающего универсальный примитив параллельного выполнения операций. В качестве инструмента реализации алгоритма в данной работе была выбрана библиотека Akka.NET [29], [30], реализующая данную модель. Akka.NET обеспечивает масштабируемую, распределенную обработку данных и в настоящее время интенсивно развивается. В распределенном приложении, построенном при помощи Akka.NET, основным блоком служит актор, который поддерживает операции отправки-получения сообщений другим акторам. В ответ на получение сообщения актор может выполнить какую-либо операцию. Состояние актора изолировано от остальных акторов и может изменяться только им самим в ответ на получаемые сообщения. Каждый актор содержит «почтовый ящик» (Mailbox) – буфер, в который помещаются полученные сообщения, которые затем обрабатываются в порядке очереди. Все акторы выполняются в отдельных потоках, однако каждый из них обрабатывает только одно сообщение в каждый момент времени. Определенного порядка, в котором сообщения будут помещены в Mailbox, нет, однако если рассматривать коммуникацию между двумя произвольными акторами, то сообщения должны быть получены в том же порядке, в каком они были посланы.

Группа акторов, разделяющих общую конфигурацию, объединяются в систему акторов. Акторы в такой системе формируют иерархию. Они могут создавать другие акторы, чтобы делегировать им задачи. На вершине иерархии находится корневой актор (/) с двумя потомками: /user и /system. Актор /user – это родитель всех создаваемых пользователем Akka.NET акторов, стартовая точка для разработчиков приложения.

Akka.Remote позволяет разворачивать системы акторов на нескольких вычислительных узлах ВС. Идентифицировать актор можно при помощи уникального адреса, формируемого аналогично URL.

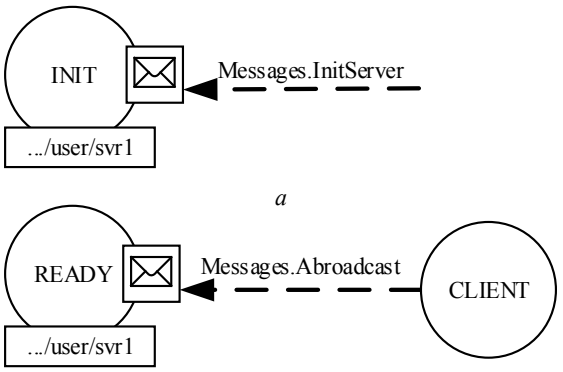
Система акторов создается посредством вызова метода ActorSystem.Create(name, config), в который передается имя и, при необходимости, объект конфигурации. Тип актора объявляется при помощи наследования от класса ReceiveActor. Создать экземпляр актора можно при помощи вызова метода ActorOf<Тип_Актора> на объекте системы акторов или внутри определения класса актора на унаследованном от RecieveActor свойстве Context. Поведение актора описывается посредством вызова методов Receive<T>(func), где T – тип получаемого сообщения, func – функция, которая будет вызвана в ответ на получение сообщения. Внутри обработчика func доступно свойство для идентификации отправителя – Sender. Тип сообщения может быть любым, однако его необходимо сделать неизменяемым. Функции Receive для всех необходимых типов сообщений можно вызвать в конструкторе класса актора, однако если набор получаемых сообщений и реакций на них может изменяться, каждый из наборов определяется в отдельных методах, которые необходимо передавать в метод Become(method), в те моменты, когда поведение актора должно быть изменено.

Если актор в данный момент не может обработать определенное сообщение, есть возможность поместить его в специальный буфер – Stash, из которого оно может быть извлечено и снова помещено в Mailbox позже.

Сообщение может быть отправлено одним из трех методов: Tell, Forward и Ask, на объекте, реализующем интерфейс IActorRef, который представляет актор-получатель. Такой объект актор может получить, создав актор (через Context или ActorSystem) либо указав его адрес объекту ActorSelection, который впоследствии попытается связаться с актором по указанному адресу и вернет объект IActorRef. Методы Tell и Forward являются неблокирующими, они посылают сообщение другому актору, не дожидаясь ответа и завершаясь немедленно. Метод Forward отличается тем, что в качестве отправителя сообщения указывается содержимое свойства Sender в текущем контексте, т. е. даже если сообщение будет передано через несколько промежуточных акторов, в нем всегда будет храниться информация об акторе, который изначально отправил данное сообщение. Метод Ask – блокирующий, он реализует ожидание ответа от получателя.

Отследить возникновение отказов в системе можно при помощи вызова метода Context.Watch(IActorRef ref). В случае, если в дальнейшем указанный актор перестает отвечать на запросы обнаружения отказа (heartbeats, выполняются автоматически компонентом Akka.Remote), следящий актор получает сообщение Terminated с информацией об отказе.

*Реализация алгоритма атомарной рассылки в модели акторов.* Акторы типа ServerActor играют роль процессов (серверов) и реализуют атомарную рассылку сообщения. Для первого старта системы необходимо указать адрес и порт хоста, а также количество процессов, которые необходимо создать. Каждый ServerActor после создания находится в состоянии ожидания инициализации: он ждет получения инициализационного сообщения Messages.InitServer (рис. 3, *а*). Сообщения других типов откладываются в Stash.



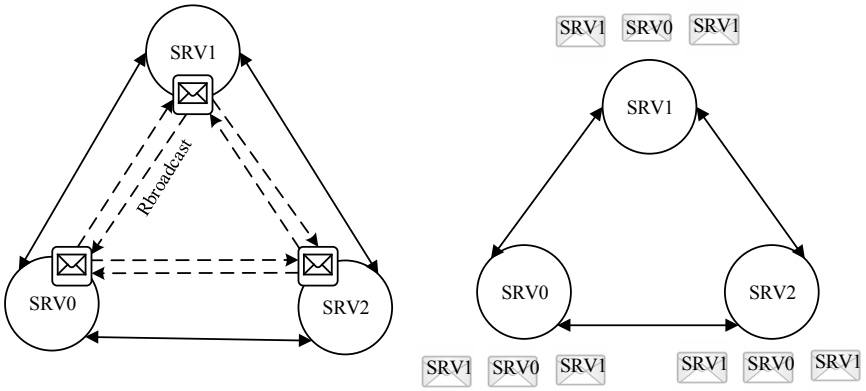

*а*

*б*

*Рис. 3*

Сообщения Messages.InitServer содержат данные, необходимые для начала работы алгоритма: массив всех ссылок IActorRef на акторы системы, выбранный граф, количество информации, которое необходимо выводить при помощи актора LogActor, номер раунда, а также информация об узлах. Акторы вычисляют граф топологии (хранится в виде списков смежности), набор своих соседей, а также запускают отслеживание отказов. После инициализации ServerActor переходит в состояние Ready при помощи метода Become().

В состоянии Ready актор выполняет алгоритм, описанный на рис. 2. В системе создается вспомогательный актор, моделирующий работу клиента: периодически он посылает сообщения Messages.Abroadcast (рис. 3, *б*), которые должны быть разосланы атомарно. Получив такое сообщение, ServerActor помещает его в очередь и отправляет, если он еще этого не делал в текущем раунде. При отправке сообщение помещается в




*а* *б*

*Рис. 4*

объект Messages.Rbroadcast. Получение этого сообщения запускает функцию, которая проверяет номер раунда сообщения, было ли оно получено ранее (сообщения уникально идентифицируются при помощи Guid), помещает его в массив *M*, ретранслирует соседям, а также посылает свое собственное сообщение, если это не было сделано в данном раунде (рис. 4, *а*). Функция TestTerm вызывается для проверки завершения рассылки. Если она завершается успешно, то сообщения сортируются при помощи Guid и доставляются в одинаковом порядке всеми серверами (рис. 4, *б*), после чего начинается новый раунд рассылки, если очередь сообщений не пуста.

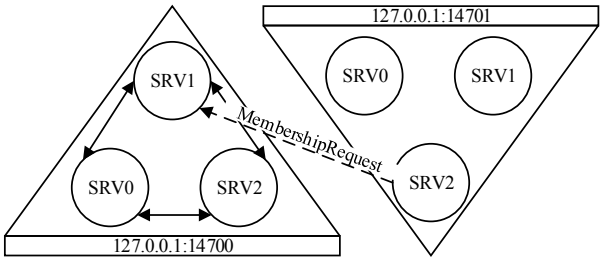
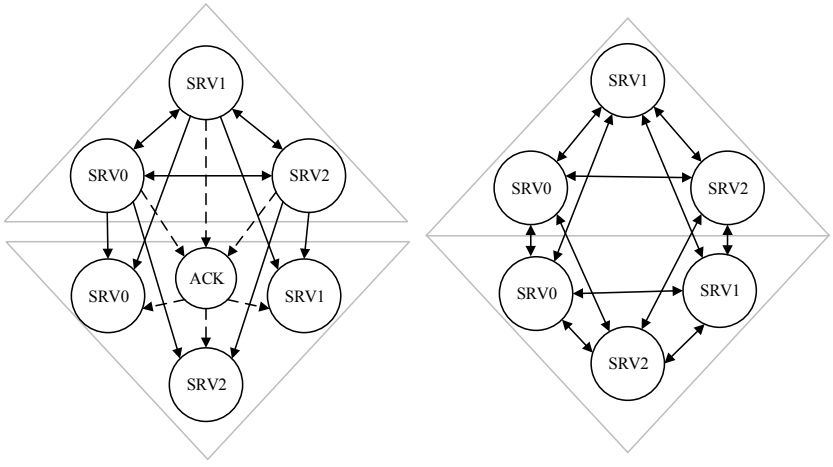



*Рис. 5*

Сообщение Terminated считается полученным, как только один из акторов перестает отвечать на запросы. В ответ на получение Terminated сообщение ретранслируется и изменяется массив графов отслеживания.

Запрос рассылается атомарно в ближайшем раунде, после чего в функции CheckTermination каждый актор повторно формирует топологию. Затем начинается новый раунд, рассылаются сообщения всех акторов, после чего специальному актору ACK, развернутому на желающем добавиться узле, посылается подтверждение с полной информацией о системе, которую можно использовать для инициализации акторов этого узла (рис. 6, *а*). Как только добавленные акторы инициализируются, они смогут закончить раунд посылкой своих сообщений, и система продолжит работать в обычном режиме (рис. 6, *б*).

**Результаты моделирования.** Для анализа эффективности было выполнено моделирование разработанного алгоритма на вычислительном кластере информационно-вычислительного центра Новосибирского национального исследовательского государственного университета. Про-




*а* *б*

*Рис. 6*

водилось два эксперимента – для ВС с общей памятью и ВС с распределенной памятью. Для моделирования на ВС с распределенной памятью использовался кластер, укомплектованный 48 узлами HP BL2 × 220c G6, каждый из которых содержит 4 8-ядерных процессора Intel Xeon E5540, коммуникационная сеть Infiniband 4x QDR. В качестве ВС с общей памятью использовался вычислительный узел на базе HP XL230a Gen9, укомплектованный двумя 12-ядерными процессорами Intel Xeon E5-2680v3 и 192 Гбайт памяти. В качестве показателя эффективности алгоритма использовалась средняя латентность – время выполнения атомарной рассылки.

На рис. 6 представлены значения латентности до момента соглашения серверов (завершение атомарной рассылки) при посылке 20 запросов с частотой 100 байт/с с использованием топологий на основе биномиального графа и графа $G_S$ для ВС на основе общей и распределенной памяти. Для системы с распределенной памятью при увеличении числа серверов существенно возрастает время синхронизации. Это объясняется увеличением накладных расходов при выполнении информационного обмена с использованием межузловой сети связи. Граф $G_S$ характеризуется меньшей латентностью по сравнению с биномиальным графом, по-

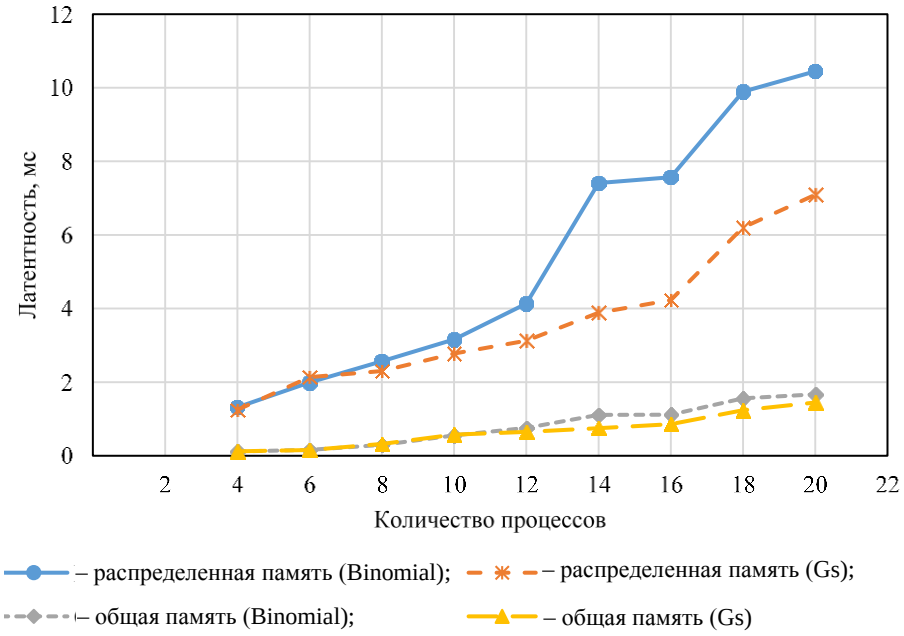



*Рис. 6*

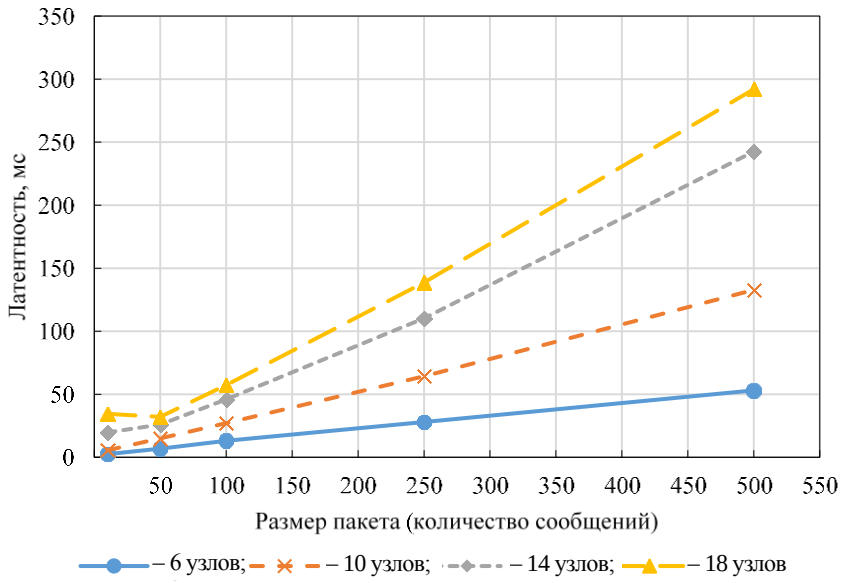



*Рис. 7*

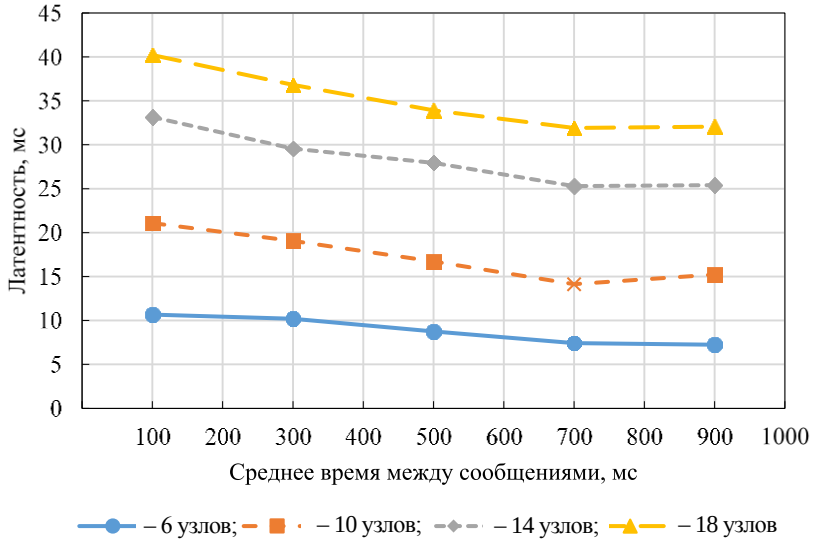



*Рис. 8*

скольку он субоптимально связный и обладает меньшим диаметром.

На рис. 7 представлены задержки в зависимости от размеров пакетов, в которые упаковываются полученные сообщения от клиентов. Время выполнения алгоритма атомарной рассылки увеличивается с ростом размера данных. Это объясняется увеличением времени выполнения передачи данных. Особенно это характерно для подсистем с большим количеством узлов. На рис. 8 представлена зависимость задержки относительно частоты получения сообщений от клиентов. Пока алгоритм успевает завершить раунд до получения очередной порции сообщений, задержки сопоставимы. В противном случае сообщения накапливаются в буфере, пока тот не будет переполнен.

В данной работе в модели акторов был разработан алгоритм атомарной рассылки без процедуры выбора лидера. Алгоритм программно реализован с помощью пакета для создания распределенных приложений Akka.NET. Проведены эксперименты на вычислительных системах с общей и распределенной памятью, показавшие, что алгоритм характеризуется приемлемым уровнем масштабирования. В качестве топологии системы рекомендуется использовать граф $G_S$. Созданный инструментарий можно использовать в качестве основы для построения распределенных приложений в вычислительных системах.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19-07-00784 и при поддержке Совета по грантам Президента РФ для государственной поддержки молодых российских ученых (проект СП-4971.2018.5).

## СПИСОК ЛИТЕРАТУРЫ

1. Herlihy M., Shavit N. The art of multiprocessor programming. Morgan Kaufmann, 2011. 539 p.

2. Paznikov A., Shichkina Y. Algorithms for optimization of processor and memory affinity for remote core locking synchronization in multithreaded applications // Information. 2018. Vol. 9. № 1. P. 1–12.

3. Аненков, А. Д., Пазников А. А. Алгоритмы оптимизации масштабируемого потокобезопасного пула на основе распределяющих деревьев для многоядерных вычислительных систем // Вестн. Томск. гос. ун-та. Управление, вычислительная техника и информатика. 2017. № 39. С. 73–84.

4. Tabakov A., Paznikov A. Algorithms for optimization of relaxed concurrent priority queues in multicore systems // Proc. of the 2019 IEEE Conf. of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus). St. Petersburg, 2019. P. 360–365.

5. Смирнов В. А., Омельниченко А. Р., Пазников А. А. Алгоритмы реализации потокобезопасных ассоциативных массивов на основе транзакционной памяти // Изв. СПбГЭТУ «ЛЭТИ». 2018. № 1. С. 12–18.

6. Табаков А. В., Пазников А. А. Алгоритмы оптимизации потокобезопасных очередей с приоритетом на основе ослабленной семантики выполнения операций // Изв. СПбГЭТУ «ЛЭТИ». 2018. № 10. С. 42–49.

7. Burrows M. The Chubby lock service for loosely-coupled distributed systems // Proc. of the 7th symposium on Operating systems design and implementation. Seattle. 2006. P. 335–350.

8. Hunt P., Konar M., Junqueira F. P. ZooKeeper: wait-free coordination for internet-scale systems // Proc. of the 2010 USENIX conf. on USENIX annual technical conference. Boston. 2010. P. 145–158.

9. Windows azure storage: A highly available cloud storage service with strong consistency / B. Calder, J. Wang, A. Ogus, N. Nilakantan // Proc. of the 23rd ACM Symposium on Operating Systems Principles. Cascais, 2011. P. 143–157.

10. Spanner: Google's globally-distributed database / J. C. Corbett, J. Dean, M. Epstein, A. Fikes // Proc. of the 10th USENIX conf. on Operating Systems Design and Implementation, Hollywood. 2012. P. 251–264.

11. Schneider F. B. Implementing fault-tolerant services using the state machine approach: a tutorial // ACM Computing Surveys (CSUR). 1990. Vol. 22, № 4. P. 299–319.

12. Défago X., Schiper A., Urbán P. Total order broadcast and multicast algorithms: Taxonomy and survey // ACM Computing Surveys (CSUR). 2004. Т. 36, № 4. P. 372–421.

13. Adaptive barrier algorithm in MPI based on analytical evaluations for communication time in the LogP model of parallel computation / V. Zharikov, A. Paznikov, K. Pavsky, V. Pavsky // Proc. Of the Intern. Multi-conference on Industrial Engineering and Modern Technologies (Far East Con-2018). Vladivostok. 2018. P. 1–5.

14. Жариков В. В., Пазников А. А. Адаптивный алгоритм барьерной синхронизации в стандарте MPI на основе модели параллельных вычислений LogP // Изв. СПбГЭТУ «ЛЭТИ». 2018. № 4. С. 26–32.

15. Курносов М. Г. Алгоритмы трансляционно-циклических информационных обменов в иерархических распределенных вычислительных системах // Вестн. комп. и информац. технол. 2011. № 5. С. 27–34.

16. Масштабируемый инструментарий параллельного мультипрограммирования пространственно-распределенных вычислительных систем / В. Г. Хорошевский, М. Г. Курносов, С. Н. Мамойленко, К. В. Павский, А. В. Ефимов, А. А. Пазников, Е. Н. Перышкова // Вестн. СибГУТИ. 2011. № 4. С. 3–19.

17. Poke M., Hoefler T., Glass C. W. AllConcur: Leaderless concurrent atomic broadcast // Proc. of the 26th Intern. Symposium on High-Performance Parallel and Distributed Computing, Washington. 2017. P. 205–218.

18. Ring Paxos: A high-throughput atomic broadcast protocol / P. J. Marandi, M. Primi, N. Schiper, F. Pedone // Proc. of the 2010 IEEE/IFIP Int. Conf. on Dependable Systems and Networks. Chicago. 2010. P. 527–536.

19. Ongaro D., Ousterhout J. In search of an understandable consensus algorithm // Proc. of the 2014 USENIX conf. on USENIX Annual Technical Conf. Philadelphia, 2014. P. 305–320.

20. Lamport L. The part-time parliament // ACM Transactions on Computer Systems (TOCS). 1998. Vol. 16, № 2. P. 133–169.

21. Lamport L. Paxos made simple // ACM SIGACT News. 2001. Vol. 32, № 4. P. 18–25.

22. Junqueira F. P., Reed B. C., Serafini M. Zab: High-performance broadcast for primary-backup systems // Proc. of the 2011 IEEE/IFIP 41st Intern. Conf. on Dependable Systems&Networks. Washington, 2011. P. 245–256.

23. Kurnosov M. G., Paznikov A. A. Efficiency analysis of decentralized grid scheduling with job migration and replication // Proc. of ACM Intern. Conf. on Ubiquitous Information Management and Communication (IMCOM/ICUIMC). Kota Kinabalu. 2013. P. 1–7.

24. Курносов М. Г., Пазников А. А. Децентрализованные алгоритмы диспетчеризации пространственно-распределенных вычислительных систем // Вестн. ТГУ. Управление, вычислительная техника и информатика. 2012. № 1(18). P. 133–143.

25. Dekker A. H., Colbert B. D. Network robustness and graph topology // Proc. of the 27th Australasian conf. on Computer science, Dunedin, 2004. P. 359–368.

26. Krishnamoorthy M. S., Krishnamurthy B. Fault diameter of interconnection networks // Computers & Mathematics with Applications. 1987. Vol. 13, № 5–6. P. 577–582.

27. Dongarra J., Bosilca G., Angskun T. A Scalable and fault-tolerant logical network topology // Proc. 5th Intern. Conf. on Parallel and Distributed Processing and Applications, Niagara Falls. 2007. P. 471–482.

28. Soneoka T., Imase M., Manabe Y. Design of a d-connected digraph with a minimum number of edges and a quasiminimal diameter // Discrete Applied Mathematics. 1990. Vol. 27, № 3. P. 255–265.

29. Maksimovic Z. Akka.NET succinctly. Syncfusion, 2018. 121 p.

30. Brown A. Reactive applications with Akka.NET. Manning, 2019. 264 p.

A. V. Gurin, A. A. Paznikov
*Saint Petersburg Electrotechnical University*

## ALGORITHM FOR CHOOSING LOCAL NEIGHBOURHOODS OF DECENTRALIZED SCHEDULERS IN GEOGRAPHICALLY DISTRIBUTED COMPUTER SYSTEMS

*Atomic broadcast is one of the fundamental synchronization primitives for organizing shared state (consensus) in distributed systems. The promising way for implementing atomic broadcast is decentralized approach, which doesn't involve leader election procedure. Such approach distributes the workload uniformly among the processes, allows robustness and supports large-scale systems. Currently, there are no algorithms for atomic broadcast in actor model, which is one of the most actively developing and is widely used today. In this work, we design decentralized algorithm of atomic broadcast in the actor model. In contract to message passing model (message passing interface, MPI) actor model implements dynamical mapping of active entity (actor) to the threads and processes of operating system. High-level actor model allows to use developed tools for wide range of distributed applications. We have done experiments on computer cluster. The obtained latency for atomic broadcast was less thatn 10 ms for the subsystem of 20 nodes and broadcasting 20 requests of 100 bytes per second. In this paper we describe the developed algorithm and the scheme for resolving system failures. The algorithm is implemented as a software library and may be used for fault-tolerant computing in distributed computer systems, for example, for implementing data replication.*

**Distributed systems, consensus, consensus protocol, atomic broadcast, actor, actor model, replication**