УСТОЙЧИВОЕ К РАЗРЫВАМ СВЯЗИ РАСПРЕДЕЛЕННОЕ МОДЕЛИРОВАНИЕ С AIVIKA

Д.Э. Сорокин (Йошкар-Ола)

Введение и постановка задачи

Реализация распределенного моделирования является непростой задачей во многом в связи с проблемой, известной как «парадокс времени» [1], когда «логическому процессу» приходит сообщение из «прошлого» модели. При рассмотрении методов [2] обычно делается предположение о безусловной надежности соединений, по которым передаются сообщения. Однако если мы пытаемся построить распределенный кластер, используя обычные ненадежные сети, такие как Интернет, то возникает вопрос, а что делать, если произошла ошибка связи?

Здесь автор исходит из того предположения, что если выход из строя какого-нибудь логического процесса является маловероятным событием, то кратковременная потеря соединения между логическими процессами может быть событием вполне рядовым. Умение обрабатывать такие ситуации открывает большие возможности по созданию распределенных имитаций, когда могут быть использованы обычные дешевые компьютеры и обычные ненадежные соединения или популярные сейчас облачные сервисы.

Если для консервативных методов, исключающих саму возможность возникновения парадокса времени, этот вопрос не столь актуален, то для оптимистичных алгоритмов, допускающих возникновение парадокса, вопрос восстановления распределенной имитации весьма актуален, особенно если сбой в передаче сообщений произошел во время отката. Поэтому практичный инструмент, построенный на основе оптимистичного алгоритма, должен предоставить способ восстановления имитации после возобновления прерванного соединения.

Идея решения

Для решения задачи автором настоящей работы предлагается следующая модификация оптимистичного алгоритма «деформации времени» (*англ.* Time Warp) [3, 2] и сопутствующего алгоритма Самади [4, 2] вычисления виртуального глобального времени. Модификация призвана наделить распределенную имитацию способностью восстанавливать себя после временного разрыва связи. Предлагаемый алгоритм асинхронный, распределенный, и он охватывает только те логические процессы, где произошел разрыв связи, а значит, алгоритм масштабируется на большое число логических процессов.

Известный алгоритм деформации времени предполагает, что у нас есть очередь входящих сообщений, которые пришли на некоторый рассматриваемый логический процесс. Также есть очередь отправленных сообщений, которые мы послали другим логическим процессам. При дополнительном использовании алгоритма Самади для определения глобального виртуального времени у нас также появляется очередь исходящих сообщений, которые мы отправили, но на которые еще не пришли уведомления о доставке с других логических процессов. Итого три очереди.

Идея следующая. Когда нам приходит от сетевой системы сообщение о том, что наш логический процесс потерял связь с таким-то удаленным логическим процессом, то сначала дается команда заново подсоединиться, а потом с некоторой временной задержкой запускается специальная восстанавливающая процедура. Проблема в том, что неизвестно, какие из сообщений дошли, а какие нет.

На все входящие сообщения, которые пришли от заданного удаленного логического процесса, посылается уведомление о доставке. Ничего плохого нет, если повторно. Повторные

уведомления на удаленном узле просто будут проигнорированы. Это дает гарантии того, что удаленный логический процесс все-таки когда-нибудь получит все уведомления о доставке.

Кроме того, у нас есть очередь исходящих сообщений, на которые нам пока не пришли уведомления о доставке. Тогда берем все исходящие сообщения заданному удаленному логическому процессу и посылаем их заново. В этом и особенность предлагаемого метода.

Теперь чтобы исключить повторную обработку одних и тех же сообщений, мы каждому сообщению приписываем уникальный номер последовательности в рамках логического процесса - отправителя. При получении такого же входящего сообщения с уже имеющимся в очереди номером последовательности от заданного отправителя, будь то само сообщение или его анти-сообщение, мы просто игнорируем такое сообщение. Поэтому дубликаты не обрабатываются, и мы можем смело заново посылать сообщения.

Только тут есть один момент, что входящее сообщение теоретически может иметь время меньшее глобального времени. Такое сообщение игнорируется. Это значит, что до разрыва связи мы его получили прежде и учли при вычислении глобального времени. Потом сервер времени успел «передвинуть» глобальное время, при этом встроенная в симулятор система очистки ненужных сообщений уже удалила такое сообщение на логическом процессе в целях поддержания устойчивого уровня потребления памяти. Если происходит потеря соединения во время процедуры восстановления, то никаких дополнительных действий предпринимать не нужно. Повторно инициируется та же самая процедура.

Собственно это был сам метод. В часто цитируемом источнике [2] по распределенному моделированию такой метод не описан.

Стоит заметить, что похожая идея с уникальными номерами сообщений и уведомлениями о доставке используется в протоколе SMPP при передаче и обработке привычных SMS-сообщений. Некоторые аналогии были почерпнуты автором из этой области, поскольку он имел опыт разработки соответствующих серверов SMPP.

Алгоритм

Теперь перейдем к более формальным определениям. Для логического процесса *LP* обозначим очередь входящих сообщений через *IQ*, очередь отправленных сообщений через *OQ*, а также очередь исходящих и ожидающих подтверждения сообщений обозначим как *TQ*.

Согласно методу Самади у каждого *LP* есть специальный логический флаг *inFind*, который отвечает за то, передал ли *LP* свое минимальное локальное время серверу времени в течение сеанса определения глобального времени или нет. Если $inFind == да$, то это значит, что мы должны соответствующим образом пометить все последующие уведомления о доставке с той целью, чтобы удаленные логические процессы учли модельное время сообщений в течение своих уже сеансов синхронизации с сервером времени. В этом собственно и суть распределенного алгоритма Самади.

Помимо этого, *LP* еще имеет очередь дискретных событий, но в описании модификации алгоритма она не используется. Для повышения скорости имитации разумно разделять внешние сообщения и локальные события. Каждому сообщению соответствует некоторое событие, но не наоборот. Такое разделение позволяет эффективно реализовать процесс-ориентированную парадигму дискретно-событийного моделирования в рамках распределенной имитации по оптимистичному методу.

Итак, если приходит уведомление о разрыве связи текущего логического процесса *LP* с удаленным логическим процессом *LP'*, то с некоторой временной задержкой применяется следующая процедура восстановления.

***Шаг* 1.** В переменную *ys* помещаем список тех входящих сообщений из *IQ*, которые мы прежде получили от удаленного логического процесса *LP'*.

***Шаг* 2.** Используя текущее значение флага *inFind*, преобразуем сообщения из списка ys в список соответствующих сообщений о доставке для *LP'*, как если бы мы получили сообщения из *ys* в первый раз. Результат из списка сообщений о доставке помещаем в переменную *zs*.

***Шаг* 3.** Посылаем сообщения о доставке *zs* логическому процессу *LP'*.

***Шаг* 4.** В переменную *xs* помещаем список тех сообщений из *TQ*, которые мы прежде послали удаленному логическому процессу *LP'*, но не получили пока на них подтверждений о доставке.

***Шаг* 5.** Посылаем сообщения *xs* логическому процессу *LP'* заново.

Теперь необходима модификация самого алгоритма деформации времени, чтобы игнорировать сообщения с уже существующими уникальными номерами последовательности.

Каждое сообщение наделяем целочисленным атрибутом *sequenceNo*. Заводим на каждом *LP* генератор монотонно возрастающей последовательности целых чисел. Каждый раз, когда во время имитации создается новое сообщение, перед посылкой удаленному логическому процессу мы запрашиваем новое число у генератора и присваиваем его атрибуту *sequenceNo* для заданного сообщения. Откаты во время имитации никак не влияют на генератор – мы просто продолжаем генерировать следующие числа в последовательности.

Таким образом, гарантируется, что для текущего логического процесса все уникальные сообщения из очереди *IQ*, пришедшие от заданного удаленного логического процесса *LP'*, будут иметь разные номера последовательности *sequenceNo*. В целях оптимизации можем ограничиться даже 64-битными целыми числами.

Вышеописанная процедура восстановления соединения может привести к повторной отправке сообщений. Поэтому необходима следующая модификация алгоритма деформации времени, которая применяется при обработке входящего сообщения *m* для текущего логического процесса.

***Шаг* 1.** Проверяем, а нет ли другого сообщения в очереди *IQ*, которое бы пришло от того же логического процесса *LP'*, и которое бы имело тот же номер последовательности *sequenceNo*.

***Шаг* 2.** Если есть такое сообщение, то заданное входящее сообщение *m* просто игнорируем, иначе переходим к шагу 3.

Шаг 3. Если модельное время получения сообщения *m* меньше глобального виртуального времени *GVT*, которое прислал текущему логическому процессу сервер времени во время очередного сеанса вычисления глобального времени, то также игнорируем сообщение *m*, иначе переходим к шагу 4.

Шаг 4. Обрабатываем сообщение *m* согласно алгоритму деформации времени.

Проверку на дубликат необязательно выделять в отдельный шаг. Для эффективной реализации собственно самого алгоритма деформации времени мы можем хранить в очереди *IQ* сообщения, отсортированные по времени обработки. И эту небольшую проверку на дубликат мы можем легко встроить в процедуру добавления нового сообщения в очередь *IQ*.

Аналогично обрабатываем подтверждения *m* о доставке.

Шаг 1. Проверяем, а ожидаем ли мы в очереди *TQ* подтверждения о доставке *m*.

Шаг 2. Если нет, не ожидаем, то просто игнорируем уведомление о доставке *m*, иначе переходим к шагу 3.

Шаг 3. Удаляем соответствующий элемент из очереди *TQ* и обрабатываем уведомление о доставке *m* согласно обычному алгоритму Самади вычисления глобального виртуального времени *GVT*.

Очевидно, что очередь исходящих, но ожидающих уведомления сообщений *TQ*, можно эффективно реализовать через структуру данных Map, где ключом фактически может выступать прототип самого уведомления о доставке за вычетом метки для *inFind*, поскольку для таких прототипов легко ввести операцию сравнения.

Реализация и тестирование

Алгоритм был реализован в общецелевом программном комплексе библиотек дискретно-событийного моделирования Aivika [5, 6], распространяемом в открытых исходных кодах по разрешительной лицензии BSD3, что позволяет свободно использовать Aivika как в академических, так и коммерческих проектах. Код написан на языке программирования Haskell.

При тестировании использовались два компьютера, соединенных в локальную сеть через кабель. Во время распределенной имитации кабель физически изымался из разъема на десятки секунд. С помощью специальных отладочных сообщений проверялось, что разрыв действительно становился непоправимым на уровне используемой системной сетевой программной библиотеки, и что запускался соответствующий метод восстановления имитации.

Удалось добиться очень хороших результатов при использовании систем Linux и macOS. К сожалению, при использовании Windows системная сетевая библиотека часто не переживала такой разрыв соединения, и просто не доходило дело до запуска восстанавливающей процедуры, но это, скорее всего, проблема используемой системной сетевой библиотеки и частной ее реализации для операционной системы Windows. Здесь важно, что код работает на системе Linux, которая чаще других операционных систем используется для создания серверов.

Для большей уверенности использовалась стрессовая тестовая модель [7], намеренно порождающая огромное число сообщений, которыми обменивались компьютеры через сетевой кабель. Использовался генератор псевдослучайных чисел с заданным начальным состоянием, который возвращал воспроизводимую последовательность чисел. Поэтому результат моделирования во всех запусках должен был быть одним и тем же, заранее известным, что и подтвердилось в тестах.

В Aivika также реализован механизм гарантированной самостоятельной остановки логических процессов при превышении определенных временных интервалов во время разрывов связи, которые не могут длиться вечно. Иначе говоря, действует принцип «все или ничего». Либо имитация завершится с положительным ответом, и это будет правильный ответ, поскольку имитация аналитическая, либо прервется, где ведущий логический процесс вернет ошибку, а все ведомые логические процессы самостоятельно остановятся, причем их соответствующие узлы могут продолжать работать в ожидании следующей имитации. В случае серьезного сбоя, повлекшего перезагрузку компьютерного узла, соответствующий вычислительный узел может самостоятельно запуститься как сервис операционной системы. Другими словами, распределенный кластер, так или иначе, но должен восстановиться.

Такая архитектура позволяет создавать простые в управлении и обслуживании надежные сервисы на основе операционной системы Linux, которые бы либо выдавали правильный ответ, либо возвращали ошибку, где простой имитации не может превышать некоторое гарантированное максимальное время.

Помимо этого, возможны средства мониторинга, по которым видно, где какие были размеры очередей, сколько откатов во время имитации, какое локальное и глобальное модельное время, какие узлы временно не отвечают и т.п. Это позволяет оптимизировать модель, находить слабые места в реализации.

Важно отметить, что благодаря уникальному свойству языка программирования Haskell можно контролировать побочные эффекты, просто физически невозможно запрограммировать модель так, что нарушилась бы целостность имитации. Особенно это касается операций ввода-вывода, которые потребуют безусловной синхронизации глобального виртуального времени.

Выводы

Предлагаемая модификация известных алгоритмов довольно проста и легко реализуема. Она хорошо масштабируется, поскольку восстановление распределенной имитации происходит локально, оно децентрализовано. Нет необходимости в управляющем центре. Поэтому метод может быть использован для модели с большим числом логических процессов. Очень важно, что метод асинхронный, не требующий немедленного подтверждения на передачу каждого сообщения. Достаточно лишь одного уведомления, что произошел разрыв связи. Очевидно, что асинхронный метод является значительно более производительным, чем синхронный.

Сложность операции восстановления зависит от числа входящих и исходящих сообщений, но это не должно являться проблемой при использовании горизонта моделирования и механизма очистки по удалению ненужных сообщений, что явно ведет к ограничению размеров очередей входящих и исходящих сообщений, по которым и происходит восстановление распределенной имитации. А без этих механизмов сложно представить практичный симулятор на основе оптимистичного метода.

Так, в Aivika используются пороговые значения для размеров заданных очередей, при превышении которых логический процесс переходит в специальный режим работы, когда обрабатываются только прошлые события в надежде «передвинуть» глобальное время, чтобы можно было удалить ненужные сообщения и сократить размеры очередей. Поэтому можно предполагать, что размеры очередей не будут слишком большими, а значит, и процедура восстановления не займет много времени и ресурсов.

В работе описана лишь часть распределенного модуля системы Aivika, которая значительно сложнее устроена, но автор попытался детально описать один из важных аспектов.

Литература

1. **Окольнишников В.В.** Представление времени в имитационном моделировании // Вычисл. технологии. 2005. Т. 10, № 5.
2. **Fujimoto R.M.** Parallel and Distributed Simulation Systems, Wiley Interscience, 2000.
3. **Jefferson D.R.**, **B. Beckman**, et al. The Time Warp operating systems. 11th Symposium on Operating Systems Principles. 21: 77–93, 1987.
4. **Samadi B.** Distributed simulation, algorithms and performance analysis. Computer Science Department, PhD Thesis, University of California, Los Angeles, 1985.
5. **Сорокин Д.Э.** Айвика: имитационное моделирование в терминах вычислений // Седьмая всероссийская научно-практическая конференция «Имитационное моделирование. Теория и практика» (ИММОД-2015): Труды конф., 21-23 окт. 2015 г., Москва: в 2 т. / Ин-т проблем упр. им. В.А. Трапезникова Рос. Акад. наук ; под общ. ред. С.Н. Васильева, Р.М. Юсупова. – Т.1. – М.: ИПУ РАН, 2015. ISBN 978-5-91450-172-0. С.262-266.
6. Aivika, http://www.aivikasoft.com.
7. Веб-страница, https://github.com/dsorokin/aivika-distributed-test.