УДК 004.75
ББК 32.973.26-018.2

ОСОБЕННОСТИ РАЗРАБОТКИ ИМИТАТОРА ЦЕЛЕВОЙ ОБСТАНОВКИ ПРИ СОЗДАНИИ РАСПРЕДЕЛЕННОГО МОДЕЛИРУЮЩЕГО КОМПЛЕКСА

Лотоцкий А. В.[1]
(*Межгосударственная акционерная корпорация «Вымпел», Москва*)

При разработке программных комплексов и систем, осуществляющих имитационное моделирование с применением распределенных вычислений, одной из ключевых количественных характеристик системы является быстродействие. На примере имитатора целевой обстановки как центрального элемента целого класса подобных систем рассмотрены основные проблемы, влияющие на быстродействие как отдельных элементов, так и системы в целом. Предложен ряд решений и рекомендаций по проектированию таких систем, позволяющих существенно увеличить быстродействие и повысить эффективность системы.

Ключевые слова: многопоточность, *select*()-сервер, синхронный, асинхронный и блокирующий ввод/вывод.

1. Введение

Разработка комплексов и систем распределенного моделирования в подавляющем большинстве случаев осуществляется с использованием клиент-серверного подхода разработки программного обеспечения. Например, при реализации распределенной модели взаимодействия различных средств наблюдения (радиолокационных, оптических и т.д.), коммуникации и управления в качестве сервера может выступать имитатор целевой

[1] *Алексей Владимирович Лотоцкий* (*alototsky@it-omega.com*).

обстановки, а в качестве клиентов – соответствующие средства и системы. Центральным элементом такого моделирующего комплекса является имитатор, обеспечивающий, в частности:

- взаимодействие других элементов системы с сервером и друг с другом;
- синхронизацию модельного времени.

Распределение вычислительных ресурсов между несколькими модулями позволяет повысить быстродействие и эффективность системы в целом и подразумевает организацию взаимодействия между такими модулями и сервером. Такое взаимодействие осуществляется средствами межмашинного и межпроцессного взаимодействия (как правило, используются сетевые протоколы семейства *TCP/IP*, а также именованные каналы, разделяемая память и т.д.). Таким образом, при условии использования всеми модулями единого модельного времени производительность системы в целом определяется как производительностью используемого коммуникационного оборудования, так и производительностью отдельных модулей комплекса и собственно сервера.

При условии, что аппаратная инфраструктура неизменна[2], повышение производительности системы в целом можно достичь повышением производительности либо сервера, либо **всех** модулей, участвующих в распределенном моделировании. Важно отметить, что повышения производительности одного или нескольких модулей в составе комплекса может оказаться недостаточным в силу ограничений, накладываемых единым модельным временем. Если хотя бы один из модулей «заставляет себя ждать», вся система будет время от времени простаивать и её производительность будет снижена. Повышения производительности сервера можно добиться уменьшением времени, за-

[2] *Как правило, так оно и есть. Например, при использовании сетевого взаимодействия между модулями комплекса повышение пропускной способности сети зачастую связано с дорогостоящей заменой оборудования. Если же модули взаимодействуют посредством разделяемой памяти, то при прочих равных увеличение скорости обмена можно достичь только путем замены модулей памяти, а это дорого и не всегда возможно.*

трачиваемого сервером на выборку данных из коммуникационного канала (например, сети или разделяемой памяти) и обработку клиентских запросов. Этот вопрос достаточно подробно рассмотрен, например, в [1] или в [2], однако традиционно быстродействие серверов рассматривается в контексте межсетевого взаимодействия при создании *Web*- или *Intranet*-приложений. В таких приложениях в первую очередь рассматриваются вопросы, связанные с количеством одновременно обслуживаемых клиентов, величиной используемой памяти, задержками при подключении очередного клиента и т.д. Вопросы снижения затрат на обслуживание каждого клиента в таких приложениях рассматриваются в последнюю очередь.

В интересующем нас случае без ограничения общности рассуждений можно считать, что количество подключений к серверу фиксировано и ограничено моделируемым сценарием. Рассмотрим различные варианты реализации сервера при реализации такого моделирующего комплекса.

2. Различные варианты реализации сервера

По большому счету существует всего три варианта реализации подобного сервера:

- однопоточный сервер, использующий неблокирующие или асинхронные функции ввода/вывода операционной системы (далее – ОС) и обслуживающий всех имеющихся клиентов;
- многопоточный сервер, использующий блокирующие функции ввода/вывода ОС (при этом каждый поток обслуживает одного клиента);
- сервер, у которого взаимодействие с клиентами реализовано с помощью специально разработанных для этого функций ядра ОС.

Все остальные варианты реализаций сервера представляют собой сочетания или комбинации упомянутых выше вариантов.

Рассмотрим эти реализации более подробно на примере сетевого взаимодействия между модулями системы.

Обслуживание группы или всех клиентов сервера в *единственном* потоке (или процессе) с использованием *неблокирующих* и/или *асинхронных* функций ввода/вывода предполагает нали-

чие на сервере бесконечного цикла, в котором происходит постоянный опрос состояний соответствующих сетевых дескрипторов. Если в течение определенного времени один или несколько дескрипторов перешли в состояние готовности к чтению/записи, то данные из этих дескрипторов обрабатываются. Важно отметить, что обработка этих данных происходит последовательно. Опрос готовности дескрипторов может осуществляться с помощью:

- ***select*()/*poll*()**. В этом случае ОС осуществляет перебор всех переданных ей дескрипторов и возвращает список тех из них, которые готовы для чтения или записи.
- Плюсами такого подхода является:
  - очевидная простота реализации;
  - переносимость между различными ОС.
- Минусы такого подхода:
  - отсутствие масштабируемости (из-за наличия единственного цикла);
  - последовательная обработка данных (один клиент может «тормозить» всех остальных);
  - дополнительные накладные расходы (необходимость опрашивать дескриптор даже если он «не готов»).
- ***/dev/poll*** – функционирующий аналогично *poll*(), но в отличие от последнего позволяющий единовременно указывать операционной системе список интересующих дескрипторов. К сожалению, удачных переносимых реализаций этой замены *poll*() нет и вряд ли будет.
- ***kqueue*()** – решение, аналогичное */dev/poll* в том смысле, что операционной системе явно указывается список дескрипторов, состояние которых необходимо отслеживать. Плюсом такого решения является возможность избежать накладных расходов, свойственных традиционному *poll*(), но есть и существенные минусы. Например, некоторые реализации *kqueue*() не поддерживают многопоточную модель, и вызов этой функции блокирует не только сам поток, но и весь процесс, что весьма пагубно сказывается на производительности сервера в целом.

- ***epoll*()** – альтернативный вариант сигналов реального времени. Это решение предполагает, что ОС формирует список дескрипторов, «готовых» к чтению/записи и возвращает его в вызывающую подпрограмму по запросу.
- ***sigwaitinfo*()** – накопление и обработка сигналов реального времени. Фактически это решение позволяет накапливать в ОС, а затем при необходимости получить у неё список всех модифицированных дескрипторов. Таким образом, появляется возможность опрашивать только те дескрипторы, состояние которых изменялось за последнее время. Обработка данных, т.е. собственно чтение и запись этих дескрипторов по-прежнему осуществляется последовательно.

Базовая реализация *многопоточного* сервера предполагает, каждый клиент обслуживается на стороне сервера соответствующим потоком с использованием *блокирующих* функций типа *read*()/*write*()/*recv*()/*send*(). К недостаткам такого подхода часто относят необходимость дополнительных затрат по памяти для каждого из создаваемых потоков. Этот недостаток действительно является существенным для *Web*-подобных систем, однако в описываемом случае рассматривается фиксированное и ограниченное количество потоков. Кроме того, современные ОС позволяют управлять количеством памяти, выделяемой потоку, что позволяет снизить негативное влияние указанного недостатка. Важным преимуществом многопоточной реализации является масштабируемость сервера в зависимости от количества используемых процессоров и процессорных ядер и относительно легкая переносимость между ОС. Использование высокоуровневых библиотек типа *Qt*, *wxWidgets* или им подобных позволяет реализовать полностью платформенно-независимый код.

Встраивание серверного кода *в ядро* ОС достаточно трудоемкая операция сама по себе и, как минимум, предполагает наличие достаточно хорошо формализованного протокола, что редко имеет место при разработке исследовательских моделей. Имеющиеся реализации встраиваемого кода (например, для протоколов *NFS* и *HTTP*) плохо переносимы между различными ОС и мало полезны сами по себе, если речь идет о разработке при-

кладных моделирующих комплексов с высокими требованиями к производительности. Поэтому вариант встраивания серверного кода в ядро ОС в дальнейшем рассматриваться не будет.

3. Сравнение многопоточного и *select*()-ориентированного сервера

Для определения наиболее эффективного варианта реализации сервера была разработана простейшая модель, в рамках которой клиенты производили подготовку модельных данных, передавали их имитатору и принимали от него ответ, после чего обмен повторялся. Модельные данные на клиентах готовились посредством специальной функции, выполняющей некоторое количество (N) тригонометрических операций. Фактически этот параметр соответствует вычислительной сложности клиентской модели и определяет количество процессорного времени, затрачиваемого на подготовку модельных данных в одной итерации обмена с имитатором. По условиям эксперимента суммарное количество обращений к серверу оставалось постоянным и составляло 10 000 обращений. Эксперимент проводился на ЭВМ с процессором *Core* 2*Duo* 2,2 ГГц. Наблюдалось суммарное время, затрачиваемое сервером на обработку всех запросов от всех клиентов в зависимости от количества клиентов и количества операций, выполняемых на сервере. Время, затрачиваемое ОС на открытие сетевых соединений, передачу данных, создание и инициализацию потоков, не учитывалось. Синхронизация модельного времени не производилась для большей наглядности результатов. Результаты серий экспериментов для N, равного 500 000 и 10 000, приведены ниже на рис. 1 и рис. 2 соответственно.

Из результатов первой серии экспериментов ($N = 500\,000$) видно, что при единственном подключении производительность сервера (как величина обратная к времени, затрачиваемому на обработку запросов) одинакова как для *select*(), так и для многопоточной реализации сервера. Как только количество подключений к серверу становится два и более, производительность *select*()-сервера падает в два раза, а точнее, в количество раз равное количеству ядер процессора. Большое значение N также

означает, что время для подготовки данных на клиенте достаточно велико (много больше, чем обработка данных на сервере), поэтому запросы к серверу происходят достаточно ритмично и продолжительность их обработки мало зависит от количества подключенных клиентов.

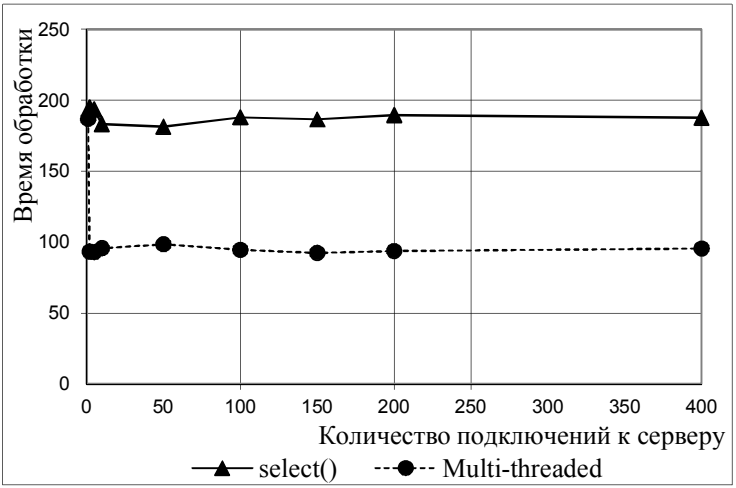

Рис. 1. Время обработки запросов (*N = 500 000*)

Вторая серия экспериментов проводилась с меньшим количеством вычислений на клиентах (*N* = 10 000) и, следовательно, большей частотой запросов к серверу. Из приведенных на рис. 2 результатов видно, что в этом случае выигрыш в производительности у многопоточной реализации сервера значительно больше, чем у *select*()-сервера. Связано это с дополнительными накладными расходами, возникающими при осуществлении перебора сетевых дескрипторов для определения «готовых». Пока количество подключений невелико, время, затрачиваемое на обработку, растет почти линейно из-за того, что обработка запросов происходит достаточно быстро, перебираемые дескрипторы в основном «не готовы» и цикл работает вхолостую тем дольше, чем больше таких дескрипторов. При увеличении количества подключений одновременно готовых дескрипторов ста-

новится больше и тем самым снижается длительность холостого цикла.

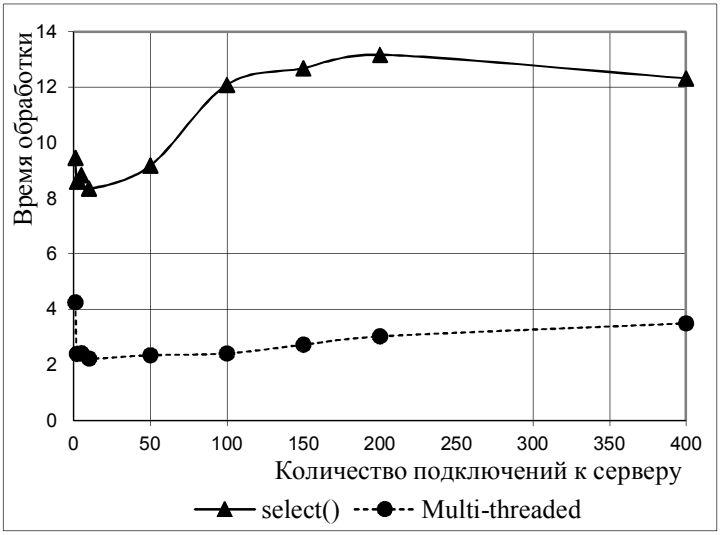

Рис. 2. Время обработки запросов (*N = 10 000*)

Дополнительно следует отметить еще одно немаловажное наблюдение. Часто возникает необходимость размещать все приложения моделирующего комплекса на одной ЭВМ. В этом случае и сервер, и клиенты используют одни и те же вычислительные ресурсы. В описанных выше экспериментах это приводило к тому, что в случае *select*()-сервера одно из ядер процессора нагружалось примерно на 80%, а второе на 20% (вне зависимости от *N*). При использовании многопоточной реализации сервера оба ядра нагружались на 100%. Таким образом, многопоточная реализация сервера позволяет более полно использовать имеющиеся вычислительные ресурсы.

4. Вопросы синхронизации

При проектировании многопоточных приложений особое внимание следует уделять защите от ситуаций, в которых возможно возникновение состояния гонки (*race condition*). Для того чтобы этого избежать, необходимо синхронизировать доступ различных потоков к общим данным. Вне зависимости от выбранного способа взаимодействия между имитатором целевой обстановки и другими модулями комплекса общими данными являются:

- модельное время (считывается подключаемыми к имитатору модулями; обновляется только имитатором);
- сценарий моделирования и связанные с ним данные (считывается подключаемыми к имитатору модулями; обновляется только имитатором);
- текущее состояние каждого модуля, подключенного к имитатору (считывается имитатором и, быть может, некоторыми другими модулями; обновляется только непосредственно данным модулем).

Обращение к этим данным должно быть организовано таким образом, чтобы гарантировать их целостность и непротиворечивость. Для этого в соответствующих блоках программного кода (критических секциях) используются специальные блокирующие примитивы (*mutex*, *read*/*write lock* и т.д.) либо элементы неблокирующей синхронизации (*lock-free* структуры). Определение критических секций и выбор конкретного способа синхронизации в общем случае определяется особенностями проектируемой системы и, в частности, выбранным способом межпроцессного взаимодействия. Кроме того, выбранный способ синхронизации определяет и величину накладных расходов при работе потоков (связанных, например, с ожиданием блокируемых потоков) и тем самым влияет на производительность сервера в целом. В этой статье мы не будем подробно рассматривать вопросы синхронизации. Наша задача учесть влияние накладных расходов при проведении экспериментов.

В описанных выше экспериментах при многопоточной реализации сервера использовался сетевой обмен между клиентами

и имитатором. В этом случае каждый из участников обмена оперирует с копиями данных, снижая тем самым количество критических секций и, как следствие, количество блокировок. Для оставшихся критических секций использовались *read*/*write* блокировки. Наблюдаемое снижение производительности в этом случае было менее 1%.

5. Заключение

Использование многопоточной парадигмы при создании клиент-серверных приложений делает процесс проектирования и разработки таких систем более сложным, чем при использовании *select*()-ориентированного подхода или его модификаций. Вместе с тем, при создании исследовательских моделей такой подход позволяет наиболее полно использовать имеющиеся вычислительные ресурсы и дает возможность создавать более эффективные, масштабируемые и высокопроизводительные приложения.

На основании изложенного выше материала можно рекомендовать построение имитатора целевой обстановки по следующей схеме:

- каждому клиенту выделяется соответствующий серверный поток, в котором используются средства межпроцессного взаимодействия непосредственно с клиентом. Количество подключений определяется конфигурацией моделируемого сценария;
- каждому клиенту выделяется структура данных на стороне имитатора, которая описывает текущее состояние клиента и может обновляться с течением модельного времени. Доступ к этим данным необходимо синхронизовать;
- при наступлении очередного момента модельного времени имитатор передает в межпроцессный канал клиента соответствующий признак. Прочитав этот признак, клиент выполняет обработку данных очередного шага имитации. Поскольку используются блокирующие операции ввода/вывода, то при отсутствии данных в канале клиент будет ожидать. Таким образом, для клиента синхронизация модельного времени выполняется автоматически без дополнительных накладных расходов.

Литература

1. CHANDRA A., MOSBERGER D. *Scalability of Linux Event-Dispatch Mechanisms* // HP Labs Technical Reports. – 2000. – C. 2–20. URL – http://www.hpl.hp.com/techreports/2000/HPL-2000-174.html.
2. STEVENS W.R. *UNIX Network Programming: Networking APIs: Sockets and XTI.* – Upper Saddle River, NJ: Prentice Hall, 1998 – 1009 p.

SOME ASPECTS OF TARGET ENVIRONMENT SIMULATOR DESIGN DURING DEVELOPMENT OF DISTRIBUTED MODELING SYSTEM

Aleksey Lototsky, Interstate Stock Corporation “Vimpel”, Moscow.

Abstract: Performance is one of the most important quantitative indicators of simulation software based on distributed calculations. For a target environment simulator (*being the main element of the whole class of such systems*) *major performance related problems are discussed. Using solutions and proposals provided one can increase overall system efficiency and performance.*

Keywords: multithreading, select()-based server, synchronous/asynchronous/blocking IO.

Статья представлена к публикации
членом редакционной коллегии Н. Н. Бахтадзе