Санкт-Петербургский государственный университет

На правах рукописи

Бойко Павел Валентинович

MAKC DSM:
Система распределённой общей памяти для мультиагентных систем в IoT

05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук

Научный руководитель
доктор физико-математических наук, профессор
Андрианов Сергей Николаевич

Санкт-Петербург – 2017

Оглавление

Введение

Актуальность темы исследования. Становление мультиагентных систем (МАС) как отдельного научного направления пришлось на 1980-е, и уже в середине 1990-х это направление получило широкое признание [52, с. 13]. В течение последующих 15-ти лет возрастающий объем исследований в данной области привел к необходимости систематизации накопленных знаний и появлению первых тематических монографий как в России [10], так и за рубежом [38, 41, 52].

Уже в начале 2000-х годов агентно-ориентированный подход широко применялся для распределённого решения сложных задач в имитационном моделировании производственных процессов, организации работы коллективов роботов и других областях [10, с. 15]. Сегодня мультиагентные технологии выглядят перспективно даже в таких традиционно консервативных сферах, как энергетика [37], отвечая соответствующим требованиям к безопасности. Вопросы же организации коллективов роботов становятся всё более актуальными как в военном, так и гражданском секторах [51], выделяя такие задачи как контроль периметра, обеспечение связи, ликвидация последствий стихийных бедствий и др.

Несмотря на многолетнюю историю развития МАС, до сих пор не существует общепринятого определения понятий «мультиагентная система» и «агент». Например, в российской монографии [10] несколько страниц посвящено определениям и высказываниям об агентах различных авторов, проводится классификация этих определений, а также предлагается собственное определение агента, описывающее десять его ключевых свойств. В качестве альтернативы можно обратиться к зарубежной монографии [41], в которой автор также предлагает собственное, но, в данном случае, очень компактное и достаточно широкое определение мультиагентной системы, обосновывая данное решение тем, что в прошлом было предложено слишком много конкурирующих и зача-

стую противоречащих друг другу определений, чтобы остановиться на одном из них.

Рассмотрим несколько определений мультиагентных систем и агентов из разных источников.

1. Мультиагентные системы — это системы, состоящие из нескольких взаимодействующих вычислительных элементов, называемых агентами. [1]

2. Агент — это активный объект со способностью «воспринимать», «рассуждать» и «действовать». [2]

3. Мультиагентные системы — это системы, включающие в себя множество автономных объектов с различающейся информацией, различающимися интересами, либо обладающих обоими признаками сразу. [3]

Первые два определения выявляют необходимость взаимодействия, а значит – обмена информацией между агентами. Последнее – обращает внимание на другое их свойство (менее очевидное, но не менее характерное): «знания» одного агента зачастую отличаются от «знаний» другого.

Таким образом, выделившись из классического искусственного интеллекта, централизованному подходу решения задач МАС противопоставили децентрализованный, предполагая, что отдельный агент может обладать лишь частичными знаниями, а для решения задачи потребуется взаимодействие группы агентов. Это обусловило концептуальную новизну решений на основе МАС [10, с. 15], стимулом же к развитию направления стали предполагаемые повышенные надежность, гибкость и масштабируемость [38, с. 8] – качества, которые существенно проще обеспечить в децентрализованных многопроцессорных системах, чем в классических – централизованных [1, с. 12].

[1] «Multiagent systems are systems composed of multiple interacting computing elements, known as agents» [52]

[2] «Fundamentally, an agent is an active object with the ability to perceive, reason, and act» [38]

[3] «Multiagent systems are those systems that include multiple autonomous entities with either diverging information or diverging interests, or both» [41]

Однако следует отметить, что упрощая создание конечных распределённых решений, концепции МАС требуют наличия базовых инфраструктурных механизмов, обеспечивающих взаимодействие агентов между собой. В операционных системах в этом качестве обычно предлагаются лишь функции обмена сообщениями (адресного, многоадресного или широковещательного) [9], остальной же функционал необходимо создавать прикладному программисту самостоятельно. Это вызывает определенные сложности, так как параллельное программирование, в особенности программирование многопроцессорных параллельных систем, требует специальных знаний (имеющихся лишь у немногих программистов [9, с. 590]), порождая отдельное направление в программировании со своими образовательными программами, исследованиями и литературой.

Основная трудность в создании подобных механизмов обусловлена описанными выше свойствами агентов, которые позволяют сделать вывод о существовании неотделимой от понятия «агент» задачи гарантирования согласованности или когерентности (англ. coherence) знаний в системе. Задача вызвана, с одной стороны, высокой вероятностью расхождения информации у разных агентов, а с другой, необходимостью их координации и, то есть, синхронизации этой информации. Говоря иначе, обмен информацией в МАС должен осуществляться таким образом, чтобы можно было гарантировать непротиворечивость (консистентность, от англ. consistency) информации в разных узлах системы (в противном случае система вела бы себя рассогласованно, нескоординированно, что противоречит самому понятию «система»).

Задача может быть решена на прикладном уровне и с помощью классических примитивов рассылки сообщений, однако многими авторами было показано, что подобные решения оказываются крайне сложными, что отрицательно сказывается на их стоимости и надежности. Чтобы упростить прикладным разработчикам решение данной задачи, была предложена [32] новая модель – распределённой общей памяти (англ. distributed shared memory – DSM). Мо-

дель оказалась настолько удачной, что вызвала волну исследований в данной области. В последующие годы было создано множество моделей консистентности [15, 20, 25, 29–31, 34, 35], балансирующих между производительностью и предсказуемостью. Параллельно разрабатывались алгоритмы (обзор известных принципиальных решений выполнен в работах [44, 45]), и создавались конечные системы [12–15, 22–24, 26–28, 32, 39, 47, 50, 53].

Если на заре развития концепции DSM основное внимание исследователей было уделено моделям консистентности и попыткам их усовершенствования для обеспечения приемлемой производительности, то затем акцент сместился к алгоритмам, эффективно реализующим ту или иную модель. DSM-системы того времени зачастую были экспериментальными и создавались для проверки той или иной гипотезы. В последние годы прослеживается тенденция разработки DSM-систем, предназначенных для решения вполне определённых задач в конкретной области (к примеру, Zeng и др. в работе 2017 года [39] описывают DSM систему, предназначенную для осуществления коммуникации между виртуальными машинами).

Концепция DSM применима к любым видам МАС или распределённых систем. Однако сфера устройств Интернета вещей (англ. Internet of Things – IoT) – одна из наиболее бурно развивающихся, стимулирующая массовый интерес к распределённым системам – остаётся данной концепцией не охвачена. По мнению автора, данная ситуация сложилась по следующим причинам.

– прежние DSM системы разрабатывались зачастую на экзотических языках и под неиспользуемые сейчас ОС;

– в области IoT часто используются маломощные, но энергоэффективные микропроцессоры, на которых не могут исполняться такие ОС как Linux без катастрофической потери производительности, а зачастую не могут и принципиально, в связи с аппаратными ограничениями – например, отсутствием в микропроцессоре блока управления памятью (англ. memory

management unit – MMU);

– специализированные ОС для подобных микропроцессоров существуют, но большинство из них являются проприетарными[1];

– открытые ОС, которые могут работать на подобных микропроцессорах, построены с использованием морально устаревших концепций, что снижает перспективность создания новых механизмов под такие ОС;

– наконец, имеет место сочетание двух факторов – бурный рост интереса к распределённым системам – тенденция всего лишь последних нескольких лет, а концепция DSM хотя и существует достаточно давно, но долгое время оставалась неизвестна широкому кругу разработчиков в связи с малой распространённостью распределённых решений.

Наконец, необходимо отметить ещё одну особенность DSM. Параллельные вычисления, устойчивость к сбоям отдельных узлов, поддержка динамических сетей, горячее резервирование оборудования и прочее – это механизмы, традиционно считающиеся крайне сложными и дорогими в реализации. В связи с этим разработка подобных механизмов оказывается обоснована только для систем с повышенными требованиями к производительности и надёжности (например, крупные промышленные, медицинские или военные решения). DSM предоставляет возможность существенно упростить решение подобных задач за счет перехода на более высокий уровень абстракции, не требуя от разработчика «ручного» управления отдельными сообщениями, курсирующими в системе.

Таким образом, в настоящий момент технология DSM оказывается востребованной в бурно развивающейся области MAC для IoT, позволяя существенно упростить создание конечных распределённых решений, привнести в них сложный функционал «больших систем», сократив при этом время на разработку.

[1]Проприетарный (англ. proprietary) – частный, несвободный. Здесь имеется в виду коммерческое, несвободное программное обеспечение.

Степень разработанности темы исследования. Масштабные работы в области теории МАС были проведены в разные годы В. Б. Тарасовым, Y. Shoham, G. Weiss, M. Wooldridge и др. Многопроцессорные вычислительные системы исследовались, в частности, А. М. Андреевым, Г. П. Можаровым, В. В. Сюзевым.

Родоначальником DSM систем принято считать K. Li, защитившего диссертацию на данную тему в 1986 году. В дальнейшем множество ученых исследовали существующие и предлагали новые модели консистентности, балансируя между производительностью и предсказуемостью. Наиболее существенные результаты были получены такими учеными как N. Bershad, M. Dubois, K. Gharachorloo, J. R. Goodman, P. W. Hutto, P. Keleher, L. Lamport, R. J. Lipton и др. Разработкой и анализом алгоритмов занимались A. Forin, R. E. Kessler, O. Krieger, M. Livny, M. Stumm, S. Zhou и др. Конечные DSM системы создавали H. E. Bal, J. K. Bennett, B. N. Bershad, B. Fleisch, D. Gelernter, E. Jul, P. J. Keleher, K. Li, M. Stumm, L. Zeng, S. Zhou и др.

В последние годы направление исследований смещается от разработки принципиально новых моделей консистентности и универсальных алгоритмов к созданию узко-специализированных DSM-систем, наиболее полно учитывающих особенности той или иной предметной области (к примеру, можно обратиться к статьям L. Zeng 2017 года). Однако сфера IoT при высоком уровне востребованности оказалась до сих пор исследователями не охвачена.

Объект и предмет исследования. В данной работе в качестве объекта исследования выступают модели, методы, алгоритмы, языки и программные инструменты для организации процесса обмена «знаниями» в мультиагентных системах. Предметом исследования является мультиагентная координация посредством механизма распределённой общей памяти в условиях беспроводной связи в анизотропном радиоэфире.

Цели и задачи диссертационной работы. С учетом проведенного анализа, целью данной работы является разработка моделей, алгоритмов и про-

граммных средств, реализующих концепцию распределённой общей памяти для мультиагентных систем в IoT и позволяющих существенно упростить и ускорить создание прикладных решений в данной области. В соответствии с поставленной целью в работе решаются задачи по разработке (созданию) следующих компонент.

1. Модели консистентности данных в распределённой системе, отвечающей требованиям и особенностям мультиагентных систем в области IoT.
2. Алгоритма организации узлов мультиагентной системы в само-восстанавливающуюся структуру, устойчивую к выходу из строя отдельных узлов.
3. Программного интерфейса для прикладного взаимодействия с разрабатываемым механизмом реализации концепции распределённой памяти.
4. Алгоритмического и программного обеспечения, реализующего концепцию распределённой общей памяти для мультиагентных систем в сфере IoT.
5. Экспериментального программно-аппаратного стенда (включая сбор характеристик разработанного программного решения).

Научная новизна данного диссертационного исследования заключается в следующем.

1. Усиленная модель консистентности по выходу (англ. enhanced release consistency) дополняет возможности известной ранее модели по выходу отдельными свойствами модели по входу. Данное сочетание свойств предложено впервые и позволяет добиться лучших характеристик в заданной предметной области, чем любая из исходных моделей.
2. Алгоритм ролей и переходов для узлов MAC, в отличие от описанных ранее алгоритмов, учитывает высокую динамичность системы и обеспечивает её устойчивость к сбоям отдельных узлов.

3. Концепция и интерфейс прикладного взаимодействия с DSM системой упрощают её использование и перенос на альтернативные аппаратные платформы, а также обеспечивают более высокий уровень защиты от ошибок прикладного программиста по сравнению с предложенными ранее.

Теоретическая и практическая значимость. Проведенное исследование стимулирует развитие MAC в сфере IoT. Разработанные методы и алгоритмы позволяют упростить создание прикладных решений в данной области. Результаты, изложенные в диссертации, могут быть использованы для разработки новых MAC систем, а также способов организации мультиагентного взаимодействия. При этом, впервые DSM система реализована для маломощных устройств без MMU, что расширяет возможности их применения.

Представленные механизмы позволяют снизить «порог вхождения» в область создания ПО для MAC, предоставляя прикладным разработчикам простой способ координации множества устройств, снижая тем самым стоимость новых разработок в данной области.

Основные научные результаты диссертационной работы внедрены в коммерческий продукт ОСРВ МАКС (операционная система реального времени для мультиагентных когерентных систем) и, вместе с ОС, используются в серийно производящемся оборудовании АО «ПКК Миландр»[1].

Методология и методы исследования. Методология исследования характерна для области предметной инженерии и заключается в идентификации и анализе проблемы, формулировании цели и задач, анализа состояния исследований и существующей литературы по вопросу, проектировании решения, выборе средств и технологий, реализации, проведении экспериментов и апробации.

В качестве методов используются перечисленные ниже.

– эмпирический метод (анализ литературы);

[1]Один из ведущих российских разработчиков интегральных микросхем.

– методы сравнения, обобщения, причинно-следственный (анализ существующих решений);

– метод индукции (формирование теоретического решения);

– методы объектно-ориентированного программирования (программная реализация);

– моделирование и эксперимент (анализ результатов реализации).

Кроме того, системный, причинно-следственный и сравнительный виды анализа были применены для получения практически всех основных научных результатов.

Положения, выносимые на защиту.

1. Модель консистентности, позволяющая добиться лучших характеристик в заданной предметной области по сравнению с моделями, взятыми за основу.

2. Алгоритм ролей и переходов для узлов MAC, обеспечивающий устойчивость системы к сбоям отдельных узлов.

3. Концепция и интерфейс прикладного взаимодействия с DSM системой, упрощающие её использование и перенос на альтернативные аппаратные платформы, а также обеспечивающие более высокий уровень защиты от ошибок прикладного программиста.

4. Модель, алгоритм и концепция воплощены в программном решении МАКС DSM, произведены измерения характеристик решения на специально созданном оборудовании и программной имитационной модели.

Степень достоверности и апробация результатов. Достоверность результатов работы обеспечивается анализом состояния исследований в данной области, докладами и публикациями по основным результатам, проведенными экспериментами и успешным внедрением.

Основные результаты диссертационного исследования докладывались и обсуждались на следующих мероприятиях.

– X Всероссийской межведомственной научной конференции «Актуальные направления развития систем охраны, специальной связи и информации для нужд органов государственной власти Российской Федерации» проводимой Академией Федеральной службы охраны Российской Федерации 7-8 февраля 2017 года в г. Орёл;

– IV Научно–практической конференции OS DAY «Операционная система как платформа», проводимой Институтом системного программирования РАН (ИСП РАН), 23-24 мая 2017 года в г. Москва;

– Семинаре «Актуальные проблемы создания бортовой системы навигации и навигационно-гидрографического обеспечения морских робототехнических комплексов (МРТК)», проводимом АО «ГНИНГИ» под руководством ФГБУ «ГНИИЦ РТ» 27 октября 2017 года в г. Санкт-Петербург.

Результаты исследования в виде программной реализации разработанных механизмов внедрены и являются существенной частью российской операционной системы реального времени МАКС.

Механизм распределённой общей памяти, интегрированный в ОСРВ МАКС, был внедрён [8] и демонстрировался в работе на серийно выпускаемой АО «ПКК Миландр» продукции на 20-й Международной выставке электронных компонентов, модулей и комплектующих «ЭкспоЭлектроника» 25-27 апреля 2017 года.

Публикации. По основным теоретическим и практическим результатам диссертации лично автором опубликовано 5 статей [2–6] в журналах из перечня, рекомендованного ВАК Минобрнауки России для публикации результатов диссертационных исследований.

Также автор является обладателем Свидетельства о государственной регистрации программы для ЭВМ № 2016617143 на ОСРВ МАКС (операционная система для мультиагентных когерентных систем) от 28 июня 2016 г., выданного Федеральной службой по интеллектуальной собственности [7].

Личный вклад автора. Все основные научные положения, выводы и рекомендации, составляющие содержание диссертационного исследования, получены автором лично.

Структура и объем диссертации. Диссертация состоит из введения, основной части (содержащей 3 главы), заключения, списка сокращений и условных обозначений, списка литературы, списка иллюстративного материала, списка таблиц и приложения. Общий объем диссертации – 133 стр., работа содержит 30 рис. и 4 табл. Список литературы включает 53 наименования на 7 страницах.

Глава 1. Исследования и терминология предметной области

Как было рассмотрено выше, обеспечение когерентности данных — одна из основных задач в MAC. Вместе с тем данная задача имеет концептуальные аналоги также и в других областях научного поиска. Выделение направления MAC, с одной стороны, позволило точнее очертить круг исследуемых вопросов, характерных именно для данного направления, с другой же – первоисточники отдельных проблем нового направления стали менее очевидны, что осложняет использование существующих решений или подходов, развитых в смежных направлениях. Ситуацию зачастую также осложняет отсутствие единой терминологической базы у исследователей разных направлений, поэтому одна и та же задача может встречаться в разных публикациях под различными «именами». С целью введения в проблемную область и знакомства с используемой разнообразной терминологией в данной главе рассмотрим ряд публикаций посвящённых исследуемой теме и выделим наиболее употребительные термины, часть из которых будем использовать в дальнейшем[1].

1.1. Концепции доски объявлений и DSM в MAC

Одна из базовых концепций координации агентов в теории MAC – «**доска объявлений**» (англ. **blackboard**) [21], описывающая систему взаимодействия нескольких «источников знаний» посредством некоторой общей структуры данных (рис. 1.1). В качестве иллюстрации часто предлагают представить группу специалистов, расположившихся рядом с доской, и совместно решающих некоторую задачу, обмениваясь промежуточными результатами посредством заметок на этой доске и наблюдая, что на ней пишут другие участники.

[1]Далее выделено заведомо больше терминов, чем используется в работе, однако каждый из выделенных терминов может представлять интерес с точки зрения развития данного исследования в будущем.

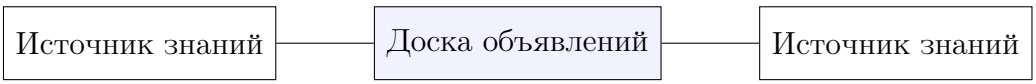


Рисунок 1.1 – Концепция доски объявлений

Данная концепция была весьма популярна на заре развития МАС, но постепенно исследовательская активность по этой теме снизилась [52, с. 398]. Возможно, дело в том, что за красивой метафорой по сути не было концептуальной новизны, и даже там, где термин «доска объявлений» использовался, наравне с ним часто можно было встретить альтернативный термин «общие данные» (англ. shared data) [52, с. 397] или «**общая память**» (англ. **shared memory**) [38, с. 82], используемый в литературе существенно чаще.

Так как МАС, по сути, являются децентрализованными системами, расширим исходную концепцию, заменив центральное хранилище данных децентрализованным — то есть распределённым. В обновленном виде концепцию можно назвать «распределённая доска объявлений» или «**распределённая общая память**» (англ. **distributed shared memory** – **DSM**). Последний термин точно соответствует предмету настоящего исследования и сопровождается множеством публикаций и по сегодняшний день.

1.2. Возникновение концепции DSM

Термин DSM существовал еще до выделения МАС в отдельное научное направление и, по всей видимости, зародился в процессе развития многопроцессорных вычислительных систем, относясь изначально к аппаратному обеспечению. В некоторых источниках можно встретить утверждение, что аппаратное обеспечение зачастую намного опережает программное [9, с. 590]. Обычно имеется в виду ситуация, когда аппаратные решения создаются быстрее, чем эффективные методики их программного использования. Однако возможна ситуация, когда проблемы, встающие перед ПО, когда-то уже были решены в той или иной форме разработчиками аппаратуры. По отношению к исследуемому

нами вопросу мы, по всей видемости, имеем дело именно с такой ситуацией.

Обзор архитектур многопроцессорных вычислительных систем можно найти, например, в книге [9]. Наиболее яркие представители – мультипроцессоры, мультикомпьютеры и распределённые системы. Кратко резюмируем соответствующее развитие вычислительных систем, выявив предпосылки возникновения DSM.

Мультипроцессоры — системы с несколькими процессорами, имеющими **общую память** (англ. **shared-memory** multiprocessor, рис. 1.2). Как мы видим, понятие «общая память» присутствует уже здесь, а сравнение рис. 1.1 и 1.2 очевидно показывает идентичность концепций. Память является в данном случае общей на физическом уровне (одни и те же микросхемы памяти доступны нескольким центральным процессорам). Проблемы синхронизации уже имеются, так как несколько процессоров одновременно работают с общим ресурсом, но синхронизировать требуется только доступ к памяти (так как память имеется в единственном экземпляре, проблема синхронизации содержимого не возникает).

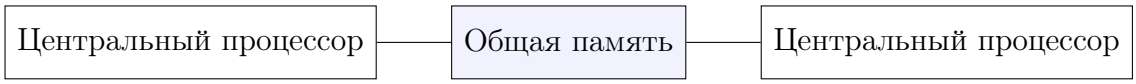
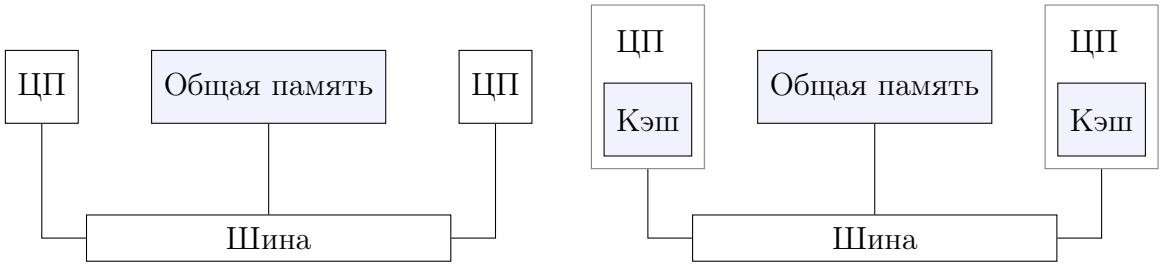


Рисунок 1.2 – Концепция мультипроцессорной системы с общей памятью

В простейшем случае проблема с синхронизацией доступа решается использованием общей шины работы с памятью (рис. 1.3, *а*), что предотвращает саму возможность одновременно выполняемых операций с последней. Однако с возрастанием количества центральных процессоров (ЦП), общая шина быстро становится узким местом системы, для предотвращения чего к каждому центральному процессору может быть добавлена собственная кэш-память (рис. 1.3, *б*). С одной стороны, это позволяет реализовать многие операции чтения данных без задействования общей шины, локально, но, с другой, возникает проблема обеспечения когерентности кэшей. Процедуры, обеспечивающие эту когерентность, называются **протоколом поддержки когерентности кэшей**

(англ. **cache-coherence protocol**). Несмотря на то, что проблема решается на аппаратном уровне, концептуально она очень близка исследуемой нами проблеме.



(*а*) Элементарный вариант (*б*) Вариант с кэшем

Рисунок 1.3 – Система с общей памятью и шиной данных

С ростом количества ядер резко возрастает сложность аппаратуры и требования к ресурсам процессора для обеспечения согласованности кэшей. Существуют опасения, что барьер применения данной технологии — несколько сотен процессоров. Проблема имеет собственное название **барьера согласованности** (англ. **coherency wall**). Один из вариантов решения — отказ от общей памяти вообще и переход к концепции локальной, а значит **распределённой памяти** (англ. **distributed memory**), в которой взаимодействие между процессорами организуется через механизмы передачи сообщений по шине данных (рис. 1.4). В отличие от мультипроцессоров, которые предлагают простую модель взаимодействия множества центральных процессоров посредством общей памяти, здесь мы имеем так называемые тесно связанные процессоры, общей памяти не имеющие — такие системы называются уже мультикомпьютерами (а также кластерными компьютерами, англ. cluster computers и COWS — Clusters of Workstations — кластерами рабочих станций).

Мультикомпьютерные системы оказались не только гораздо привлекательнее в плане масштабируемости, но и проще в создании, чем мультипроцессорные (хотя два этих термина часто используются как синонимы и иногда сложно понять, какого типа система имеется в виду). Однако сложность программирования таких систем существенно возросла. Действительно, если в мультипроцессо-

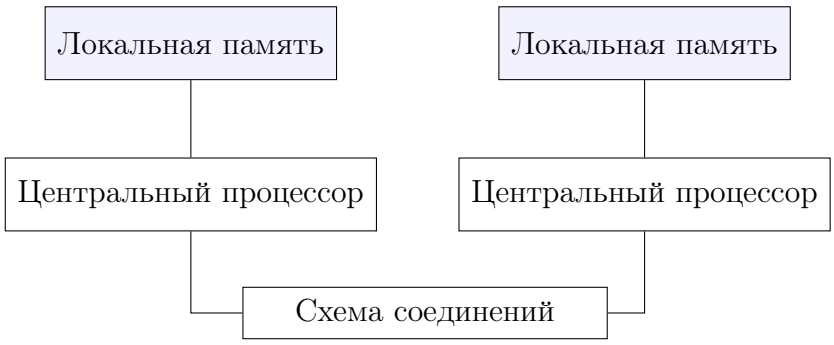


Рисунок 1.4 – Концепция мультикомпьютерной системы с распределённой памятью

рах с общей памятью, а тем более с локальным кэшем, вопросы синхронизации и обеспечения когерентности решались аппаратно, то теперь они полностью перекладывались на программистов. Традиционно, коммуникации в многопроцессорных системах без общей памяти реализуются в модели, подразумевающей передачу данных. Это означает, что если программисту необходимо наладить какое-либо взаимодействие между процессорами, он вынужден использовать такие примитивы как классические Send и Receive. В некоторых случаях эти примитивы могут быть представлены чем-то более высокоуровневым – например, как в модели удаленного вызова процедур (англ. remote procedure call – RPC). Тем не менее в каждом из случаев мы имеем дело с механизмами явной передачи данных.

Программные решения, основанные на передаче сообщений (данных) между компьютерами, оказались чрезвычайно сложны, и программисты стали создавать новые коммуникационные абстракции. Закономерно, одной из таких абстракций стала концепция **общей памяти** (англ. **shared memory**), также называемой совместно используемой памятью, с которой разработчики уже были знакомы по мультипроцессорам. Даже в ситуации, когда общая память физически отсутствует, оказалось возможным создать программную прослойку, успешно такую память имитирующую.

Дальнейшим логичным шагом по повышению масштабируемости вычислительных систем стал переход от тесно- или сильносвязанных (англ. tightly coupled) к **слабо- или гибкосвязанным системам** (англ.

loosely coupled), также известным как **распределённые системы** (англ. **distributed systems**). Концепция общей памяти оказалась применима и к ним, закономерно получив название **распределённой общей памяти** (англ. **distributed shared memory – DSM**) [44].

Отметим, что несмотря на выделение DSM в независимую задачу в области распределённых систем, имеется существенное пересечение в назначении данного механизма с аналогичными механизмами ряда других технологий: как уже было отмечено, это организация работы процессорных кэшей в мультипроцессорах с физически общей памятью, а также кэширующие распределённые файловые системы, распределённые базы данных, NUMA-мультипроцессоры с общей памятью неоднородного доступа (где различается время доступа к разным областям памяти) и других. В каждой области имеется своя существенная специфика, тем не менее, общие принципы остаются едины и решения в одной области зачастую могут быть применены (по крайней мере частично) в другой.

1.3. Описание концепции DSM

Итак, DSM – это концепция совместно используемой (или общей) памяти, примененная к распределённым (слабо связанным) системам.

В отличие от модели явной передачи данных (сообщений), модель общей памяти предоставляет процессорам системы возможность работы в едином адресном пространстве. Кроме прочего, это позволяет реализовывать распределённые приложения тем же способом, что и классические, централизованные, а также относительно легко портировать уже существующее ПО, приспосабливая его для работы в распределённом окружении. По сравнению с моделью явной передачи данных, на смену операциям Send и Receive выходят операции Read и Write:

```
data = Read( address )
Write( address, data )
```

Операция Read возвращает данные, расположенные по адресу address, указанному параметром. Write же имеет два параметра, записывая по адресу address данные data. Возможны различные реализации данных примитивов – под другими именами, некоторыми изменениями в параметрах, однако с сохранением общих принципов.

Таким образом, базируясь на классических Send и Receive, новая абстракция скрывает всю сложность низкоуровневого взаимодействия узлов распределённой сети под более высокоуровневыми интерфейсами специальных системных библиотек или ядра операционной системы.

Впервые модель коммуникаций DSM была предложена в диссертации 1986 года [32] (автор использовал теперь редко употребляющееся наименование «shared virtual memory»). Организация виртуальной памяти была страничной, и когда происходило обращение к странице, отсутствующей в данный момент на конкретном узле системы, производилась пересылка этой страницы с другого узла сети. Новая система отлично зарекомендовала себя, обеспечивая простое программирование распределённых систем без общей физической памяти. Платой же стала относительно низкая производительность (по сравнению с механизмами простой пересылки сообщений). Интенсивные исследования последующих лет позволили нивелировать этот недостаток, предложив ряд менее строгих моделей консистентности (что обычно означает внедрение новых ограничений и допущений, а также требований к способам использования), старясь сохранить баланс между удобством для разработчика (основной причиной появления DSM) и производительностью. Наиболее известные модели будут рассмотрены в следующем разделе, заметим лишь что исследования в этой области до сих пор продолжаются, сосредоточившись в целом на нюансах той или иной модели, эффективности её реализации и привнесении в вычислительную систему новых свойств – например таких, как устойчивость к сбоям.

1.4. Модели консистентности

Рассмотрим сценарий, аналогичный разобранному в работе [32] – имеется распределённая вычислительная система из рабочих станций (узлов), объединенных в сеть. Виртуальная память организована странично, и каждая страница имеет единственного владельца. Допустим, один из узлов системы производит обращение к данным, расположенным на странице, которая отсутствует локально. В таком случае системой производится пересылка необходимой страницы с удаленного узла. Запросивший страницу узел становится её новым владельцем. В случае, когда разные узлы часто обращаются к данным, расположенным на данной странице, система вынуждена часто пересылать страницу между узлами, создавая высокую нагрузку на каналы передачи данных и снижая общую производительность системы.

Улучшить ситуацию с нагрузкой на каналы можно, в том числе, допустив наличие множества копий одной и той же страницы. В таком случае пересылки данных во многих случаях удастся избежать, однако возникает другая проблема – необходимо обеспечить консистентность (согласованность) всех копий одной и той же страницы. В идеале, обращение любого узла к своей копии любой страницы должно приводить ровно к тому же эффекту, как и в случае, когда владелец у страницы один, и её единственная копия постоянно пересылается. На практике же оказывается, что для достижения данного результата всё ещё требуется большое количество служебных сообщений (например, чтобы убедиться, что имеющиеся у узла данные до сих пор актуальны), что в случае распределённых систем и, соответственно, медленных каналов связи, не позволяет добиться высокого уровня производительности.

В результате стремления снизить нагрузку на сеть и повысить производительность DSM, было предложено множество альтернативных моделей консистентности – менее строгих, чем интуитивно понятная эталонная (далее упоминаемая как строгая консистентность, англ. strict consistency – SC). В работе

[36] проводится тщательный обзор основных моделей консистентности, а также их влияния на язык программирования, компилятор и среду выполнения. Хотя в данной работе указывается на отсутствие общепринятой классификации моделей, уже год спустя, в 1994, выходит книга [46], предложившая простую, но вполне удобную классификацию, которой мы и будем придерживаться.

1.4.1. Строгая консистентность

Модель строгой консистентности – самая строгая модель, реализующая интуитивно ожидаемое поведение системы. Формальное определение звучит следующим образом:

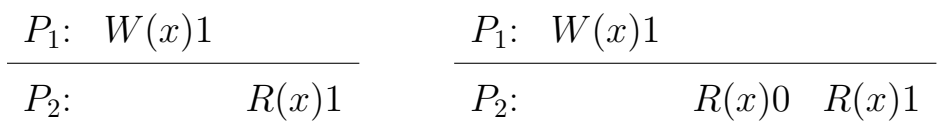
Определение 1. *Система соответствует модели строгой консистентности в том случае, если чтение из памяти по адресу x всегда возвращает значение, сохраненное последней предшествующей операцией записи по данному адресу.*[1]

В однопроцессорных системах данная модель справедлива по умолчанию, поэтому может представляться единственно возможной. Однако несмотря на кажущуюся простоту и очевидность, в случае распределённых систем модель оказывается нереализуемой на практике (часто её даже не упоминают), так как разница во времени между операциями записи и чтения, если они производятся в разных узлах, может быть бесконечно малой, что препятствует любым попыткам добиться эталонной синхронизации этих операций.

При объяснении и сравнении различных моделей часто используют определенную нотацию (см. работы [36, 46], а также работу [43] с более детальной разновидностью той же нотации), подходящую и в данном случае. Пусть P_x – процесс номер x. При этом подразумевается, что каждый процесс выполняется на отдельном узле. Операции чтения и записи будем обозначать соответственно R и W (от англ. read и write). Тройка $op(var)val$ будет обозначать операцию

[1] «Any read to a memory location x returns the value stored by the most recent write operation to x». [46]

над памятью, где op – вид операции (R или W), var – обозначение адресуемого объекта (например, переменной), а val – значение, являющееся входным для операции записи или же выходным в случае операции чтения. Примем начальное значение всех переменных за 0. Расположим процессы системы по вертикали, а шкалу времени по горизонтали (рис. 1.5).



($а$) Корректный вариант ($б$) Некорректный вариант

Рисунок 1.5 – Модель строгой консистентности

Слева (рис. 1.5, $а$) мы видим ситуацию, допустимую в модели строгой консистентности: первый процесс записывает значение 1 в переменную x, позже второй процесс считывает из переменной x это значение. Однако справа (рис. 1.5, $б$) ситуация уже иная – несмотря на то, что первый процесс уже записал в переменную x значение 1, второй процесс сначала считывает из нее значение 0, и лишь затем, выполняя еще одну операцию чтения вдруг обнаруживает значение 1. Описанная ситуация на первый взгляд может показаться абсурдной и не соответствующей строгой модели консистентности (см. определение 1 на стр. 24), однако далее будет показано, что в более слабых моделях такая ситуация оказывается допустимой. Более того, система может оставаться при этом вполне предсказуемой и удобной для программирования.

1.4.2. Последовательная консистентность

Хотя, как было показано выше, строгой консистентности практически невозможно достичь в распределённых системах, в параллельном программировании она не используется даже в случае многопоточного программирования на единственном ядре. Основной принцип корректного параллельного программирования состоит в том, что никакой поток не должен рассчитывать на какую-

либо определенную скорость выполнения другого потока. В случаях же, когда необходимо быть уверенным в выполнении того или иного фрагмента кода соседнего потока, используются специальные примитивы синхронизации (например, мьютексы). Таким образом, ослабление строгой модели консистентности до модели консистентности последовательной не приводит к возрастанию сложности программирования конечных решений. При этом данная модель – самая строгая из практически применимых в распределённых системах. Рассмотрим определение:

Определение 2. *Система соответствует модели последовательной консистентности в том случае, если результат любого выполнения системы всегда такой же, как если бы операции со всех процессоров системы выполнялись в некотором порядке, формируя общую последовательность операций, причем операции каждого отдельно взятого процессора должны встречаться в этой общей последовательности операций именно в том порядке, который определен программой этого процессора.*[1]

Из определения 2 следует, что операции могут выполняться в целом в любой последовательности и с любыми задержками, однако операции каждого из процессоров должны сохранять свою последовательность, а операции всех узлов с общей памятью должны быть видны всем узлам упорядоченными в одну и ту же последовательность. Хотя, от запуска к запуску, эта последовательность может выглядеть по-разному. На рисунке 1.6 представлено два варианта выполнения одной и той же программы, причем оба варианта соответствуют определению 2.

Заметим, что рис. 1.5, *б* и 1.6, *а* совпадают – поведение, неприемлемое для модели строгой консистентности оказывается допустимым в последователь-

[1] «The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program». [31]

P_1:	$W(x)1$		
P_2:		$R(x)0$	$R(x)1$

(*а*) Вариант 1

P_1:	$W(x)1$		
P_2:		$R(x)1$	$R(x)1$

(*б*) Вариант 2

Рисунок 1.6 – Модель последовательной консистентности

ной модели. Не важно, когда «реально» произошла операция записи на процессоре P_1 – процессор P_2 может «увидеть» её как до своих операций чтения (рис. 1.6, *б*), так и в любой другой момент времени, например, между двух своих операций чтения (рис. 1.6, *а*). Главное, что все узлы в системе должны иметь единое видение последовательности этих операций – таким образом, недопустима ситуация, когда один узел воспринимает происходящее как на рисунке 1.6, *а*, а другой – как на рисунке 1.6, *б*. Такая недопустимая ситуация изображена на рисунке 1.7.

P_1:	$W(x)1$		
P_2:		$R(x)0$	$R(x)1$
P_3:		$R(x)1$	$R(x)1$

Рисунок 1.7 – Ситуация, недопустимая в модели последовательной консистентности

Далее обзорно ознакомимся с другими «глобальными» моделями и перейдем к моделям, основанным на синхронизирующих операциях, позволяющим заметно повысить производительность, поддерживая консистентность только на определенных участках выполняющейся программы.

1.4.3. Другие глобальные модели

Так как ослабление модели (введение в нее новых ограничений и допущений) позволяет добиться более высокой производительности, было создано множество альтернативных моделей, применимых в каких-либо частных случаях. Наиболее известны следующие модели:

– Причинная

– PRAM

– Процессорная

Причинная модель акцентирует наше внимание на том, что не имеет смысла контролировать порядок всех обращений к общей памяти, важна лишь последовательность операций, потенциально влияющих друг на друга. Например, если параллельно производится запись в переменные x и y, последовательность этих операций важна лишь в том случае, если значения переменных взаимосвязаны (к примеру, y вычисляется через x). Однако контролировать причинность, в общем случае, отдельная сложная задача, и модель применяется редко.

Модель PRAM гарантирует соблюдение лишь последовательности операций на каждом процессоре, при этом взаимное расположение операций разных процессоров на каждом узле может видеться по-разному. Такую модель легко реализовать, но разрабатывать конечное ПО в такой модели – задача нетривиальная.

Процессорная модель добавляет к условиям PRAM модели еще одно – порядок операций записи в общую память должен все-же видеться на всех узлах одинаково.

Тем не менее, модель последовательной консистентности остается самой «понятной» и использовалась наиболее широко вплоть до появления нового подхода к ослаблению модели, основанного на операциях синхронизации. Данный подход породил группу моделей, описанных ниже, и внимание современных исследователей сместилось к ним. Данные модели необходимо рассмотреть более внимательно, так как именно на них основано новое решение, выработанное в исследовании, которому посвящена данная диссертация.

1.4.4. Слабая консистентность

Появление данной модели стало возможным благодаря наблюдению, что конечным приложениям обычно «не интересна» большая часть операций с общей памятью, происходящих на других узлах системы. Это похоже на то, как два параллельно выполняющихся потока лишь иногда «интересуются» состоянием друг друга. Соответственно, было найдено решение – определить в конечном ПО так называемые критические секции, где и гарантировать консистентность. В остальных же участках кода такой гарантии не будет, как не будет и накладных расходов на пересылку данных между узлами. Строгое определение было сформулировано следующим образом:

Определение 3. [1] *Система соответствует модели слабой консистентности, если выполняются следующие условия:*

1. *Доступ к синхронизационным переменным осуществляется в модели последовательной консистентности.*
2. *Доступ к синхронизационной переменной выполняется только после окончания операций записи в общую память во всех других узлах.*
3. *Доступ к общей памяти выполняется только после завершения текущих операций с синхронизационной переменной.*

Таким образом, через обращение к глобальной (распределённой) синхронизационной переменной можно обозначить начало и конец критической секции, входя в которую, процесс должен быть уверен в актуальности распределённых

[1] Основано на определении из работы [20]:

1. accesses to global synchronizing variables are strongly ordered and if
2. no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and if
3. no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

данных, и, в свою очередь, гарантируя актуальность распределённых данных после своей работы с ними по выходу из критической секции. Состояние общей памяти вне критической секции не определено. При этом только один процесс одновременно может находиться в критической секции.

За счет кардинального сокращения обмена данными (теперь он стал требоваться только при входе в критическую секцию и выходе из нее), модель оказалась гораздо эффективнее модели последовательной консистентности. Тем не менее, новую модель всё-таки можно рассматривать именно как модель последовательной консистентности, лишь с одним дополнением: модель выдерживается только в пределах критических секций.

1.4.5. Консистентность по выходу

Слабая консистентность была существенным шагом в задаче повышения производительности, однако не разделяла ситуации «вход в критическую секцию» и «выход из секции», требуя в любом случае завершения всех активных операций с общей памятью (распространения информации о них по всем узлам вычислительной системы). Модель консистентности по выходу усовершенствовала слабую модель, разграничив эти две ситуации, соответствующие операции над синхронизационной переменной были названы «захват» и «освобождение» (англ. acquire и release соответственно). Впервые модель была предложена в работе [35], и определена следующими правилами:

Определение 4. [1] *Система соответствует модели консистентности по выходу, если выполняются следующие условия:*

[1] Основано на определении из работы [35]:

1. before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and
2. before a *release* access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and
3. *special accesses* are processor consistent with respect to one another.

1. *Перед осуществлением доступа к общей памяти все предшествующие операции данного процессора по захвату синхронизационной переменной должны быть завершены.*
2. *Операция освобождения синхронизационной переменной должна быть выполнена только после того, как все предшествующие операции данного процессора над общей памятью будут завершены.*
3. *Операции захвата и освобождения синхронизационной переменной должны подчиняться модели процессорной консистентности.*

В работе [35] показано, что требования процессорной консистентности в данном случае достаточно, при этом операции внутри критической секции можно считать упорядоченными согласно модели последовательной консистентности.

При этом, операция захвата не гарантирует того, что локально произведенные операции над общей памятью будут немедленно распространены по всем узлам сети (лишь гарантирует то, что операции других узлов в данный момент будут обнаружены). А опреация освобождения также не гарантирует моментального получения информации об операциях, произведенных на других машинах (зато обеспечивает распространение информации о произведенных внутри критической секции операциях).

Интересно, что данная модель позволяет «защищать» критической секцией не всю общую память, а только ту её часть, с которой производится работа в секции (таким образом, критических секций может быть много). Соответственно, нет необходимости пересылать между узлами все изменения – достаточно изменений, затрагивающих конкретную критическую секцию. Кроме того, различные узлы получают возможность одновременного нахождения в критических секциях, что еще более повышает производительность системы за счет минимизации ожиданий – требуется лишь чтобы в этих секциях работа велась с непересекающимися данными распределённой памяти.

За счет привнесения дополнительной информации о ходе исполнения программы (вход или выход из критической секции), реализация модели получается более эффективной, чем для предыдущей модели. Однако эту дополнительную информацию должен предоставить разработчик, явным образом помечая начало и конец критических секций – либо через операции захвата и освобождения одной и той же синхронизационной переменной, либо через выполнение одной и той же операции над двумя разными, специально выделенными синхро-переменными.

1.4.6. Ленивая консистентность по выходу

Модель ленивой консистентности по выходу (англ. lazy release consistency) была предложена в работе [30]. В отличие от изначальной модели консистентности по выходу (которую, во избежание путаницы, стали часто называть моделью активной консистентности по выходу, англ. eager release consistency), новая модель не требовала немедленной рассылки информации об обновленных данных по выходу из критической секции. Лишь узлы, входящие в критическую секцию (иными словами, осуществляющие захват синхронизационной переменной), гарантированно получали эти данные в момент захвата.

Такой подход позволил еще более сократить обмен информацией между узлами – ведь если данные ни одному узлу сейчас «не нужны», то и пересылаться в новой модели они не будут.

1.4.7. Консистентность по входу

Модель консистентности по входу (англ. entry release consistency) потребовала [15], чтобы каждая переменная из распределённой памяти была закреплена за какой-либо синхронизационной переменной. Это позволило легко контролировать, какие именно данные нуждаются в обновлении при входе и выходе из критической секции. Модель консистентности по выходу при вхождении в кри-

тическую секцию была вынуждена «догадываться» о том, какие данные необходимо обновить. Возможности модели по выходу были таким образом усилены, количество передаваемых между узлами данных уменьшилось, а возможности по параллелизму возросли (за счет возросшей способности разных узлов одновременно находиться в разных критических секциях). Также было введено понятие эксклюзивной и неэксклюзивной блокировки синхронизационной переменной, что позволило отделить операции чтения общих переменных от операций записи. Очевидно, что в случае отсутствия эксклюзивной блокировки, в неэксклюзивной секции может находиться несколько узлов одновременно. Формулировка правил модели по входу выглядит так:

Определение 5. [1] *Система соответствует модели консистентности по входу в том случае, если выполняются следующие условия:*

1. *Операция захвата синхронизационной переменной может быть завершена только после того, как все операции записи других узлов в защищаемые ей переменные общей памяти также будут завершены.*
2. *Операция эксклюзивного захвата синхронизационной переменной может быть завершена только после того, как все узлы освободят данную переменную (даже если она была захвачена неэксклюзивно).*
3. *По завершении эксклюзивного доступа к синхронизационной переменной другой узел может захватить переменную неэксклюзивно, но такая*

[1] Основано на определении из работы [15]:

1. An *acquire* access of s is not allowed to perform with respect to processor p_i until all updates to D_s have been performed with respect to p_i. An update to a memory location is said to *perform with respect to processor* p_i at a point in time when a subsequent read of that location by p_i returns the value written by the update.
2. Before an exclusive mode access to a synchronization variable s by processor p_i is allowed to perform with respect to p_i, no other processor may hold s in non-exclusive mode.
3. After an exclusive mode access to s has been performed, any processor's next non-exclusive mode access to s may not be performed until it is performed with respect to the owner of s.

операция захвата может быть завершена только после того, как будут учтены результаты операций над соответствующими общими переменными владельца этой синхронизационной переменной.

Вместе с разделением захвата синхронизационной переменной на эксклюзивный и неэксклюзивный, было введено понятие владельца этой переменной. Владелец – тот, кто последним осуществлял её захват. Процесс, являющийся в данный момент владельцем переменной, не обязан осуществлять какой-либо информационный обмен с другими узлами сети при многократном входе и выходе из соответствующей критической секции до тех пор, пока другой узел не попробует осуществить захват этой переменной – в таком случае информация о новых значениях соответствующих общих переменных будет передана на запрашивающий узел.

1.4.8. Заключение

Исследования в данной области продолжаются: создаются новые модели (например, view-based consistency model [49]), обобщаются все существующие и выводятся общие закономерности [43], а некоторые исследователи считают, что мы дошли до рубежа, за которым нужно подумать о принципиально новых аппаратных и программных идеологиях [11]. Тем временем, описанные выше модели активно используются, адаптируясь к тому или иному программно-аппаратному окружению и воплощаясь в различных реализациях. Краткий обзор этих реализаций мы рассмотрим ниже. Но сначала необходимо рассмотреть алгоритмы, лежащие в основе – принципиальные подходы к реализации моделей консистентности.

1.5. Алгоритмы

В основе любого DSM механизма лежит некоторая модель консистентности, основные модели были разобраны в разделе 1.4. Однако одна и та же

модель может быть реализована множеством разнообразных способов, сильно отличающихся между собой по критериям производительности, надежности и другим. В данном разделе рассмотрим основные известные алгоритмы с целью выбора или формирования собственного в последующих главах. Алгоритмы детально рассмотрены в работах [44] и [45] (в текущей работе использованы минимально адаптированные изображения из них), причем в первой работе рассматриваются их базовые версии, а во второй – возможные усовершенствования с целью обеспечения устойчивости к отказам отдельных узлов. Однако авторы данных работ не разделяли понятия модели консистентности и алгоритмов реализации, представляя вторые самодостаточно, не выделяя модель в качестве базового соглашения по правилам использования общей памяти между программистом и «оборудованием». Мы же рассматриваем именно модель консистентности в качестве основы DSM системы, а алгоритмы – как способ более или менее эффективной реализации конкретной модели. Такой подход позволяет оперировать в каждом случае только существенными для данного случая деталями, отделяя принципиальные требования и свойства модели от частных решений и алгоритмов, направленных на повышение эффективности в заданных моделью рамках.

1.5.1. Алгоритм с центральным сервером

Наиболее очевидное решение задачи управления распределёнными данными – выбрать один из узлов так называемым сервером, и производить все операции через него (рис. 1.8). Например, когда модель консистентности требует от некоторого узла (скажем, №2) обновить у себя те или иные общие данные – обращаться за этими данными следует именно к серверу (узел №1). И наоборот – если модель требует от узла (например, №2) распространить его изменения общих данных по всем остальным узлам – узлу нужно делегировать эту задачу серверу (№1), передав ему и постановку задачи, и данные для неё. Сервер же самостоятельно свяжется с остальными узлами (№3 и №4) и передаст им

нужную информацию.

Обособление в распределённой системе какого-то узла потенциально опасно, так как выделенный узел становится слабым местом системы – выход из строя лишь одного данного узла может привести к тому, что вся система выйдет из строя. Воспрепятствовать этому можно, например, через резервирование узла-сервера (рис. 1.9).

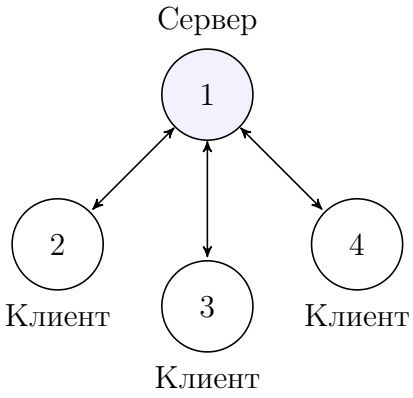


Рисунок 1.8 – Алгоритм с центральным сервером

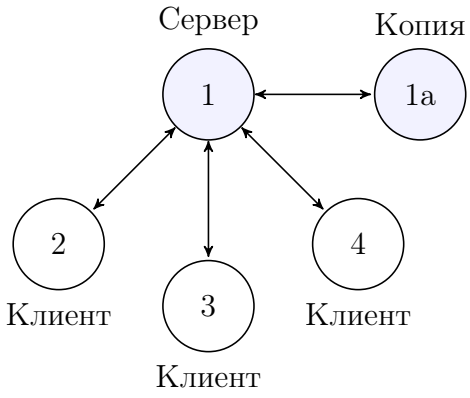


Рисунок 1.9 – Отказоустойчивый алгоритм с центральным сервером

В систему вводится новый узел (или данные функции возлагаются на один из уже существующих узлов), являющийся «зеркалом» узла-сервера. И теперь, при выполнении любой операции записи сервер (узел №1) сначала информирует о ней свою копию (узел №1a), и только получив от копии ответ, продолжает свою работу. Если выйдет из строя узел-копия, узел-сервер это заметит по срабатыванию таймаутов на операции, с которыми сервер обращается к копии, и выделит другой узел в качестве своей копии. Заметить выход из строя самого сервера несколько сложнее – для этого необходим ввод в систему дополнительного сервиса, регулярно опрашивающего сервер (корректное функционирование данного сервиса, в свою очередь, также нужно регулярно проверять), в частности, с этой задачей может справиться и узел-копия. В случае обнаружения выхода из строя сервера, потери данных не происходит, так как они имеются на узле-копии, который становится новым сервером, выде-

ляя какой-то из оставшихся узлов в качестве своей копии. Клиент, заметивший сработавший таймаут может разослать широковещательное сообщение, чтобы найти новый сервер, либо же немного подождать – новый сервер может сам послать широковещательное сообщение, информируя узлы в системе о смене узла-сервера.

Очевидно, что при одновременном выходе из строя и сервера и его копии, восстановиться системе уже не удастся, и здесь разработчику необходимо определиться – какую надежность должна обеспечивать его система. В случае необходимости у узла-копии может изначально иметься и собственная копия, что, соответственно, повысит надежность системы, но увеличит задержки при выполнении операций и нагрузку на каналы передачи данных.

Данный алгоритм применим ко многим моделям консистентности, так как не вносит новых ограничений на пересылку данных, а лишь требует, чтобы операции записи, когда бы они (согласно требованиям конкретной модели) ни распространялись между узлами, происходили не напрямую, а через выделенный серверный узел.

1.5.2. Алгоритм миграции данных

В данном алгоритме данные всегда хранятся лишь на одном узле. В случае, если они потребовались на каком-то другом – осуществляется их миграция (рис. 1.10).

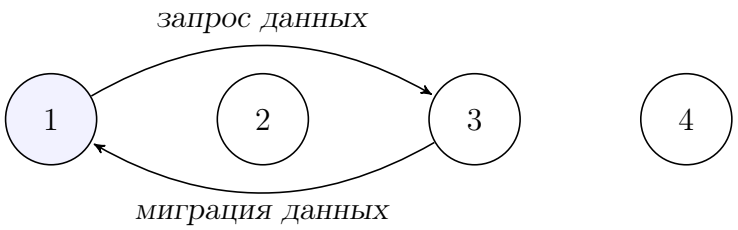


Рисунок 1.10 – Алгоритм миграции данных

При таком подходе, завладевший данными узел, может сколь угодно долго осуществлять операции чтения и записи без необходимости обмена данными

с другими узлами (по крайней мере до тех пор, пока какой-то другой узел не потребует те же данные). Текущий владелец может быть обнаружен через рассылку широковещательного запроса, либо каждый элемент данных может быть связан с конкретным узлом-менеджером (однозначно определяемым, например, по некоторой хэш-функции из свойств элемента), который будет контролировать перемещение данного блока и всегда «знать» его текущего владельца.

Устойчивость ко сбоям отдельных узлов привносится в алгоритм достаточно просто – нужно лишь обеспечить наличие копий у всех блоков данных, что проще всего сделать следующим образом: в момент, когда осуществляется трансфер данных от одного узла к другому, данные на первом узле не уничтожаются, а лишь помечаются как запасные (в англоязычных работах обычно используется слово invalid). Одновременно с запрошенными данными следует отправлять и остальные, которые были изменены на узле, но пока никому не отправлялись (так называемые dirty данные). При получении, такие данные маркируются как invalid, а в совокупности всё это приводит к наличию копии у любых данных, из чего достаточно просто можно создать алгоритм восстановления в случае сбоя любого узла, аналогично алгоритму, представленному в случае с центральным сервером.

Алгоритм миграции данных хорошо применим для реализации моделей, подобных модели ленивой консистентности по выходу, и совершенно неприменим в случаях, когда измененные данные нужно как можно скорее распространить по всем узлам, а также в случаях, когда требуется обеспечить возможность неэксклюзивного захвата синхро-переменной (в котором допускается одновременное чтение несколькими узлами одних и тех же данных). Отсюда вытекает и еще один минус алгоритма – высокая нагрузка на каналы передачи данных в случае если различные узлы часто обращаются к одним и тем же данным, даже если производят лишь операции чтения.

1.5.3. Алгоритм репликации по чтению

Данный алгоритм призван повысить эффективность алгоритма миграции, описанного выше. Он допускает создание множественных копий одних и тех же данных в случае если к этим данным осуществляется доступ только на чтение. Очевидно, что во многих случаях затрат на пересылку данных в этом алгоритме удастся избежать, однако операции записи становятся несколько более затратными – ведь теперь им нужно ещё и оповестить всех владельцев копий соответствующих данных о том, что эти данные больше не актуальны. За неимением лаконичного русскоязычного синонима английскому слову invalidate, назовем такое оповещение инвалидацией.

На рисунке 1.11 показано, как выглядит операция записи в общие данные в данном алгоритме: данные являются реплицированными на узлах №3 и №4; узел №1 собирается произвести в них запись, и выполняет соответствующий запрос к текущему владельцу №3 (в нашем случае к одному из владельцев, так как данные реплицированы); запрос выполняется и новым владельцем данных становится узел №1, который тут же оповещает прежних владельцев №3 и №4 о том, что их данные теперь считаются устаревшими.

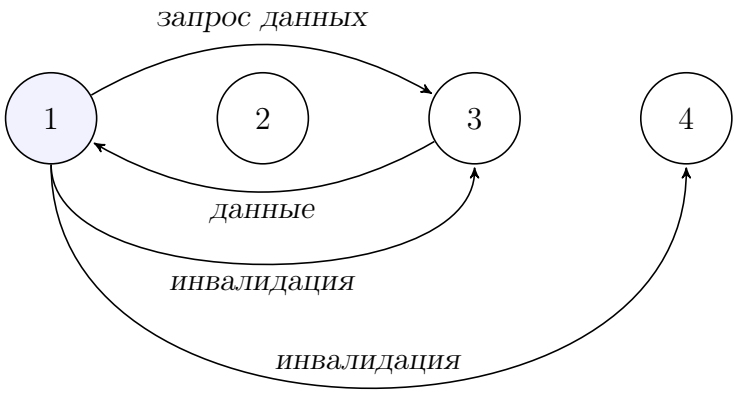


Рисунок 1.11 – Операция записи в алгоритме репликации по чтению

Устойчивость ко сбоям в данном алгоритме может быть достигнута таким же способом, как было описано в случае для алгоритма миграции выше.

1.5.4. Алгоритм полной репликации

По сравнению с алгоритмом репликации по чтению, алгоритм полной репликации идёт еще дальше, позволяя нескольким узлам одновременно не только читать, но и записывать одни и те же данные. Чтобы сохранять в таких условиях консистентность, все операции записи должны быть каким-то образом упорядочены. В мультипроцессорах упорядоченность достигается автоматически, при использовании шины данных, которая одновременно может заниматься передачей информации только об одной операции. В условиях отсутствия такой шины, среди узлов может быть выделен особый узел, осуществляющий необходимое упорядочивание. В англоязычных работах данный узел обычно называется sequencer, мы же назовём его просто «сервер».

Роль этого сервера похожа на роль одноименного узла, описанного в разделе 1.5.1, однако в отличие от последнего, в данном случае сервер не хранит все данные, а лишь гарантирует единственно верную для всех узлов последовательность операций записи. Для этого, как и раньше, любому узлу, желающему произвести операцию записи в общую память, необходимо сообщить об этом серверу. Задача же сервера – оповестить о данной операции все узлы, включая и узел-источник изменения, который по получении такого сообщения может быть уверен, что операция действительно произведена (см. 1.12).

Каждое своё оповещение сервер может снабжать номером, который возрастает с каждой рассылкой. Благодаря данному номеру узлы могут быть уверены, что получают оповещения в правильном порядке (а пропустив какое-то из них, имеют возможность запросить его снова, указав в запросе соответствующий номер).

Так как данные распределены по всем узлам, выход из строя любого из них оказывается обратим без дополнительных усовершенствований алгоритма. Сервер может быть восстановлен как и любой другой узел, так как всё, что ему нужно «знать», кроме состояния общей памяти, это последний использованный

его предшественником номер, а эта информация имеется по крайней мере в одном из узлов (нужно лишь выбрать максимальное значение из имеющихся). Для ускорения процесса восстановления, можно поддерживать копию сервера – так же как было описано в разделе 1.5.1.

Алгоритм может быть полезен не только для реализации полностью реплицированных данных, но и (частично), скажем, в ситуации, когда каналы передачи данных не могут обеспечить упорядоченную рассылку широковещательных сообщений (иными словами, не могут гарантировать, что сообщение, отправленное узлом раньше, придет ко всем остальным узлам тоже раньше, чем отправленное следующим).

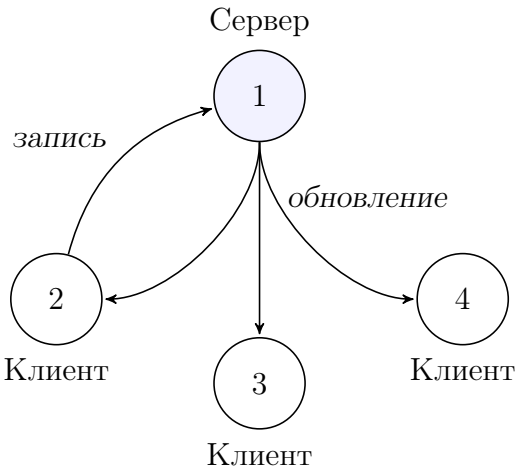


Рисунок 1.12 – Операция записи в алгоритме полной репликации

1.5.5. Заключение

Выше рассмотрены лишь основные подходы и принципы, на основе которых может быть создано бесконечное множество алгоритмов, уместных в том или ином окружении. Собственный алгоритм, применимый в нашей ситуации, будет выработан в разделах ниже.

1.6. Реализации

С момента возникновения концепции DSM было создано множество реализующих её систем. Так как автору не удалось найти подходящий их список (в обнаруженных работах рассматривается от одной, до, в лучшем случае, пяти систем), он собрал его самостоятельно (см. раздел 1.6.8).

Мы не рассматриваем здесь аппаратные и гибридные DSM системы (такие как Alewife, DASH, FLASH, Typhoon и др.), сосредотачиваясь именно на программных решениях.

Для детального обзора каждой из систем потребовалось бы слишком много места, но мы выделим из списка, по крайней мере, наиболее известные системы и укажем их наиболее интересные (для данной работы) свойства.

1.6.1. Linda

Linda [24], по всей видимости, была предтечей известных нам сегодня DSM систем. Основываясь на уже существующем языке программирования, Linda предлагает расширить этот язык несколькими операциями, создать препроцессор, который «развернет» эти операции в вызовы рантайм библиотек Linda и далее собирать программу пользователя обычным для него способом. В частности были поддержаны языки Си и Фортран.

Ключевой объект в Linda – так называемое «пространство кортежей» – глобальное пространство, в которое каждый узел системы может добавить новый элемент или же прочесть/удалить существующий. Данное пространство выглядит как DSM, а кортеж напоминает структуру из языка Си. Однако операции для работы с этими структурами нетипичны для данного языка – к примеру, операция «`out("abc", 1, 2, 3)`» создаст кортеж с именем «abc», и значением, состоящим из нескольких чисел. Операция «`in(\"abc\", 1, ? i, ? j)`», которая может выполняться и на другом узле, извлечет этот кортеж, поместив значения 2 и 3 в переменные i и j соответственно. Если бы подходящего под «шаблон» кортежа в пространстве кортежей не нашлось, операция «in» была бы заблокирована до его появления. Кроме этого, система включает операции чтения без удаления и создания новых процессов.

Таким образом, разработка под Linda, с одной стороны, позволяет пользователю не изучать новые языки программирования, с другой же, эти несколько новых операций оказались настолько необычны, что разработка под Linda тре-

бовала перестройки подходов к программированию, а перенос готового ПО в Linda или обратное портирование в другие системы были нетривиальной задачей.

1.6.2. IVY

IVY [32] во многих источниках считается первой DSM системой, а указанная работа – первой, где в явном виде сформулирована известная и по сегодняшний день концепция DSM.

Реализует модель последовательной консистентности (см. раздел 1.4.2). Поддерживает страничную организацию распределённой памяти; при отсутствии страницы на локальном узле – обеспечивает её трансфер по сети.

Данная система была создана для того, чтобы имитировать мультипроцессоры и выполнять ПО, разработанное для них, без изменений, таким образом осуществляя мягкий переход к распределённым системам. С этой задачей система справилась, однако производительность решения была не очень высока. Для ликвидации этого недостатка были разработаны системы Munin и Midway (см. ниже), которые просят разработчика явно указывать, где допустимы более слабые модели консистентности – таким образом, изначальное ПО в случае этих систем приходится не сильно, но всё же модифицировать.

1.6.3. Munin

Munin [14, 17, 18] создавался в качестве заменя IVY, ставя перед собой те же цели, но стараясь достичь гораздо более высокой производительности. Организация памяти, как и в IVY, в Munin страничная (с использованием MMU), однако Munin умеет размещать отдельные переменные (точнее структуры данных) на разных страницах, что создает видимость DSM основанной на переменных.

С целью повышения производительности вместо последовательной конси-

стентности реализует модель консистентности по выходу (см. раздел 1.4.5), и каждый выход из критической секции сопровождает рассылкой по узлам системы дельты изменений соответствующих страниц. Другой способ повышения производительности – требование к программисту явным образом указывать категорию (одну из четырех предопределенных: read only, migratory, write-shared, conventional) каждой из общих переменных, с целью применения к ним наиболее эффективных алгоритмов.

Обладая высокой производительностью, Munin все же оказался сложен в использовании (программисту сложно контролировать у какой переменной какая должна быть категория и какие на каждой из переменных из-за этого ограничения), а программирование с ним должно вестись очень аккуратно – к примеру, доступ к общим переменным разрешен и за пределами критических секций, что может приводить к сложноуловимым ошибкам, так как консистентность памяти для такого доступа моделью не обеспечивается.

1.6.4. Midway

Midway [15, 16] появился чуть позже Munin, с той же целью, но иными подходами. В отличие от Munin, имеющего четыре категории общих переменных и одну модель консистентности, Midway имеет только одну категорию переменных, зато поддерживает три модели консистентности. Для корректной работы программисту необходимо явным образом связать общие переменные с синхронизационными.

Недостатком у Midway стало то, что если программист не выполнит требований системы (маркировка переменных как общих, связывание с синхронизационной переменной, доступ к общим переменным только внутри критических секций) – на этапе компиляции никаких ошибок выдано не будет, и, как и в случае с Munin, система будет работать некорректно при крайне затрудненных возможностях для обнаружения причин такого поведения.

1.6.5. Orca

Orca [12] представляет собой систему параллельного программирования с общением между узлами через DSM, организованную в виде объектов. Система состоит из языка программирования (причем не надстройки, а принципиально нового, хотя и похожего на Modula-2), соответствующего компилятора и системы выполнения. Последняя при этом независима и может использоваться в различных языках. Объекты в системе похожи на структуры языка Си – в них есть и члены, и методы, наследование же отсутствует.

Система немного похожа на Linda, но только на первый взгляд – объекты Orca гораздо богаче, а алгоритмы сложнее. Так как объекты Orca могут содержать методы, для возможности их удаленного вызова Orca поддерживает RPC протокол. В Orca имеется множество интересных алгоритмов. Например, Orca умеет работать как в сетях, поддерживающих надежные широковещательные сообщения, так и в сетях их не поддерживающих – в последнем случае Orca реализует этот функционал самостоятельно, поверх функционала сообщений точка-точка. Решение же о том, реплицировать ли ту или иную переменную или хранить локально, принимает компилятор, основываясь на своих расчетах соотношения операций записи и чтения соответствующего объекта (учитывая и циклы и наличие широковещательных сообщений в системе).

В целом, система очень мощная, но, следовательно, и «тяжелая» – Orca требует наличия собственного компилятора (соответственно имеет сложности с портированием на другие платформы) и изучения довольно экзотического по современным меркам языка программирования.

1.6.6. TreadMarks

TreadMarks [47, 48] пошел иным путём чем Orca и постарался стать для разработчика максимально понятным за счет API, очень схожего с POSIX Threads. TreadMarks – страничная DSM для Unix систем, написанная на Си и реализую-

щая модель ленивой консистентности по выходу (см. раздел 1.4.6), причём сразу три разновидности этой модели (lazy invalidate, lazy hibrid, eager invalidate). Подобно Munin, TreadMarks обнаруживает обращения к страницам через MMU, и, подобно, Munin рассылает затем дельту изменений. Однако так как модель ленивой консистентности не требует выполнять рассылку сразу же, система ожидает запросов с других узлов, зачастую накапливая дельту за несколько входов-выходов из критической секции.

Вероятно благодаря эффективной модели консистентности, качественной реализации, распространенной платформе (Unix) и удобстве для программиста, система стала очень популярна.

1.6.7. Grappa

Grappa [26] - современный DSM фреймворк для программирования кластеров, осуществляющих большие объемы вычислений в памяти и активно работающих с данными. Система исполнения реализована на Си++. Поддерживается оптимизация работы с общими переменными на уровне компилятора. Поддерживает модель последовательной консистентности (в случае отсутствия так называемых «гонок», англ. data-race, иначе говоря, в отсутствие в ПО фрагментов, зависящих от порядка выполнения кода относительно других узлов). Доступ к общей памяти осуществляется через операции делегирования – данные не обязательно перемещаются на узел, совершающий операцию записи, возможно перемещение самой операции в узел, владеющий данными. Для повышения эффективности использования каналов передачи данных, агрегирует отдельные сообщения в блоки.

В целом перспективно выглядящая система с четкой областью применения.

1.6.8. Перечень известных DSM решений

В таблице 1.1 представлен список существующих DSM решений. Перечень далеко не полон (в него не вошли Amber, Piranha, Mirage+, Samhita и многие другие). Однако в нём представлены наиболее известные решения[1]. Для большей наглядности список расположен в хронологическом порядке[2]. Решения сопровождаются отсылками на соответствующие работы, а также кратким комментарием[3].

1.6.9. Заключение

Мы кратко рассмотрели наиболее известные системы, реализующие концепцию DSM. Первыми решениями стали своеобразные эксперименты – попытки популризации DSM за счет облегчения миграции ПО для мультипроцессоров в распределённую вычислительную среду. Решения постепенно усложнялись в поисках «золотой середины» между удобством использования и производительностью. Вершиной на этом пути, стало, пожалуй, решение TreadMarks, но и оно не оказалось абсолютным. Развитие концепции сделало виток, и, обладая уже гораздо большим опытом, сегодняшние исследователи (по-прежнему замечая в DSM не полностью раскрытый потенциал) создают решения, как и на заре возникновения этой концепции – узконаправленные, но максимально эффективно решающие конкретные задачи в конкретном окружении.

[1]На субъективный взгляд автора. К тому же известность наиболее «молодых» решений оценивать пока рано.

[2]Следует учесть, что год создания того или иного решения довольно затруднительно определить точно и обычно в его качестве используется год ключевой публикации по теме.

[3]Который, пожалуй, более полезен для автора, так как позволяет быстро вспомнить решение, но ни в коем случае не является его исчерпывающей характеристикой.

Год	Название, публикации	Комментарий
1985	Linda [24]	Языковая надстройка для работы с кортежами
1986	IVY [32]	Последовательная консистентность, страничная DSM
1987	Emerald [22]	Объектная DSM
1989	Mirage [23]	Страничная, как расширение UNIX System V
1990	Munin [14, 17, 18]	DSM на переменных
1991	Midway [15, 16]	DSM на переменных
1991	Orca [12]	Отказоустойчивая, объектная, язык программирования, построена на Amoeba
1992	Mermaid [27]	Гетерогенная DSM
1994	TreadMarks [47, 48]	Unix, lazy release, Pthreads, команда Munin
1995	Calypso [13]	Простая DSM, расширяющая Си++
1999	JIAJIA [28]	DSM для расширения памяти
2006	VODCA [50]	Реализует view-based модель консистентности
2014	Grappa [26]	DSM система для разработки ПО кластеров
2015	Tardis [53]	DSM протокол, не требующий широковещательных рассылок
2017	Naplus [39]	Система DSM-коммуникации виртуальных машин (используется JIAJIA)

Таблица 1.1 – Известные DSM решения

1.7. Выводы

В данной главе были рассмотрены предпосылки возникновения концепции DSM, приведено её описание, история развития, а также проанализированы исследования других авторов.

Развитие вычислительных систем привело к широкому распространению многопроцессорных архитектур. От архитектур с общей памятью, характерных для тесносвязанных систем, наблюдается движение к слабосвязанным или распределённым системам, представляющим собой совокупность независимых вычислительных устройств, объединённых для решения некоторой задачи и формирующих таким образом мультиагентную систему, устройства которой общей памяти уже не имеют. Программирование распределённых систем оказывается гораздо более сложной задачей, чем разработка ПО для однопроцессорных или сильносвязанных систем, что стимулирует поиски альтернативных концепций организации обмена информацией в подобных системах. Концепция DSM предложила существенно более простой (с точки зрения пользователя) способ организации коммуникаций в распределённых системах, скрывая сложность реализации данной концепции под относительно простым интерфейсом, работа с которым напоминает программирование систем с общей памятью. На прикладном уровне DSM позволяет относительно легко создавать механизмы, традиционно считающиеся непростыми и характерными, ввиду высокой стоимости их разработки, скорее для «больших» систем (промышленных, военных, медицинских и др.)

Ранние реализации концепции DSM обладали низкой производительностью, что стимулировало активные исследования с целью дополнить удобство использования таких систем приемлемыми показателями производительности. Результатом стало создание множества моделей консистентности данных, наиболее поздние из которых позволяют достигать результатов производительности, сравнимых с результатами систем, созданных в рамках классической кон-

цепции обмена сообщениями. Однако так как условия функционирования систем и требования к ним со временем изменяются, продолжаются исследования, направленные на выработку новых моделей, более подходящих к тем или иным условиям использования.

С целью эффективной реализации той или иной модели консистентности множеством исследователей создаются алгоритмы, отвечающие разнообразным требованиям функционирования конкретных систем и обладающие различными характеристиками производительности и надёжности. Вопрос надёжности выделился в отдельную тему для исследований, и по сегодняшний день содержит множество нерешённых проблем.

Параллельно создаются конечные реализации DSM систем. Ранние реализации были скорее экспериментальными, демонстрирующими, например, возможность лёгкой адаптации программного обеспечения, разработанного для систем с общей памятью, к работе в распределённой среде. Постепенно решения становились всё более универсальными, а в последние годы тенденция вновь изменилась – сегодня DSM системы создаются для решения узконаправленных задач и всё чаще уже изначально внедряются в состав более крупных систем, расширяя их возможности DSM механизмами.

Несмотря на то, что концепция DSM применима к любым видам MAC или распределённых систем, сфера устройств Интернета вещей (англ. Internet of Things – IoT) – одна из наиболее бурно развивающихся, стимулирующая массовый интерес к распределённым системам – остаётся данной концепцией не охвачена. Таким образом, в настоящий момент имеется возможность привнести в бурно развивающуюся область MAC для IoT технологию DSM, позволяющую заметно упростить создание конечных распределённых решений, привнести в них сложный функционал «больших систем», сократив при этом время на разработку. Параллельно возникают задачи анализа, адаптации или создания новых моделей консистентности и алгоритмов, более полно соответствующих требованиям данной сферы.

Глава 2. Постановка и решение задачи

2.1. Назначение, требования и соглашения

Прежде чем приступить к созданию решения, определим предпосылки его возникновения, назначение и условия применимости.

2.1.1. Назначение решения

Всплеск интереса к распределённым решениям обусловлен бурным развитием сферы Интернета вещей (англ. Internet of Things – IoT), в том числе её индустриального сегмента, отражающего потребности Индустрии 4.0. В этой области типична ситуация, когда на относительно небольшой площади имеется множество интеллектуальных устройств, которым необходимо взаимодействовать между собой для решения стоящих перед ними задач. Системы могут быть самые разнообразные – от АСКУЭ до беспилотных летательных аппаратов, однако организация информационного обмена между ними может выть выполнена единым образом.

Для удобства дальнейших формулировок разделим все механизмы взаимодействия в MAC на три иерархических уровня: нижний уровень обеспечивает возможность связи вообще; верхний – определяет семантику передаваемой информации; средний же, основываясь на нижнем, учитывает специфику MAC, решает присущие такой системе задачи, предоставляя уровню выше возможность сосредоточиться на семантике, обусловленной целью создания конкретной системы. В данной терминологии целью исследования является разработка среднеуровневого DSM механизма для обеспечения взаимодействия агентов в мультиагентных системах.

Поскольку среднеуровневый механизм не может работать «сам по себе», ему необходим нижний уровень, обеспечивающий его связь с конкретным оборудованием. Данный уровень обычно реализуется в виде операционной системы,

а в случае микроконтроллеров – операционной системы реального времени. По причинам, описанным в разделе «Введение», существующие сегодня на рынке ОСРВ не вполне отвечают нашим запросам. Кроме инженерных и политических факторов (обусловленных современной внешнеполитической обстановкой), имеется фактор и идеологический – автор считает, что будущее в разработке под микроконтроллеры – за языком Си++ (несмотря на то, что на данный момент гораздо более популярен в данной области язык Си). На взгляд автора, здесь прослеживаются параллели в развитии подходов к программированию с настольными системами, где сегодня язык Си++ держит уверенный паритет с языком Си, а большинство авторов соглашаются, что Си++ позволяет создавать более структурированные и компактные решения. Имеющиеся же на рынке ОСРВ начали своё развитие достаточно давно – когда подобные идеи ещё не были достаточно распространены – а затем в созданных решениях оказалось уже слишком поздно что-то кардинально менять без риска отпугнуть миллионы пользователей из-за потери совместимости со старыми решениями.

Российская ОСРВ МАКС[7] изначально создавалась объектно-ориентированной, а основным её языком является Си++. Более того, назначение ОСРВ МАКС – упрощение создания МАС и IoT решений, а DSM механизм изначально планировался в качестве конкурентного преимущества системы. В связи с вышеизложенным, назначение создаваемого DSM решения – включение результатов в состав именно этой операционной системы (что нашло отражение и в названии создаваемого механизма – МАКС DSM).

Тем не менее, результаты исследования предполагаются достаточно универсальными для того, чтобы быть применимы и к другим системам и окружениям. Выделим основные требования к МАКС DSM:

- в системе может быть от одного до полутора десятков узлов;
- система допускает выход из строя отдельных узлов без потери общей функциональности.

Отметим, что последнее требование обеспечивает высокую надежность решения в условиях, когда отдельные узлы/агенты могут «выпадать» из системы (по причине проблем со связью, поломки и др.). Это заметно сужает круг возможных технических решений и вносит новую проблематику, которая реже рассматривается в теории организации многопроцессорных систем.

Итак, если говорить неформально, система МАКС DSM должна решать базовые проблемы МАС и предоставлять прикладному программисту возможность относительно простой реализации таких традиционно непростых верхнеуровневых и контекстно-зависимых возможностей, как резервирование, горячая замена оборудования, масштабирование и др.

Далее опишем требования к решению. Заметим, что в некоторых случаях требования обусловлены интеграцией с ОСРВ МАКС (например, требования к среде разработки), что однако не является существенным препятствием к портированию решения на другие платформы.

2.1.2. Аппаратное окружение

Как правило, ввиду большого количества одновременно задействуемых устройств и их повышенной автономности, основные требования к этим устройствам можно выразить «формулой» SWaP – размер, вес и энергопотребление (англ. size, weight and power). Очевидно, что классические рабочие станции не удовлетворяют данным требованиям, однако с ними прекрасно справляются микроконтроллеры. Одной из наиболее распространенных платформ в данном секторе являются платформы группы ARM Cortex-M (M0, M1, M3, M23, M4, M33), однако мы будем использовать их лишь как «ориентир», и хотя опробовать конечное решение будем именно на Cortex-M, при разработке МАКС DSM не будем ограничивать себя какими-либо особенностями данной или любой другой платформы (например, наличие MMU, в отличие от решений типа TreadMarks, нам требоваться не должно).

Итак, требования к аппаратному окружению:

– решение исполняется на микроконтроллерах ARM Cortex-M4 (совместимость со всей линейкой ARM Cortex-M предполагается, но не проверяется);
– требования или ограничения по наличию/отсутствию каких-либо платформо-специфичных особенностей – отсутствуют.

2.1.3. Программное окружение

Создаваемое решение планируется включить в состав операционной системы ОСРВ МАКС, разрабатываемой с целью обеспечения разработчиков удобными средствами проектирования распределённых решений на микроконтроллерах, что должно способствовать популяризации как операционной системы, так и DSM парадигмы ей предлагаемой.

Конечное решение, однако, должно быть выполнено в виде отдельного модуля, что позволит при необходимости встраивать его и в другие продукты. С целью обеспечения максимальной переносимости на различные платформы, разработка должна вестись на языке Си++ (в частности, доступном для микроконтроллеров Cortex-M). Отказ от языка Си вызван соображениями, описанными в разделе 2.1.1. В соответствии с требованиями раздела 2.1.2, МАКС DSM не должна использовать платформо-специфичные особенности конкретной системы, а значит и требовать наличия соответствующих API интерфейсов в ОС (платформо-специфичные инфраструктурные задачи решаются на уровене операционной системы, но доступ к ним «из вне» может отсутствовать). Также МАКС DSM должна собираться (компилироваться) в наиболее распространенных средах разработки под микроконтроллеры.

Итоговые требования к программному окружению и оформлению:

- язык программирования Си++;
- поддержка следующих сред разработки:
  - Keil MDK-ARM 5;
  - IAR Embedded Workbench for ARM 7.5;
- функционирование под управлением ОСРВ МАКС;
- реализация в виде отдельного модуля.

2.1.4. Физическое окружение

Разрабатываемая система предназначена для поддержки DSM парадигмы в коммуникациях MAC, и, в первую очередь, ориентирована на динамические группы агентов или устройств, в которых одни узлы со временем могут исчезать, а другие появляться. Как правило, такие сети организуются в радиоэфире (например, группа совместно действующих БПЛА), поэтому, в первую очередь, мы ориентируемся на беспроводное применение. Радиоэфир является разделяемой физической средой (в общем случае два сообщения не могут быть отправлены одновременно из-за эффекта интерференции[1]), и этим свойством мы можем пользоваться. При таком подходе, без потери общности и производительности, наше решение сможет быть использовано и в других разделяемых средах (например, для автомобильного оборудования на общей шине CAN или в соответствующих промышленных сетях). Применение в неразделяемых средах также допустимо, однако возможно снижение эффективности в виду более сложной реализации механизма обеспечения широковещательных сообщений (см. ниже).

[1]Хотя имеются технологии типа OFDMA, при использовании которых несколько узлов получают возможность излучать одновременно, даже стандарт WiMax, активно эту технологию использующий, в протоколе подуровня MAC среди четырех классов предоставляемых сервисов содержит один, в котором станции по-прежнему вынуждены конкурировать в попытках захватить канал, так как соединение в данном случае еще не установлено и персональный набор поднесущих станциям ещё не выделен.

Таким образом, требования к физическому окружению оказываются достаточно свободны:

– радиоэфир ИЛИ

– проводное подключение, желательно по схеме разделяемой физической среды.

2.1.5. Сетевое окружение

Во многих работах по DSM прослеживается сильная связь с конкретным сетевым окружением. Это вызвано тем, что DSM создаются для распределённых систем, а для них принципиально важен вопрос организации связи. Однако сложность создания DSM системы может быть заметно снижена, в случае если мы можем рассчитывать на наличие надежных примитивов групповой коммуникации. Один из наиболее известных таких примитивов – упорядоченные широковещательные сообщения (англ. total order broadcast, также известные как atomic broadcast)[1]. Кратко, данный примитив предназначен для рассылки сообщений сразу нескольким адресатам, причем таким образом, что все адресаты получают данные сообщения в одном и том же порядке, вне зависимости от того, кто их отправляет.

В DSM системах часто возникает задача упорядочивания распределённых операций записи (см. раздел 1.5.4), однако, на взгляд автора, эта задача сводится к обеспечению механизма упорядоченных широковещательных сообщений (действительно, при наличии последнего механизма, первая задача решается тривиально). В связи с этим, выглядит обоснованным выделить последнюю задачу в отдельную. Подобно сетевой модели OSI, разделяющей уровни протоколов, что позволяет решать задачи, свойственные каждому из уровней

[1]Иногда вместо термина «широковещательные» (англ. broadcast) используется термин «мультиадресный» (англ. multicast) – если имеется в виду рассылка не «всем», а фиксированной группе адресатов. В данной работе второй вариант не требуется и далее не рассматривается.

независимо, выделим задачу обеспечения упорядоченных широковещательных сообщений в уровень протоколов связи, расположенный ниже DSM системы. С одной стороны, это позволит сосредоточиться в решении именно на DSM специфике, с другой – такой подход обеспечит возможность использования нашей DSM системы в различных сетевых окружениях (и в различных физических средах, соответственно разделу выше) – нужно лишь четко обозначить, что от этого окружения системе требуется. К примеру, DSM система Orca (см. раздел 1.6.5) как раз умеет использовать имеющиеся в её окружении соответствующие механизмы, и только при их отсутствии реализует данные механизмы самостоятельно. Использованный подход крайне интересен, однако создание собственных механизмов в области обеспечения связи вынесем в проработку на будущее. Заметим, что имеется большое количество публикаций по теме обеспечения упорядоченных широковещательных сообщений – предложены десятки алгоритмов – но большое количество решений проблемы – само по себе является проблемой. Прекрасная работа проделана в материале [19] – в нём сделана попытка обобщить все накопленные по данной теме знания, описать все сложности и тонкие моменты, а главное, в данной работе классифицировано около шестидесяти известных алгоритмов, что сильно облегчает поиск подходящих в конкретной ситуации решений. В будущем данная работа может быть крайне полезна.

Отметим, что так как работа системы предполагается в разделяемой физической среде (хотя и не ограничивается ей), вопросы разделения должны быть решены уже на канальном уровне сетевой модели OSI. В таком случае сообщения в системе оказываются вынужденно упорядочены, и реализация механизма, описанного выше, может не потребовать дополнительных усилий. Однако ситуация может усложниться в случае mesh сетей, где два узла физически могут и «не слышать» друг друга, а связь обеспечивается через ретрансляцию данных другими узлами.

Стандарты беспроводной связи в целом фиксируются документами IEEE,

входящими в группу 802 (802.11, 802.16 и др.), при этом имеется немалое сходство с проводными стандартами подгруппы 802.3. Мы, однако, не будем требовать наличия полной поддержки какого-либо из известных стандартов, а вместо этого выделим явно – какой функционал от протокола нам нужен, оставляя при разработке сетевого протокола возможность идти либо по пути реализации стандартов, либо, жертвуя совместимостью, добиваться более высокой производительности через создание собственного специализированного решения.

В результате, итоговые требования к сетевому окружению состоят всего из одного пункта:

– наличие поддержки упорядоченных широковещательных сообщений.

2.2. Решение задачи

Решение задачи складывается из нескольких компонентов. Сначала – создадим модель консистентности, которой будем придерживаться и в рамках которой будущему пользователю системы предстоит создавать свои решения. Затем – разработаем ключевые алгоритмы для обеспечения требований модели и обоснованных показателей как производительности, так и устойчивости ко сбоям. И, наконец, выработаем принципы построения прикладного интерфейса создаваемого программного продукта.

Однако прежде чем приступить к созданию вышеописанных компонентов, выберем общую «стратегию». В решении задачи обеспечения когерентности памяти их выделяют две: invalidation и write-broadcast [33]. Первая допускает лишь одного владельца для каждого распределённого объекта, за счет чего сокращает накладные расходы на пересылку изменившихся данных; вторая – требует обновления данных во всех узлах-копиях после каждой операции записи. Write-broadcast подход рассматривается реже, так как необходимые для организации когерентности накладные расходы при нем особенно велики. Однако он хорошо применим в нашей ситуации, так как требует максимально

быстрого распространения изменившейся информации по всем узлам в системе, не допуская их уникальности. А так как наша система разрабатывается для функционирования преимущественно в условиях разделяемой среды, где любое сообщение по сути является широковещательным, данная стратегия не оказывает негативного влияния на производительность нашей системы. В соответствии с классификацией, принятой в [40], можно обозначить необходимую нам систему как full-replication (имеется множество копий одних и тех же данных), multiple reader/single writer (MRSW, допускающий параллельное чтение одних и тех же данных множеством узлов «одновременно», с упорядочиванием и разделением во времени операций записи) software (не зависящий от аппаратных особенностей и реализованный программно) алгоритм.

2.2.1. Усиленная модель консистентности по выходу

В разделе 1.4 мы описали основные модели консистентности. На первый взгляд, нам идеально подходит модель консистентности по выходу. Действительно, данная модель не требует так много пересылок данных как модели, описанные до неё, но, в то же время, минимизирует время обладания узлом уникальной информацией (что важно для обеспечения устойчивости системы ко сбоям отдельных узлов), так как в момент выхода из критической секции произведенные в общей памяти изменения должны быть распространены между остальными узлами сети.

Модели, описанные после, последним свойством уже не обладают – модель ленивой консистентности не только не гарантирует отправку измененных данных по выходу из критической секции, но и наоборот – старается повысить свою эффективность именно за счёт того, что требует рассылки данных только тем узлам, что явным образом нуждаются в них, и только тогда, когда они выполнят соответствующий запрос.

Модель консистентности по входу также не предоставляет подобных гарантий, однако содержит усовершенствование, которое представляется весьма

ценным, и, в то же время, не являющимся чем-то неотрывно связанным с идеологией самой модели: требование наличия связи общих переменных с синхронизационными переменными. Данным свойством хотелось бы дополнить модель консистентности по выходу, не перенося, однако, другого свойства – модель консистентности по входу содержит новое понятие – «владелец» распределённой переменной.

Как следует из раздела 1.4.7, понятие «владелец» было введено лишь для того, чтобы узел, работающий с общими данными (которые, однако, в данный момент не интересуют другие узлы), мог работать с этими данными без какого-либо информационного обмена с остальными узлами (в том числе многократно входя и выходя из критической секции), снижая тем самым нагрузку на каналы передачи данных в системе. В нашей новой модели это свойство избыточно, так как противоречит стремлению минимизировать время существования уникальных данных в узле. А если, подобно модели консистентности по выходу, данные будут распространяться в системе в момент выхода узла из критической секции, надобность в понятии «владелец» отпадает.

Второй интересный нам момент в модели консистентности по входу – разделение понятий эксклюзивного и неэксклюзивного захвата синхро-переменной. Свойство обеспечивает возможность одновременной работы нескольких узлов над одними и теми же данными (защищаемыми одной и той же синхро-переменной), если ими производятся только операции чтения. Свойство не является неразрывно связанным с другими свойствами модели и может быть из неё выделено.

Таким образом, мы готовы предложить новую модель – усиленную модель консистентности по выходу (англ. enhanced release consistency). По сути, представляющую собой оригинальную модель консистентности по выходу, дополненную двумя свойствами из модели консистентности по входу:

1. Наличие связи между общими и синхронизационными переменными.
2. Разделение захватов на эксклюзивные и неэксклюзивные.

Заметим, что модель ленивой консистентности по выходу (см. раздел 1.4.6) можно представить аналогичным образом – как модель консистентности по выходу со свойством отложенной рассылки обновлений из модели консистентности по входу.

Опишем предложенную модель в виде совокупности требований:

1. Любая переменная в общей памяти (ОП) проассоциирована с одной и только одной синхронизационной переменной (СП). Таким образом, каждой СП_N соответствует группа переменных в ОП (ГП_N).
2. Захват СП_N может быть эксклюзивным (разрешающим операции записи в ГП_N) и неэксклюзивным (разрешающим только чтение ГП_N).
3. Операция эксклюзивного захвата СП_N может быть завершена только после того, как все узлы освободят данную переменную (даже если она была захвачена неэксклюзивно).
4. Операция захвата СП_N может быть завершена только после того, как все предшествующие операции над ГП_N на всех узлах будут завершены.
5. Доступ к ГП_N возможен только в случае нахождения СП_N на данном узле в захваченном состоянии.
6. Операции захвата и освобождения СП должны подчиняться модели процессорной консистентности.

Используя нотацию, введённую в разделе 1.4.1, схематично данную модель можно выразить рисунком 2.1, введя новые обозначения – $S(X)$ для входа в критическую секцию X (от англ. start) и $F(X)$ для выхода из неё (от англ. finish). При этом подразумевается, что синхронизационная переменная X проассоциирована с общей переменной x.

P_1:	$S(X)$	$W(x)1$	$W(x)2$	$F(X)$				
P_2:					$S(X)$	$R(x)2$	$F(X)$	
P_3:								$R(x)1$

Рисунок 2.1 – Модель расширенной консистентности по выходу

К сожалению, этот рисунок оказался бы справедлив и для модели консистентности по входу, так как консистентность обеспечивается тоже только после входа в соответствующую критическую секцию (что видно на примере процессов P_2 и P_3). Однако принципиальное отличие нашей модели в том, что она использует операции входа в секцию типа $S(X)$ не для того, чтобы начать поиск соответствующих данных в сети, а лишь для того, чтобы дождаться завершения их передачи в случае если передача к данному моменту еще не успела завершиться.

Далее мы покажем, что внесенные в модель новые свойства могут быть реализованы способом, который не повлечет дополнительной нагрузки на конечного разработчика и не увеличит склонность конечного решения к ошибкам по невнимательности. В ближайших же разделах сосредоточимся на воплощении принципов описанной модели в алгоритмах.

2.2.2. Роли узлов и алгоритм смены роли

Все узлы в МАКС DSM с точки зрения пользователя – равноправны. У каждого есть доступ к общей памяти, и каждый может производить с ней операции, обмениваясь тем самым данными с другими узлами. Однако сохраняя равноправие на верхнем уровне (с точки зрения пользователя), внутри системы узлы могут выполнять различные функции, приобретая соответствующую специализацию. Введем в систему роли, перечисленные в таблице 2.1[1] и далее опишем их назначение.

Роль «Сервер» вводится с целью обеспечить максимальную надежность си-

[1]Сокращения в таблице образованы от английских слов new, server, backup, client соответственно.

Название	Сокращение	Описание
Новичок	new	Новый узел, роль не определена
Сервер	srv	Главный узел в системе
Копия	bck	Резервная копия сервера
Клиент	cln	Обычный узел

Таблица 2.1 – Роли узлов в МАКС DSM

стемы и простоту восстановления в случае сбоев. Здесь мы используем подход полной репликации данных, описанный в разделе 1.5.4. Данный подход требует выделения среди узлов особой роли – в англоязычной литературе эта роль известна как sequencer, мы же используем название «Сервер».

Роль «Копия» необходима для быстрого восстановления Сервера, в случае его сбоя. Несмотря на то, что Сервер не обладает уникальными данными, а лишь координирует работу остальных узлов, восстановление Сервера – задача более сложная и более продолжительная чем восстановление любого другого узла. Аналогично подходу, описанному в разделе 1.5.1, Копия будет «протоколировать» все операции Сервера, а в случае сбоя последнего, быстро возьмёт на себя его функции.

Роль «Клиент» введём для обозначения всех остальных узлов, не являющихся ни Сервером, ни Копией.

И, наконец, роль «Новичок» – временное состояние узла в момент его возникновения.

Так как МАКС DSM является динамической системой, роли узлов не могут быть фиксированными. Для наглядности схема возможной смены ролей представлена на рисунке 2.2. Каждый узел потенциально может выполнять любую роль, а также менять её в процессе работы, за единственным исключением – оставив однажды роль Новичок, вернуться к этой роли узел уже не сможет[1].

[1]Ситуации подобно перезагрузке узла мы рассматриваем как исчезновение узла и возникновение нового.

Условия, в которых узел может изменить свою роль, выражены алгоритмом на рисунке 2.3. Для удобства, названия процедур отражают текущую роль узла. Значения переменных x_1, x_2, x_3 определяются низлежащим сетевым уровнем и зависят от свойств конкретных каналов.

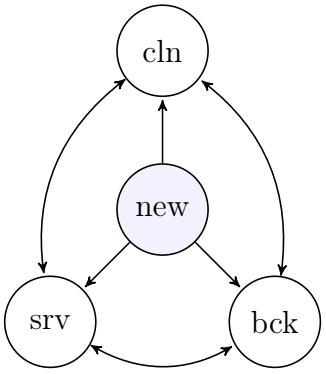


Рисунок 2.2 – Схема возможных изменений ролей узла в МАКС DSM

1: **procedure** Новичок
2: **if** продолжительность тишины в эфире $> x_1$ **then**
3: роль ← Сервер
4: **else if** у Сервера нет Копии **then**
5: роль ← Копия
6: **else**
7: роль ← Клиент
8: **procedure** Клиент
9: **if** продолжительность «молчания» Сервера $> x_2$ **then**
10: роль ← Сервер
11: **else if** сервер требует стать Копией **then**
12: роль ← Копия
13: **procedure** Копия
14: **if** продолжительность «молчания» Сервера $> x_3$ **then** ▷ $x_3 > x_2$
15: роль ← Сервер
16: **procedure** Сервер
17: **if** нет Копии И есть Клиенты **then**
18: потребовать от произвольного Клиента стать Копией

Рисунок 2.3 – Алгоритм смены роли узлом в МАКС DSM

2.2.3. Организация сообщений в типичных операциях системы

В разделе 2.2.2 мы сделали выбор, что пойдем по пути полной репликации данных и определили роли узлов в будущей системе.

Согласно принципам, изложенным в разделе 1.5.4, а также решению поддерживать копию Сервера, типичные операции в МАКС DSM будут выглядеть так, как изображено на рисунке 2.4.

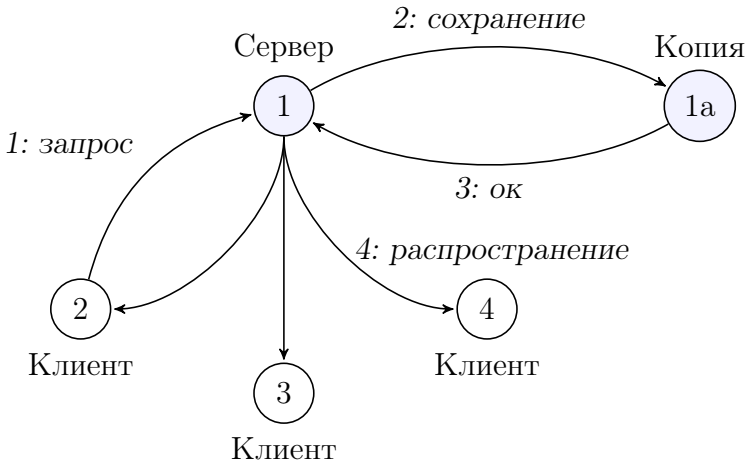


Рисунок 2.4 – Операция записи в МАКС DSM

Клиент (узел №2) пытается выйти из критической секции, и соответствующая инструкция блокируется на время осуществления следующих действий системы:

1. Клиент отправляет запрос на Сервер.
2. Сервер обращается к своей копии с целью сохранить информацию о запросе.
3. Копия производит запрошенную операцию и сообщает об этом Серверу.
4. Сервер оповещает все узлы системы, включая инициатора, об операции.

Только по получении сообщения, соответствующего шагу 4, узел-инициатор может быть уверен, что операция произведена и разблокировать инструкцию выхода из критической секции, продолжив выполнение прикладного кода

в обычном режиме[1].

Рассмотрим теперь действия узла при попытке входа в критическую секцию. Так как поведение оказывается достаточно тривиальным, представим шаги только в виде списка (номера шагов не имеют отношения к предыдущему рисунку). Как и в прошлый раз, операции, обрамляющие данный алгоритм, опишем за пределами списка.

Итак, Клиент пытается захватить определенную синхронизационную переменную (или, что то же самое, войти в критическую секцию) и блокируется. Управление передается механизму поддержки консистентности:

1. Клиент отправляет запрос Серверу с обозначением своего желания и способа, которым он хочет осуществить захват (эксклюзивный или неэксклюзивный).
2. Сервер, в случае если переменная не числится уже захваченной кем-либо эксклюзивно, обращается к Копии, чтобы учесть данное действие. Если же уже имеется эксклюзивный захват данной переменной – Сервер просто ожидает её освобождения (Клиент в это время остаётся заблокированным).
3. Копия, как обычно, учитывает происходящее и рапортует по завершении своих действий Серверу. При этом Копия не забывает сохранить информацию о том, какую именно переменную и в каком режиме захватил данный узел.
4. Сервер также запоминает информацию, описанную пунктом выше, после чего отправляет Клиенту сообщение с разрешением осуществить запрошенное действие.
5. Клиент разблокируется, входит в критическую секцию и осуществляет запланированную серию операций с общими переменными (в процессе чего

[1]Разблокировка также возможна по таймауту, данная ситуация рассматривается в разделе ниже.

никаких сообщений во вне не отправляется). Затем Клиент пытается выйти из секции, чем вновь вызывает свою временную блокировку и вызов механизма, описанного ранее.

2.2.4. Обеспечение отказоустойчивости

В теории обеспечения отказоустойчивости известно [42, с. 428] пять типов сбоев: авария, пропуск сообщения, временной сбой, ошибка отклика, случайная ошибка (так же известная как византийский сбой). Не углубляясь в обзор каждого из типов, отметим, что большинство из них необходимо рассматривать на низлежащем относительно DSM сетевом уровне, обеспечивающем упорядоченную доставку широковещательных сообщений. Византийский тип мы принципиально не рассматриваем ввиду экзотичности данного вида сбоя, алгоритмы защиты от которого при этом весьма ресурсоёмки (так как вынуждены учитывать возможность злонамеренного поведения отдельных узлов системы). Таким образом, в данной работе рассматривается единственный тип сбоя – «авария» (англ. crash).

В связи с тем, что наша система призвана надёжно функционировать в условиях выхода из строя отдельных узлов, необходимо рассмотреть все потенциально возможные ситуации возникновения аварий и соответствующие сценарии восстановления работоспособности системы.

Прежде всего необходимо защититься от ситуации, когда часть данных, представляющих одно логическое сообщение (например, пакет изменений, произошедших на узле в течение блокировки), получена и применена другим узлом лишь частично (например, передающий узел вышел из строя не завершив процесс передачи). Способ реализации данного требования описан в разделе 3.4, на данном же этапе нам достаточно отметить факт наличия данного требования.

Далее нам достаточно определить все возможные состояния системы и определить её поведение по нивелированию сбоев в каждом из состояний. На рис. 2.4 стр. 65 в предыдущем разделе изображены все возможные типы и на-

правления сообщений в МАКС DSM. Разобрав ситуации аварии каждого из узлов в различные моменты относительно отправки и приёма всех возможных сообщений, мы рассмотрим все потенциально возможные ситуации аварий в системе. Для удобства объединим узлы на рис. 2.4 под номерами 2, 3 и 4 в единый узел (так как они выполняют одну и ту же роль и нет необходимости рассматривать их независимо). Для удобства представим получившийся результат на рис. 2.5 (попутно уточнив описания курсирующих сообщений) и далее будем рассматривать именно его.

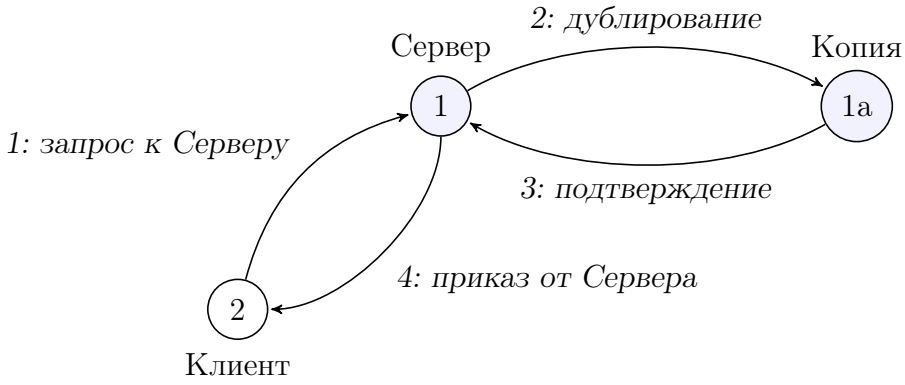


Рисунок 2.5 – Типы и направления сообщений в МАКС DSM

Далее вопрос отказоустойчивости рассматривается в разрезе реализации – в разделе 3.4.

2.2.5. Модель прикладного интерфейса

Одна из наиболее существенных характеристик DSM системы – удобство её использования. Характеристика в целом субъективная, однако некоторые объективные свойства всё же можно выделить. Рассмотренные выше реализации (см. раздел 1.6), как правило, обладали одним или несколькими из следующих свойств, отрицательно сказывающихся на широком распространении данных систем:

1. Реализация в виде нового языка программирования

   Яркий пример – система Orca [12]. Плюсы системы не отменяют факта,

что прикладному программисту в данном случае необходимо изучить новый язык программирования, а при портировании системы на новую платформу, портировать нужно и компилятор, что в общем случае является крайне трудоёмкой задачей.

2. Реализация в виде расширения стандартного языка программирования

   В данном случае для функционирования системы также необходима поддержка со стороны компилятора. Как минимум, требуется разработка препроцессора, трансформирующего расширенный язык в стандартный. Соответственно, использование и портирование подобных систем осложняется. Характерный пример – система Linda [24].

3. Введение новых концепций программирования

   Как правило, данное свойство сопровождается одним из вышеописанных. В целом, DSM система создаётся именно для того, чтобы предложить альтернативную, более удобную концепцию программирования, но иногда реализации этой концепции усложняются до такой степени, что разработка ПО под соответствующие системы начинает требовать существенных трудозатрат, что входит в некоторое противоречие с желанием упростить разработку. В качестве примеров можно привести всё те же Linda (с её концепцией кортежей) и Orca (с её новым языком программирования).

4. Повышенная нагрузка на прикладного программиста

   Некоторые системы, обладая высокой производительностью, достигают её за счет вовлечения прикладного программиста в процесс оптимизации. Например, система Munin [14, 17, 18] требует определения пользователем категории каждой из распределённых переменных, для чего необходимо хорошо владеть математическим аппаратом данной системы.

5. Отсутствие контроля ошибок на этапе компиляции

   Данный пункт в меньшей степени касается прикладного интерфейса систе-

мы, однако имеет существенные связи с ним. К примеру, в системе Midway [15, 16] доступ к распределённым переменным допускается не только внутри секций, гарантирующих консистентность, но и вне их (что является предметом определения прикладного интерфейса системы, её модели программирования). На фоне отсутствия соответствующих сообщений на этапе компиляции данная возможность приводит к сложнообнаруживаемым ошибкам. При этом ниже будет показано, что данной проблемы можно избежать даже без привлечения компилятора.

С целью избежать свойств, описанных выше, прикладной интерфейс системы МАКС DSM будет соответствовать следующим требованиям:

1. Реализация на стандартном языке Си++.
2. Классическая концепция DSM интерфейса – маркировка распределённых переменных и специальные секции для работы с ними.
3. Жесткая связь каждой распределённой переменной с конкретной секцией. Определяется пользователем один раз, в процессе выполнения программы не изменяется.
4. Ограничение возможности доступа к распределённым переменным за пределами соответствующих секций.
5. Синтаксическая лаконичность и самоочевидность результирующего прикладного кода.
6. Контроль выполнения заданных правил на этапе компиляции.

На первый взгляд, описанные требования могут показаться взаимоисключающими и невыполнимыми. Однако ниже будет показано, что задействование языка Си++ и препроцессора (что не типично для ранее создававшихся DSM систем) позволяет обеспечить выполнение данных принципов.

2.3. Выводы

В данной главе было рассмотрено назначение создаваемого решения, уточнены требования (к решению и его окружению), разработана новая модель усиленной консистентности по выходу, основные алгоритмы, а также принципы (требования) к прикладному интерфейсу, учитывающие недостатки ранее созданных систем.

Разрабатываемое решение предназначено для использования в мультиагентных системах в сфере IoT, включая системы АСКУЭ и беспилотные летательные аппараты. Создаваемая система призвана функционировать на маломощных микроконтроллерах без MMU, интегрируясь и задействуя в качестве низлежащего сетевого слоя операционную систему реального времени МАКС. Решение должно обеспечивать создание сетей из полутора десятков устройств, допускающих выход из строя отдельных узлов без потери общей функциональности.

Созданные ранее модели консистентности предлагают широкий выбор решений по критериям величины накладных сетевых расходов и удобства программирования. Так как нашей задачей является создание отказоустойчивого механизма, основной стратегией был избран write-broadcast [33] подход. Несмотря на то, что ранее данный подход не был популярен в связи с повышенной нагрузкой на сеть, в условиях радиоэфира (или иной разделяемой и принципиально широковещательной среды) его недостатки нивелируются, что позволяет эффективно задействовать его с целью минимизации времени, в течение которого отдельный узел системы может обладать уникальными данными (которые не могут быть восстановлены в случае выхода данного узла из строя). На основе модели консистентности по выходу, была создана её усиленная версия, включившая такие новые свойства как наличие связи между общими и синхронизационными переменными и разделение захватов на эксклюзивные и неэксклюзивные.

За внешним равноправием и взаимозаменяемостью узлов системы был расположен слой абстракции, вводящий понятие роли узла. Соответственно была разработана система ролей, их функций а также алгоритм смены роли узлом, отвечающий условиям динамичности системы. Были разработаны принципы информационного обмена в сети (верхнеуровневое описание протокола) и принципиальное решение задачи отказоустойчивости. Анализ слабых сторон ранее созданных DSM систем, отрицательно сказывающихся на их популярности, позволил выработать принципы к созданию прикладного интерфейса, свободного от выявленных недостатков.

Таким образом, в данной главе были решены все поставленные во введении теоретические задачи и «подготовлена почва» для программной реализации создаваемого механизма, рассматриваемой в следующей главе.

Глава 3. Программная реализация

Принципиальное решение части сформулированных во Введении задач описано в главе 2 (см. Выводы на стр. 71). В данной главе сосредоточимся на нюансах, важных с точки зрения конкретной реализации описанных принципов и алгоритмов, продолжив решение оставшихся задач.

3.1. Описание реализации прикладного интерфейса

Начинать знакомство с новым решением зачастую оказывается удобнее всего рассмотрев простейшую программу, в которой это решение используется. В соответствии с данной традицией приведем пример такой программы на рис. 3.1.

```
1  MDSM_DECLARE(X)
2      int i;
3  MDSM_DECLARE_END
4
5  int main()
6  {
7      int result = 2;
8
9      MDSM_ACCESS_RW(X)
10         MDSM_ITEM(i) = result + 1;
11     MDSM_ACCESS_END
12
13     MDSM_ACCESS_RO(X)
14         result = MDSM_ITEM(i) - 3;
15     MDSM_ACCESS_END
16
17     return result;
18 }
```

Рисунок 3.1 – Пример программы с использованием МАКС DSM

Представленная компактная программа не выполняет какой-либо практи-

чески значимой работы, но задействует, тем не менее, почти все возможности МАКС DSM. Более сложный пример рассмотрен в разделе 3.6, здесь же – рассмотрим основные возможности и интерфейс системы.

Строки 1–3 объявляют группу распределённых переменных с именем X. В данном примере группа содержит лишь одну переменную целочисленного типа с именем i, но может содержать произвольное количество переменных или объектов любого типа[1].

Строки 9–11 демонстрируют осуществление доступа к переменным группы X, причём доступ осуществляется на запись (эксклюзивный доступ). В терминах принятой модели консистентности (см. раздел 2.2.1) конструкции на строках 9 и 11 определяют и ограничивают соответствующую «критическую секцию».

Строки 13–15 аналогичным образом осуществляют доступ на чтение (неэксклюзивный).

Все конструкции МАКС DSM, доступные прикладному программисту в качестве интерфейсных операций, сведём в таблицу 3.1[2].

Рассмотрим подробности реализации данных интерфейсных операций и, параллельно, найденные способы удовлетворения требований модели интерфейса, описанные в разделе 2.2.5.

Синтаксис языка Си++ жестко определён, и, на первый взгляд, если действовать в рамках данного синтаксиса, не удастся создать лаконичное и простое в использовании расширение языка. По сути, в данной ситуации нам необходима техника метапрограммирования[3], позволяющая создать собственный метаязык, превращающийся в язык стандартный при помощи стандартного же компилятора. К счастью, стандарт языка Си++ описывает две функциональ-

[1]В текущей реализации отсутствует поддержка ссылок и указателей.

[2]Параметр <имя> в любой конструкции синтаксически должен представлять собой идентификатор языка Си++ с соответствующими ограничениями по именованию.

[3]Метапрограммирование – техника программирования, порождающая метапрограммы (программы, генерирующие другие программы).

Название	Описание
MDSM_DECLARE(<имя>)	Начало объявления группы распределённых переменных с именем <имя>.
MDSM_DECLARE_END	Конец объявления группы распределённых переменных.
MDSM_ACCESS_RW(<имя>)	Начало критической секции для доступа к распределённым переменным группы <имя>. Доступ на чтение и запись.
MDSM_ACCESS_RO(<имя>)	Начало критической секции для доступа к распределённым переменным группы <имя>. Доступ только на чтение.
MDSM_ACCESS_ON(<имя>)	Начало секции-обработчика события обновления одной или нескольких переменных группы <имя>.
MDSM_ACCESS_END	Конец критической секции.
MDSM_ITEM(<имя>)	Конструкция доступа к распределённой переменной <имя>.

Таблица 3.1 – Операции прикладного интерфейса МАКС DSM

ные возможности, которые потенциально могут быть использованы для реализации техники метапрограммирования – шаблоны и директивы препроцессора. Шаблоны, однако, подходят нам в меньшей степени – синтаксические конструкции с применением шаблонов оказываются слишком сложны для большинства программистов, а уровень поддержки со стороны компиляторов может различаться от продукта к продукту, что особенно характерно для компиляторов в области embedded. Таким образом, интерфейсная часть МАКС DSM строится на директивах стандартного препроцессора не требуя дополнительных инструментов среды разработки.

В соответствии с примером на рис. 3.1, строки 1–3, прикладной программист имеет возможность описывать распределённые переменные таким же образом, как и переменные обычные (локальные), единственное требование – обрамлять группы таких переменных директивами `MDSM_DECLARE` и `MDSM_DECLARE_END`. Однако внутри DSM необходимо иметь возможность работы со всеми переменными одной группы как с единым целым, непрерывной областью памяти (например, с целью (де)сериализации при передаче по сети). С этой целью обрамляющие директивы генерируют вокруг переменных пользователя структуру (`struct`). Структура, в свою очередь, помещается в класс (`class`), предоставляющий необходимые операции управления соответствующей группой распределённых переменных. С целью предотвращения генерации дубликатов кода, данный класс наследуется от некоторого базового класса, реализующего общие для всех подобных классов механизмы. С целью обеспечения возможности универсальной работы с любым из подобных классов в механизмах, описываемых ниже, все классы генерируются с одним и тем же именем. Возможность создания множества групп переменных в одной программе обеспечивается размещением каждого класса в индивидуально сгенерированном пространстве имен (`namespace`), содержащем в названии уникальный идентификатор, задаваемый пользователем в качестве параметра конструкции `MDSM_DECLARE`.

Конструкции `MDSM_ACCESS_RW` и `MDSM_ACCESS_RO`, с точки зрения пользователя, открывают критическую секцию для работы с соответствующей группой распределённых переменных. С точки зрения реализации, данные конструкции создают переменную-ссылку на экземпляр класса, содержащий интересующие пользователя переменные (параллельно сгенерировав уникальное имя соответствующего пространства имён). В зависимости от типа конструкции (`MDSM_ACCESS_RW` используется для доступа на чтение и запись, `MDSM_ACCESS_RO` – только для чтения), ссылка создается либо обычная, либо константная (`const`). Таким образом предотвращаются возможные ошибки пользователя,

при которых секция открывается на чтение, но в ней производится модификация распределённой переменной. С целью последующей унификации операций над распределёнными переменными, в любой секции ссылка создается всегда с одним и тем же именем. Возможность создания пользователем нескольких критических секций в пределах одного блока[1] обеспечивается генерацией для каждой критической секции обрамления из фигурных скобок, создающего индивидуальную область видимости для каждой переменной-ссылки. Сразу после создания ссылки на класс, генерируется код для вызова метода этого класса, обеспечивающего консистентность распределённых переменных соответствующей группы.

Конструкция `MDSM_ACCESS_END` завершает критическую секцию. Несмотря на то, что конструкция синтаксически не зависит от типа закрываемой секции (чтение-запись или только чтение), функционально поведение генерируемого кода различается: в случае доступа на чтение, дополнительных вызовов механизмов DSM не производится. В случае же доступа на чтение-запись, вызывается механизм распространения изменений по узлам системы. Синтаксической идентичности при различном поведении удаётся достичь использованием признака константности ссылки на класс, содержащий распределённые переменные. В обоих случаях генерируется вызов специального метода данного класса, параметром которому является эта же ссылка. Внутри класса определено две функции с одним и тем же именем (техника перегрузки), но различающихся наличием и отсутствием спецификатора `const` у параметра. Соответственно, константная версия функции содержит пустую реализацию, вторая же – вызов механизма распространения данных.

Конструкция `MDSM_ITEM` генерирует лишь обращение к указанной пользователем распределённой переменной, используя известное системе константное имя ссылки на класс, данную переменную содержащий.

[1]Блок – область программы, ограниченная фигурными скобками. Видимость локальной переменной ограничивается блоком, в котором она определена и распространяется на вложенные блоки.

Конструкция `MDSM_ACCESS_ON` стоит несколько особняком – поясним её назначение чуть подробнее. На практике часто возникает необходимость отслеживать изменение распределённых данных. Это нужно, например, при отображении на экране информации зависящей от таких данных, или для выполнения действий, являющихся реакцией на подобное изменение. Задачу можно решить путём регулярного опроса текущего значения данных в секции `MDSM_ACCESS_RO`, однако постоянные блокировки (даже на чтение) создадут существенную и непроизводительную нагрузку на систему DSM. Если же, с целью сократить нагрузку, использовать вызовы функций задержки – возрастёт латентность, что также не может считаться удачным решением. Описываемая конструкция как раз и призвана решить данную задачу. Вход в определяемую ей секцию произойдёт лишь после обновления какой-либо распределённой переменной из группы с именем, указанным в качестве параметра. Так как секция не выполняет никаких распределённых блокировок, с целью обеспечения целостности общих данных, в момент входа в секцию создаётся константная копия соответствующих данных, и именно она используется при обращении к переменным внутри секции. Удобно размещать подобные секции в отдельных потоках внутри бесконечных циклов – ресурсы процессора используются в данном случае эффективно, так как в ожидании обновления переменных соответствующей группы задача остается заблокированной.

Благодаря описанным выше механизмам в МАКС DSM удалось совместить простой синтаксис, проверку на соблюдение пользователем соглашений системы на этапе компиляции и отсутствие зависимости от каких-либо дополнительных компонентов среды разработки.

3.2. Сообщения

С целью упрощения дальнейших описаний, сведём основные сообщения МАКС DSM в единую таблицу 3.2[1].

Название	Описание
CLN_ASK_SRV_RO_LOCK SRV_ASK_BCK_RO_LOCK BCK_CNF_SRV_RO_LOCK SRV_CNF_CLN_RO_LOCK	Блокировка на чтение (RO).
CLN_ASK_SRV_RW_LOCK SRV_ASK_BCK_RW_LOCK BCK_CNF_SRV_RW_LOCK SRV_CNF_CLN_RW_LOCK	Блокировка на запись (RW).
CLN_ASK_SRV_UPDATE<данные> SRV_ASK_BCK_UPDATE<данные> BCK_CNF_SRV_UPDATE SRV_CNF_CLN_UPDATE<данные>	Снятие блокировки с обновлением данных на всех узлах.
CLN_ASK_SRV_RELEASE_LOCK SRV_ASK_BCK_RELEASE_LOCK BCK_CNF_SRV_RELEASE_LOCK SRV_CNF_CLN_RELEASE_LOCK	Снятие блокировки без обновления данных.

Таблица 3.2 – Внутренние сообщения МАКС DSM

Для удобства восприятия, все сообщения маркированы префиксами, обозначающими их источник и получателя. При этом используются следующие обозначения:

[1]Редкие сообщения, такие как приказ Сервера Клиенту стать Копией, в данной таблице опущены.

– CLN – Клиент (от англ. client)

– SRV – Сервер (от англ. server)

– BCK – Копия (от англ. backup)

– ASK – Запрос (от англ. ask)

– CNF – Подтверждение (от англ. confirmation)

Например, сообщение `CLN_ASK_SRV_RO_LOCK` означает запрос Клиента к Серверу на осуществление блокировки на чтение, `SRV_ASK_BCK_RO_LOCK` – перенаправление этого запроса Сервером к своей Копии, `BCK_CNF_SRV_RO_LOCK` – подтверждение Копией выполнения операции, и `SRV_CNF_CLN_RO_LOCK` – разрешение Сервера на осуществление блокировки Клиентом.

Остальные цепочки сообщений выглядят аналогичным образом[1], и, чтобы подчеркнуть данную общность, все сообщения в таблице объединены в соответствующие четвёрки, а описание для каждой расшифровывает назначение всей цепочки (назначение отдельных сообщений можно расшифровать, используя информацию выше).

3.3. Процесс блокировки

Рассмотрим реализацию этапов процесса блокировки – на запись (эксклюзивную) и на чтение (неэксклюзивную).

3.3.1. Реализация блокировки на запись

1. Клиент отправляет сообщение `CLN_ASK_SRV_RW_LOCK`, в котором указан идентификатор разделяемых данных. Задача, инициировавшая запрос, блокируется до получения ответа.

[1]Следует отметить, что сообщения SRV_CNF_CLN_UPDATE обрабатываются не только инициатором, но всеми Клиентами системы, обновляя локальные значения соответствующих распределённых переменных (см. ниже).

2. Сервер, получив запрос на блокировку, проверяет текущее состояние требуемых данных. Если ресурс уже заблокирован, то запрос на блокировку ставится в очередь FIFO. Если же блокировка возможна, то сервер помечает данные как блокированные и отправляет сообщение `SRV_ASK_BCK_RW_LOCK`.

3. Копия, получив запрос на сохранение блокировки, также помечает у себя данные как блокированные и отправляет сообщение `BCK_CNF_SRV_RW_LOCK`. Заметим, что здесь попутно проверяется активность как сервера, так и копии, поскольку продолжение сценария требует обмена сообщениями между ними.

4. Сервер, получив подтверждение сохранения блокировки, отправляет сообщение `SRV_CNF_CLN_RW_LOCK`. Поскольку гарантируется как порядок, так и доставка сообщений, то клиент получит это сообщение только после всех обновлений данных (см. ниже) и его локальное значение будет актуальным.

5. Клиент, получив разрешение на блокировку, возобновляет выполнение задачи, потребовавшей эксклюзивный доступ к разделяемым данным. Теперь клиент может быть уверен, что имеющиеся у него данные актуальны и не будут изменены другими узлами, пока блокировка сохраняется.

6. Закончив работу с данными, клиент отправляет сообщение `CLN_ASK_SRV_UPDATE`, вместе с которым передаётся новое значение данных. Их целостность обеспечивается системой доставки сообщений.

7. Сервер, получив запрос на обновление данных, сохраняет у себя новое значение и посылает сообщение `SRV_ASK_BCK_UPDATE`, в котором также передаётся новое значение данных.

8. Копия, получив запрос на сохранение данных, также обновляет у себя их текущее значение и отправляет сообщение `BCK_CNF_SRV_UPDATE`. Одно-

временно осуществляется контроль функционирования как Сервера, так и Копии.

9. Сервер, получив подтверждение сохранения данных, отправляет сообщение `SRV_CNF_CLN_UPDATE` с новым значением данных. Это сообщение предназначено для всех узлов. После отправки сообщения пометка о блокировке данных снимается, и сервер проверяет наличие других запросов на блокировку в очереди. Если такие запросы имеются, то начинается их обработка. Заметим, что в случае запроса блокировки на чтение (см. ниже), совместно с ним будут обработаны и все остальные накопившиеся запросы блокировки на чтение, так как они не мешают друг другу.
10. Клиенты, получив команду на обновление данных, устанавливают новое значение. Поскольку сообщения могут дойти до узлов с разными задержками, синхронность данных в реальном времени не достигается, но благодаря отсутствию потерь и сохранению порядка сообщений, требования модели консистентности будет выполнены.

3.3.2. Реализация блокировки на чтение

1. Клиент отправляет сообщение `CLN_ASK_SRV_RO_LOCK`, в котором указан идентификатор разделяемых данных. Задача, инициировавшая запрос, блокируется до получения ответа.
2. Сервер, получив запрос на блокировку, проверяет текущее состояние требуемых данных. Если ресурс уже заблокирован на запись, то запрос на блокировку ставится в очередь FIFO. Если же блокировка возможна, то сервер помечает данные как блокированные и отправляет сообщение `SRV_ASK_BCK_RO_LOCK`. Заметим, что наличие блокировок на чтение в данном случае не является препятствием.
3. Копия, получив запрос на сохранение блокировки, также помечает у себя данные как блокированные и отправляет сообщение

BCK_CNF_SRV_RO_LOCK. Заметим, что здесь попутно проверяется активность как сервера, так и копии, поскольку продолжение сценария требует обмена сообщениями между ними.

4. Сервер, получив подтверждение сохранения блокировки, отправляет сообщение SRV_CNF_CLN_RO_LOCK. Поскольку гарантируется как порядок, так и доставка сообщений, то клиент получит это сообщение только после всех обновлений данных (см. выше) и его локальное значение будет актуальным.

5. Клиент, получив разрешение на блокировку, возобновляет выполнение задачи, потребовавшей неэксклюзивный доступ к разделяемым данным. Теперь клиент может быть уверен, что имеющиеся у него данные актуальны и не будут изменены другими узлами, пока блокировка сохраняется.

6. Закончив работу с данными, клиент отправляет сообщение CLN_ASK_SRV_RELEASE_LOCK. Поскольку данные не изменялись, передавать их значение нет необходимости.

7. Сервер, получив запрос на снятие блокировки, посылает сообщение SRV_ASK_BCK_RELEASE_LOCK.

8. Копия, получив запрос на сохранение снятия блокировки, отражает эту операцию в своих системных данных и отправляет сообщение BCK_CNF_SRV_RELEASE_LOCK. Параллельно происходит контроль функционирования как Сервера, так и Копии.

9. Сервер, получив подтверждение сохранения снятия блокировки, удаляет узел из списка блокирующих. Если список оказался пустым, то пометка о блокировке данных снимается, и Сервер проверяет наличие других запросов на блокировку в очереди. Если такие запросы имеются – начинается их обработка.

3.4. Отказоустойчивость

3.4.1. Термин «сообщение» и атомарность

Прежде всего необходимо уточнить значение термина «сообщение». В разделах выше мы использовали данный термин в двух значениях – говоря о сущности, формируемой примитивами `Send` и `Receive`, а также при рассмотрении сущностей, курсирующих на уровне DSM, как, например, представлено на рис. 2.4 стр. 65. Временно разделим эти понятия, назвав первое «низкоуровневым сообщением», а второе, соответственно, «высокоуровневым сообщением». На уровне реализации последний тип не всегда совпадает с первым. Весь информационный обмен в МАКС DSM построен вокруг ключевых высокоуровневых сообщений, являющихся по сути целыми пакетами низкоуровневых сообщений с данными о произошедших в узле изменениях распределённых переменных. Если информации, которую необходимо передать, немного, и она помещается в одно низкоуровневое сообщение, пакет будет состоять из него одного, и значения двух терминов в данном случае будут совпадать. Если же данных много, пакет будет состоять из нескольких низкоуровневых сообщений. С целью избежать ситуаций, когда одна часть такого пакета доставлена, а другая часть утеряна (например, вследствие сбоя передающего узла), пакеты в МАКС DSM реализованы атомарными. То есть, несмотря на то, что пакет может состоять из нескольких сообщений низкого уровня, все такие сообщения составляют единый информационный блок, который интерпретируется на узле-получателе только в момент получения последнего из составляющих такой блок сообщений. Такой подход используется для абсолютно всех типов высокоуровневых сообщений в МАКС DSM, поэтому в системе как для низкоуровневых так и для высокоуровневых сообщений остаются возможны лишь две ситуации: 1) сообщение доставлено 2) сообщение не доставлено. В связи с этим, нет практической пользы от дальнейшего разделения двух сущностей и в будущем обе сущности мы будем называть единым термином «сообщение».

3.4.2. Действия при выходе узлов из строя

Воздействие Клиента на систему в протоколе МАКС DSM в конечном счёте сводится к отправке ключевых сообщений `CLN_ASK_SRV_UPDATE` или `CLN_ASK_SRV_RELEASE_LOCK` после завершения Клиентом работы с распределёнными данными. Если Клиент запросил блокировку и вышел из строя, то Сервер, обнаружив наличие блокировки в течение времени, превышающего заданное[1], принудительно её снимает. При этом, в случае блокировки на запись, она передаётся следующему в очереди узлу. В случае же блокировки на чтение (или отсутствии узлов в очереди), Сервер посылает Копии сообщение `SRV_ASK_BCK_RELEASE_LOCK` и считает блокировку исчезнувшим узлом снятой. Узел помечается как «пропавший», и если от него всё-таки придёт сообщение `CLN_ASK_SRV_UPDATE` или `CLN_ASK_SRV_RELEASE_LOCK`, то оно игнорируется. Клиент может продолжить работу с сообщений `CLN_ASK_SRV_RW_LOCK` или `CLN_ASK_SRV_RO_LOCK`.

Функционирование Копии проявляется в системе отправкой подтверждающих сообщений `BCK_CNF_*` в ответ на сообщения сервера `SRV_ASK_BCK_*`. Не получив от Копии ответа в течение заданного времени, Сервер делает вывод о выходе Копии из строя и назначает один из Клиентов новой Копией, отправив ему специальное сообщение. Поскольку все узлы поддерживают текущее состояние блокировок в актуальном состоянии, любой из них может занять роль Копии.

Выход из строя Сервера возможен в следующих случаях:

1. После получения сообщения `CLN_ASK_SRV_*_LOCK`.

   Копия отслеживает появление таких сообщений и ожидает получить от Сервера сообщение `SRV_ASK_BCK_*_LOCK` в течение заданного времени. Если ожидаемое сообщение не было получено, Копия «делает вывод» о вы-

[1] Этот и другие таймауты определяются низлежащим сетевым уровнем и зависят от способа организации механизма упорядоченной доставки широковещательных сообщений.

ходе Сервера из строя и берёт на себя его роль. Один из Клиентов назначается новым Сервером новой Копией, и система продолжает работу.

2. После получения сообщения `BCK_CNF_SRV_*_LOCK`.

   Здесь Копия отслеживает отправку Сервером сообщения `SRV_CNF_CLN_*_LOCK` и, в случае его отсутствия, действует аналогично предыдущему пункту.

3. После получения сообщения `CLN_ASK_SRV_UPDATE` или `CLN_ASK_SRV_RELEASE_LOCK`.

   Сервер должен отреагировать сообщением `SRV_ASK_BCK_UPDATE` или `SRV_ASK_BCK_RELEASE_LOCK`. Обработка аналогична пунктам выше.

4. После получения сообщения `BCK_CNF_SRV_UPDATE` или `BCK_CNF_SRV_RELEASE_LOCK`.

   Сервер должен отреагировать сообщением `SRV_CNF_CLN_UPDATE` или `SRV_CNF_CLN_RELEASE_LOCK` соответственно. Обработка аналогична пунктам выше.

5. При выделении блокировки следующему по очереди узлу.

   Копия не поддерживает очереди запросов на блокировки и не может распознать такую ситуацию. Однако если узел, ожидающий выполнения запроса на блокировку, не обнаруживает активности Сервера в течение заданного времени, то запрос на блокировку повторяется. В этом случае Копия отработает процедуру согласно пункту 1.

Одновременный выход из строя Сервера и Копии является максимально сложным случаем. Ситуация определяется по отсутствию активности Сервера в течение длительного времени. В этом случае Клиент с минимальным значением ID берет роль Сервера на себя и назначает новую Копию. Список имеющихся ID предоставляется системой гарантированной доставки сообщений[1].

[1]Что выглядит лишней связью между уровнями абстракции, и в перспективе данный механизм может

3.5. Программная архитектура

3.5.1. Верхнеуровневая архитектура

На верхнем уровне узел МАС, функционирующий под управлением ОСРВ МАКС с интегрированным решением МАКС DSM можно представить в виде, изображенном на рис. 3.2.

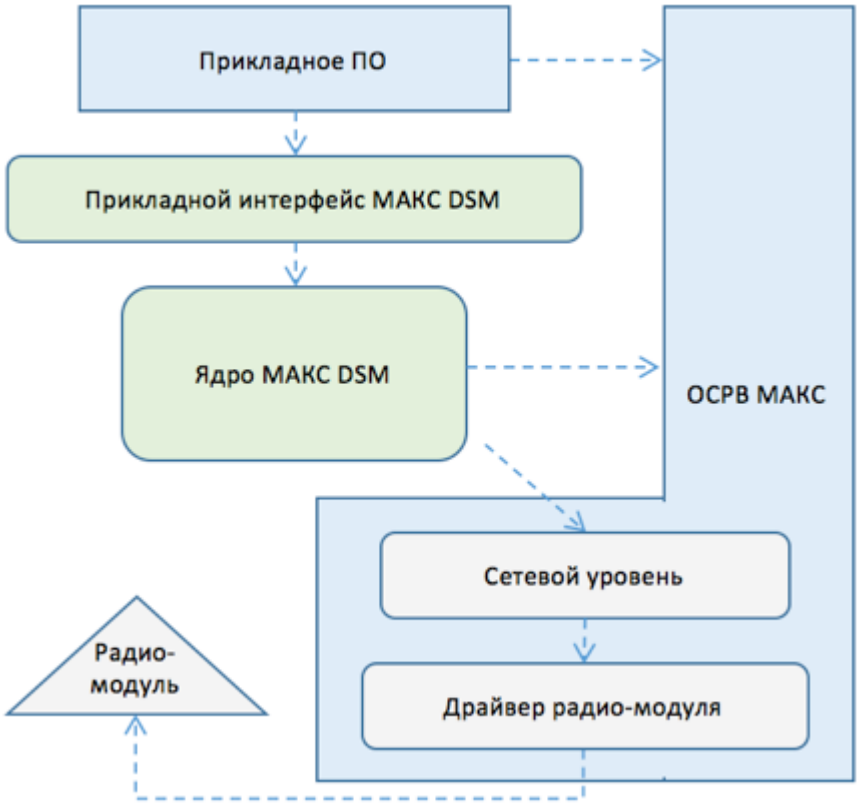


Рисунок 3.2 – МАКС DSM в составе функционирующего узла МАС

Основными компонентами являются:

1. Прикладное ПО. Соответствует пользовательскому приложению, разработанному для функционирования под управлением ОСРВ МАКС и использующему возможности МАКС DSM для совместной работы с данными группы устройств в системе.
2. Прикладной интерфейс МАКС DSM. Предоставляет интерфейс для ис-

быть реализован иначе.

пользования разделяемых данных. Содержит средства для описания разделяемых переменных, а также определения секций доступа.

3. Ядро МАКС DSM. Содержит алгоритмы DSM и осуществляет реализацию прикладного интерфейса. Выполняет операции, необходимые для совместного доступа к данным (блокировки, копирование, управление объектами синхронизации), формирует и обрабатывает сообщения протокола DSM, обеспечивает выполнение узлом назначенной ему роли.
4. Радиомодуль. Аппаратура для передачи данных через эфир. Содержит приемопередатчик и средства управления им.
5. ОСРВ МАКС. Является средой для выполнения всех программных компонентов. Управляет пользовательскими и служебными задачами, предоставляет интерфейс к аппаратуре, содержит вспомогательные алгоритмы, а также указанные ниже компоненты.
6. Сетевой уровень. Служит для приёма и отправки широковещательных сообщений с гарантированной доставкой. Является низлежащим уровнем системы связи относительно МАКС DSM.
7. Драйвер радиомодуля. Предоставляет программный интерфейс к возможностям радиомодуля. Преобразует пакеты сетевого уровня для отправки через физический канал и обратно.

Результаты данного исследования воплощены в компонентах, указанных в списке выше под номерами 2 и 3. Однако так как реализация прикладного интерфейса МАКС DSM уже была достаточно подробно рассмотрена в разделе 3.1, сосредоточимся на описании ядра созданной системы.

3.5.2. Основные компоненты ядра МАКС DSM

В нотации UML основные классы МАКС DSM представлены на рис. 3.3. Используется упрощенный вариант нотации (опущена информация по членам

классов, а также множество менее значимых классов).

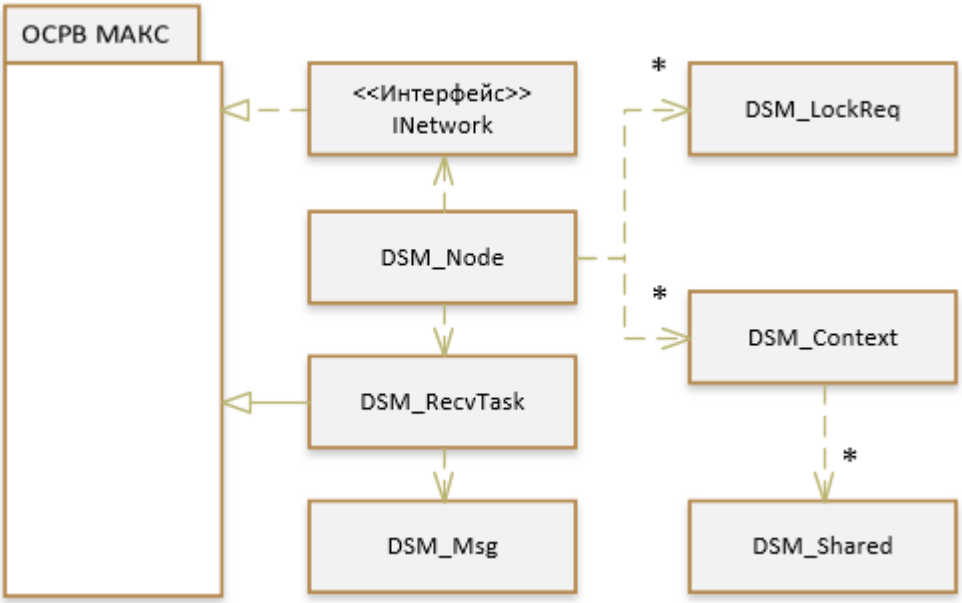


Рисунок 3.3 – Основные классы МАКС DSM в нотации UML

Основой реализации МАКС DSM является класс `DSM_Node`. Существует единственный экземпляр этого класса для каждого узла (паттерн проектирования «Одиночка»). Объект хранит всю информацию, относящуюся к МАКС DSM, контролирует приём и передачу сообщений через низлежащий сетевой уровень, а также обеспечивает пользовательский интерфейс, представленный секциями доступа к разделяемым данным.

Для хранения информации о разделяемых данных предназначен класс `DSM_Context`. Список экземпляров классов данного типа хранится внутри `DSM_Node`. В `DSM_Context` также содержится опущенная на схеме информация о состоянии блокировки, очередь запросов на блокировку (см. класс `DSM_LockReq`), мьютексы, обеспечивающие потокобезопасность, а также отраженный на схеме список объектов типа `MDSM_Shared` для секций, ожидающих изменения данных.

Класс `MDSM_Shared` служит для работы с разделяемой переменной. Существует единственный экземпляр, хранящий её текущее значение. Помимо этого, могут быть созданы экземпляры с копией значения, используемые в секциях для отслеживания изменений и мгновенного доступа без блокировки.

Сообщения МАКС DSM представлены классом `DSM_Msg`.Для получения сообщений служит класс `DSM_RecvTask` (задача ОСРВ МАКС). Он получает пакет из низлежащего сетевого уровня, преобразует этот пакет в сообщение, и, в зависимости от типа сообщения, вызывает метод из `DSM_Node` для его обработки. Кроме этого, производится фильтрация сообщений – таким образом, узел в системе получает только те сообщения, что предназначены для его роли.

Класс `DSM_LockReq` представляет собой запрос на блокировку. Он содержит ID узла, пытающегося осуществить блокировку и признак эксклюзивности доступа. Экземпляры образуют очередь FIFO, связанную с объектом `DSM_Context`. После того, как разделяемая переменная становится доступна для блокировки, Сервер выбирает из очереди следующий по порядку запрос и выполняет соответствующую операцию. Если требуется неэксклюзивная блокировка, то в течение одного сеанса будут удовлетворены все запросы данного типа в очереди, так как они не зависят друг от друга, и принципы модели консистентности не нарушаются.

3.6. Эксперимент

В данном разделе рассмотрим простую задачу, демонстрирующую работоспособность созданной системы МАКС DSM.

Постановка задачи

Пусть необходимо обеспечить отказоустойчивое выполнение некоторой вычислительной задачи несколькими устройствами. Под отказоустойчивостью здесь понимается продолжение выполнения задачи при выходе из строя одного или более произвольных устройств системы до тех пор, пока остаётся, по крайней мере, одно исправное устройство. При выходе из строя устройств системы, потери данных (промежуточных результатов) решаемой задачи происходить не должно.

Решение задачи

Под вычислительной задачей примем простое увеличение целочисленного счётчика. Система будет состоять из трёх устройств, аналогичных описанным в разделе 3.7.1 и дополненных цветными LCD экранами. Имитация выхода устройств из строя производится путём их выключения (отключения питания). Визуализация работы системы производится путём отображения на экранах устройств текущего значения счётчика. С целью повышения наглядности, увеличение счётчика должно происходить примерно один раз в секунду.

Ключевые фрагменты программы представлены на рис. 3.4, при этом опущен код для создания потоков и реализация функции `Show`, осуществляющей вывод на экран.

```
1  MDSM_DECLARE(X)
2      int counter;
3  MDSM_DECLARE_END
4
5  // Поток №1
6  while (true)
7  {
8      MDSM_ACCESS_RW(X)
9          ++ MDSM_ITEM(counter);
10         Task::Delay(1 SEC);
11     MDSM_ACCESS_END
12 }
13
14 // Поток №2
15 while (true)
16 {
17     MDSM_ACCESS_ON(X)
18         Show(MDSM_ITEM(counter));
19     MDSM_ACCESS_END
20 }
```

Рисунок 3.4 – МАКС DSM программа «счётчик»

Аналогично представленному в разделе 3.1 примеру, в строках 1–3 опре-

деляется группа распределённых переменных с именем X, содержащая единственную переменную `counter`.

Строки 6–12 являются ключевыми – именно здесь выполняется «вычислительная задача» – увеличение распределённого счётчика. Данный код выполняется в отдельном потоке и представляет собой бесконечный цикл, что, однако, не приводит к повышенной нагрузке на процессор, так как фрагмент содержит паузу в одну секунду, а также операцию `MDSM_ACCESS_RW`, которая блокирует выполнение потока до момента осуществления распределённой блокировки. Так как модификация счётчика выполняется в секции, определяемой конструкцией `MDSM_ACCESS_RW`, гарантируется, что лишь одно устройство в один момент времени изменяет счетчик. Задержка же, размещенная внутри критической секции, гарантирует, что блокировки не будут случаться чаще, чем раз в секунду. Как только время задержки истечёт, будет осуществлена блокировка другим (или тем же самым)[1] устройством и выполнится очередное увеличение значения счётчика.

Строки 15–20 показывают содержимое второго потока, отвечающего за отображение значения счётчика на экране. Здесь используется секция, блокирующая выполнение потока до возникновения события изменения распределённых данных. Как только текущее устройство обнаружит факт изменения, выполнится вход в секцию (причём значением объектов группы X будет результат произошедшего обновления). Функция `Show` выводит значение, переданное в качестве параметра на экран. После выхода из секции начинается новый цикл ожидания.

После загрузки данной программы в тестовые устройства, отладочный стенд вёл себя в полном соответствии с ожиданиями: устройства отображали одно и то же значение, возрастающее на единицу каждую секунду[2]. При этом

[1]Поскольку узел МАКС DSM, выполняющий роль Сервера (см. раздел 2.2.2), поддерживает очереди запросов на блокировки, запрос от текущего устройства окажется в конце очереди, а блокировку получит какое-то другое устройство. Впрочем, это не существенно для выполнения условий поставленной задачи.

[2]Возникновения дополнительных задержек удалось избежать благодаря поддержке очередей запросов

отображение нового значения выполнялось на всех устройствах синхронно[1].

Выполнение требования отказоустойчивости было проверено методом бета-тестирования – в серии экспериментов произвольным образом отключалось питание у одного, а затем и второго устройства из трёх. Ни в одном из экспериментов сбоя системы добиться не удалось – счётчик корректно и монотонно увеличивался на остающихся устройствах (остающемся устройстве). Однако было замечено, что секундная задержка в случае выключения устройств не всегда выдерживается. Проведенный анализ показал, что:

1. Если устройство владеет блокировкой и отсчитывает секундную паузу, выключение такого устройства в самом начале его отсчёта приводит к сокращению паузы между приращениями счётчика. Это происходит по причине передачи блокировки другому устройству, которое при её получении сразу выполняет увеличение счётчика, и только затем начинает отсчитывать новую задержку. Таким образом, информация о задержке отключённого устройства теряется.
2. Выключение Сервера примерно в момент завершения интервала задержки приводит к увеличению стандартной паузы за счет дополнительного времени, необходимого на восстановление Сервера.

Эффект под номером 2 визуально заметить сложно, однако первый эффект вполне различим, и его можно усилить, увеличив интервал задержки до нескольких секунд. Таким образом, время задержки в данном решении задачи – величина не константная, что, однако, не противоречит изначально сформулированным условиям решения.

Несмотря на то, что полученные результаты полностью соответствуют поставленной задаче, может возникнуть вопрос – какое именно устройство изме-

---

на блокировку, обеспечиваемой ролью Сервер и упомянутых выше.

[1]Данный эффект – следствие использования конструкции MDSM_ACCESS_ON. Иная реализация, скажем через постоянный опрос наблюдаемой переменной с некоторой задержкой, неминуемо привёл бы к визуально рассинхронизованному поведению устройств.

няет значение счётчика? Исходя из реализации очередей блокировок можно ожидать, что устройства будут захватывать счетчик по очереди. Для проверки данной гипотезы расширим экспериментальную программу до представленной на рис. 3.5.

```
1  MDSM_DECLARE(X)
2      int counter;
3      COLOR color;
4  MDSM_DECLARE_END
5
6  // Поток №1
7  while (true)
8  {
9      MDSM_ACCESS_RW(X)
10         ++ MDSM_ITEM(counter);
11         MDSM_ITEM(color) = MY_COLOR;
12         Task::Delay(1 SEC);
13     MDSM_ACCESS_END
14 }
15
16 // Поток №2
17 while (true)
18 {
19     MDSM_ACCESS_ON(X)
20         Show(MDSM_ITEM(counter), MDSM_ITEM(color));
21     MDSM_ACCESS_END
22 }
```

Рисунок 3.5 – МАКС DSM программа «цветной счётчик»

Группа распределённых переменных X пополнилась новой переменной `color` (строка 3), содержащей код цвета устройства, которое последним произвело увеличение распределённого счётчика. Для достижения данного функционала в первый поток программы добавлена инструкция на строке 11, в которой используется определение `MY_COLOR`, представляющее собой определение кода цвета устройства в зависимости от его идентификатора[1]. Соответственно,

[1]Инициализация/вычисление MY_COLOR в листинге не приводится как не существенное.

функция `Show` была расширена параметром, определяющим код цвета, который передается в неё во втором потоке на строке 20.

Описанное усовершенствование позволило наблюдать смену цвета выводимого значения счётчика при каждом его изменении, что подтвердило вышеописанную гипотезу.

3.7. Производительность

3.7.1. Производительность для двух узлов

Для проведения измерений был создан программно-аппаратный стенд, состоящий из двух плат STM32F429I-Discovery с подсоединенными к ним радиомодулями nRF24L01. Данные модули обеспечивают приём и передачу данных на частоте 2.4 ГГц с заявленной скоростью до 2 Мбит/сек. Однако достигнутая на практике скорость оказалась существенно ниже, так как, во-первых, данные передаются аппаратными пакетами длиной до 32 байт, причём время передачи пакета сложным образом зависит от его длины, а, во-вторых, существенное время занимает переключение между режимами приём/передача. Таким образом, для получения надежных результатов следует начать с определения реальной скорости передачи на нижнем уровне. Измерения скорости производились для передачи данных, представленных в виде пакетов различной длины. Скорость определялась как в виде пропускной способности (bps – от англ. bytes per second, байт в секунду), так и количества пакетов в секунду (pps – от англ. packets per second). Каких-либо попыток оптимизации по скорости не производилось, так как задача заключается не в достижении максимальных показателей, а в проведении сравнительного анализа различных способов взаимодействия в радиоэфире. Результаты измерений представлены в виде графиков ниже. Исходные данные в виде таблицы вынесены в приложение А.

Эксперимент состоит из четырёх этапов – измерения скорости передачи данных на разных логических уровнях:

1. «Сырая» скорость канала.
2. Контроль целостности данных.
3. Гарантированная доставка.
4. Показатели DSM уровня.

Рассмотрим каждый из этапов подробнее.

«Сырая» скорость канала

Цель этапа — определить максимально возможную «сырую» скорость передачи данных на данном оборудовании.

Достаточно большое количество пакетов[1] различной длины отправлялось сериями[2] с одной платы на другую, при этом на приемной плате определялось количество принятых пакетов и время передачи. Зная количество отправленных пакетов, можно было убедиться в наличии связи между платами и надежности канала (с точки зрения потерь[3]), которая оказалась достаточно высокой (потери на уровне нескольких процентов). Результаты замеров представлены на рис. 3.6.

Как можно видеть (в данном случае точную величину определить по графику затруднительно, однако точные величины приводятся в приложении A), скорость изменяется от 1,800[4] bps для пакетов длиной 1 байт до 36,500 bps для достигнутых максимумов. Как и можно было предположить, максимумы оказались ярко выраженными и достигаются при размерах пакета кратных 32 байтам

[1]На данном уровне понятие программного пакета ещё отсутствует (оно вводится позже), и пакет имеется в виду «аппаратный». После отправки определённого количества байт выполняется вызов функции flush драйвера радиомодуля, что вызывает немедленную отправку накопленных данных невзирая на аппаратную буферизацию.

[2]Использовались серии из не менее чем 100 пакетов и не менее чем 10 килобайт – опытным путём было обнаружено, что данные условия позволяют добиться достаточного уровня повторяемости результатов с отклонением между двумя замерами около 2%.

[3]Так как контроль целостности на данном уровне ещё отсутствует, процент потерь определялся через объем принятой информации и его соответствие объему отправленной.

[4]Здесь и далее, для удобства восприятия, величины скорости приводятся в округленном до 100 bps виде.

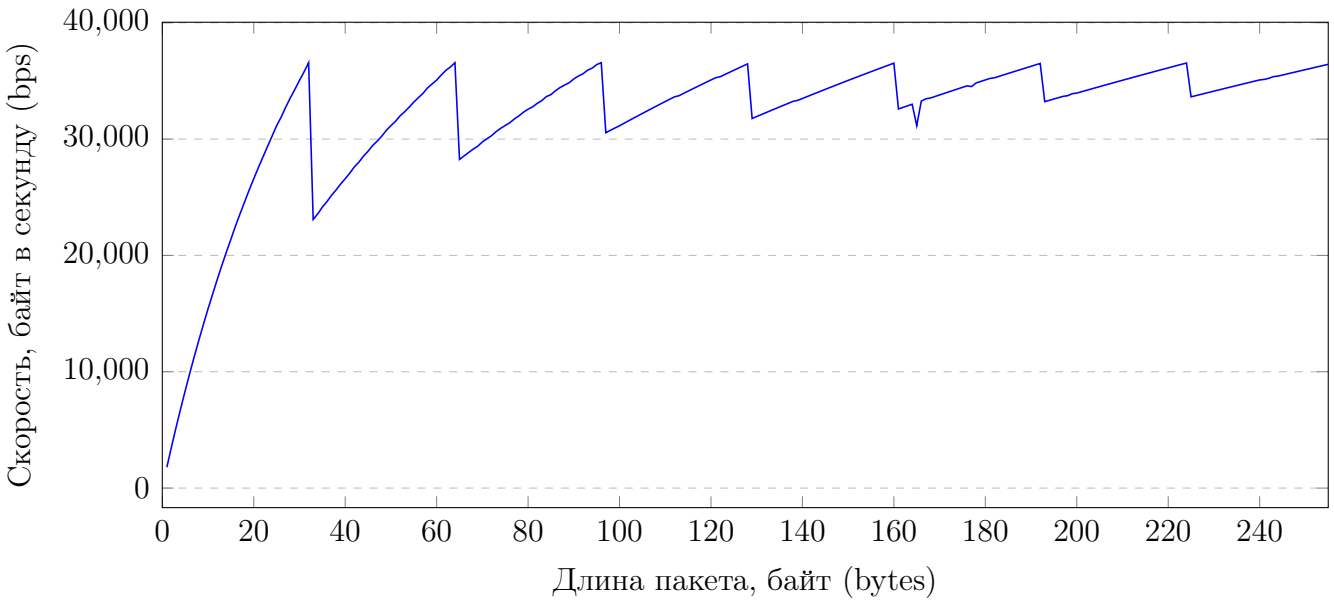


Рисунок 3.6 – «Сырая» скорость канала

– наблюдаемые на графике изломы соответствуют необходимости использовать большее количество аппаратных пакетов. При этом достигнутый при размере пакета в 32 байта максимум далее не превышается. Средняя скорость составила приблизительно 32,000 bps. Если же учитывать только пакеты размером более чем 16 байт, средняя скорость составит 33,400 bps. По мере увеличения длины пакета разброс между минимальной и максимальной скоростью ожидаемо сокращается.

На графике наблюдается падение скорости на отметке 165 байт, а также, гораздо менее заметное, при длине пакета 177 байт (точные величины можно найти в приложении А). Данные «отклонения», по всей видимости, обусловлены возникновением помех в эфире при передаче соответствующих серий (повторные эксперименты выявляют аналогичные «всплески», возникающие на разных участках графика без видимой закономерности).

Контроль целостности данных

На данном этапе к пакетам была добавлена служебная информация, позволяющая контролировать целостность данных (на основе `CRC32`), а также опре-

делять начало пакета в потоке данных и его размер[1]. Результаты измерений отражены на рис. 3.7.

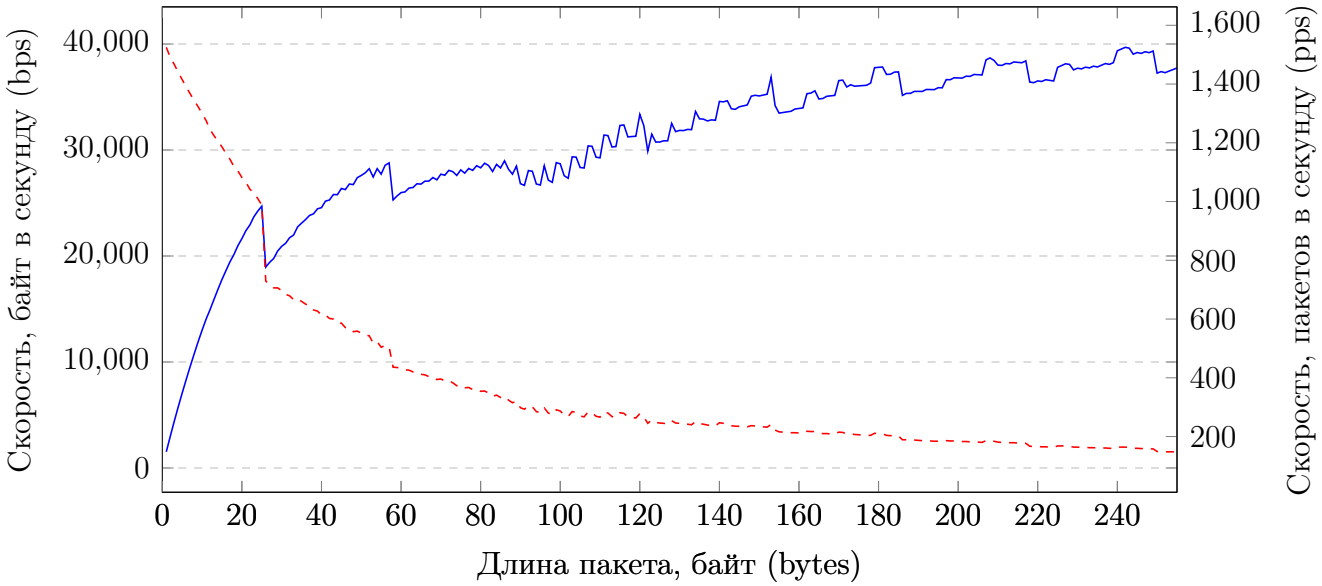


Рисунок 3.7 – Скорость с контролем целостности данных

Объем указанной выше служебной информации составил 7 байт. Таким образом, передача пакета длиной 1 байт эквивалентна 8 байтам нижнего уровня (рассмотренного выше). Например, пакет в 5 байт соответствует пакету 5+7=12 байт «сырых» данных. Для пакетов 12 байт на первом этапе эксперимента определена скорость 17,900 bps. Следовательно, для полезной нагрузки остается полоса в 5*(17,900/12)=7,500 bps. Это хорошо соответствует результатам измерений: 7,100 bps для пакетов 5 байт[2].

Однако максимально достижимая на данном логическом уровне скорость неожиданно оказалась несколько выше максимальной скорости предыдущего уровня. Дополнительное исследование показало, что причина заключается в

[1]Соответственно, начиная с данного этапа, замеряется не количество байт, а количество пакетов. Количество же байт выводится простым перемножением количества принятых пакетов на размер пакета в текущем эксперименте.

[2]Можно заметить, что размер пакета в эксперименте варьируется от 1 до 255 байт – верхняя граница не уменьшена на объем системной информации. Это связано с тем, что поле «размер пакета» (как и остальные поля текущего уровня) входит в данную системную информацию, и, соответственно, под «полезные» данные остаётся именно 255 байт в максимуме. Последующие уровни протоколов уже потребуют снижения верхней границы, так как размещают свои служебные данные в составе «пользовательских» данных текущего уровня.

накладных расходах на обращение к функции выборки поступающих данных из драйвера радио-модуля. На предыдущем этапе у нас не было возможности определить количество ожидаемых байт, и байты извлекались по одному. На текущем этапе среди служебной информации имеется и поле размера пакета, что позволяет выполнить выборку нужного количества байт за один раз. Соответственно, чем размер пакета больше, тем обнаруженный эффект становится более заметен.

В целом эксперимент подтвердил результаты предыдущего этапа, а также позволил исследовать канал на надежность в смысле искажения данных, которая также оказалась на уровне двух процентов.

Поскольку на данном этапе мы уже работаем с реальными программными пакетами, то скорость в байтах можно отразить и в виде скорости в пакетах – соответствующий график для наглядности также представлен на рис. 3.7 (в виде пунктирной линии). Очевидно, чем меньше по размеру пакет, тем быстрее он должен передаваться, но и тем меньшую полезную нагрузку нести – данное ожидание воплощается в предсказуемом убывании пунктирного графика.

Примечательным является возникновение небольших периодических колебаний скорости передачи, особенно хорошо заметных в районе длины пакета 100 байт. Подобные волны наблюдаются и далее, хотя их частота и регулярность постепенно сходят на нет. Если проанализировать самое яркое проявление эффекта в диапазоне 100-120 байт, то хорошо заметна четкая последовательность минимумов и максимумов, которые отстоят друг от друга на расстоянии 2 байта. Это можно объяснить, например, особенностями работы аппаратуры или драйвера нижнего уровня при разном выравнивании длины пакета относительно машинного слова. Возможны и другие причины, вопрос, очевидно, требует дополнительных исследований, однако, ввиду несущественного влияния данной «аномалии», её исследование вынесено за рамки данной работы.

Гарантированная доставка

Данный этап посвящён измерению производительности системы гарантированной доставки пакетов (аналог протокола TCP для широковещательного режима). В дополнение к возможностям второго этапа, обеспечивающего целостность данных, в данном случае осуществлялась отправка подтверждений о получении очередного пакета, а потерянные пакеты отправлялись повторно. Результаты измерений представлены на рис. 3.8.

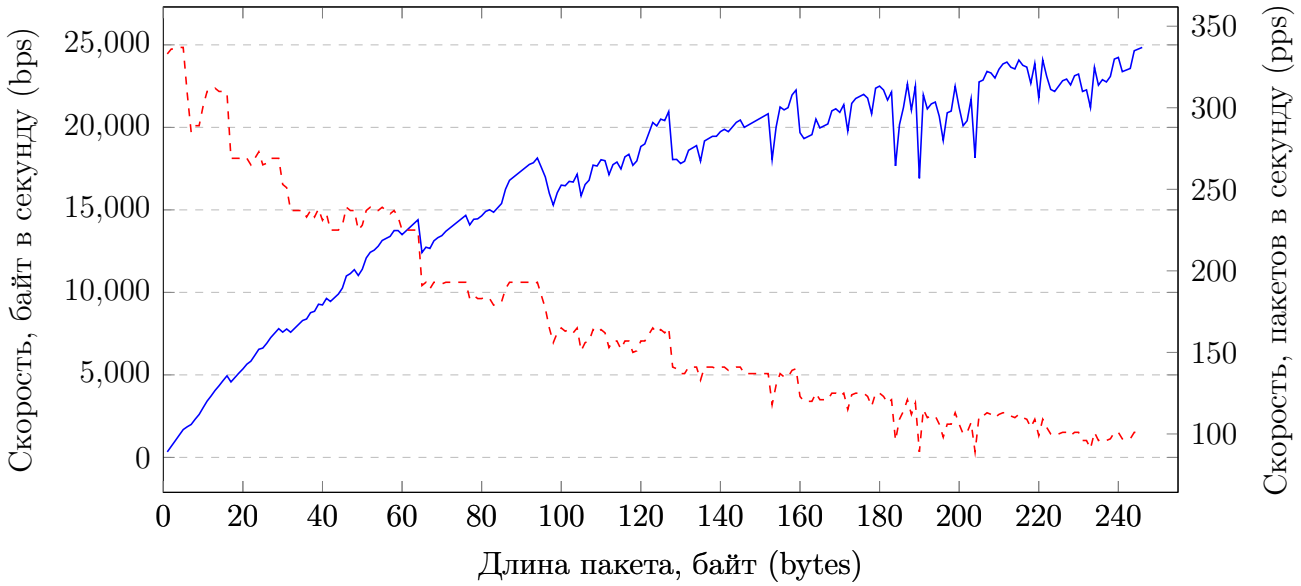


Рисунок 3.8 – Скорость с гарантированной доставкой

В связи с существенным ростом нагрузки на канал показатели ожидаемо снизились – примерно до 20,000 bps, с максимумом в районе 25,000 bps. Заметно, что график теряет стабильность по мере увеличения длины пакета. Предположительно это вызвано возрастанием вероятности его потери/повреждения и, соответственно, необходимостью его повторной передачи. Данную гипотезу можно проверить, наложив график скорости на график количества повторных посылок, однако в рамках текущего исследования данный эффект также оставим без внимания как несущественный.

Показатели DSM уровня

Последним этапом стало измерение показателей системы МАКС DSM. Уровнем ниже располагались механизмы, рассмотренные на предыдущих этапах. В соответствии с общей схемой эксперимента, проверялась работа на данных разного размера. При этом, осуществлялся эксклюзивный захват синхронизируемого объекта и, соответственно, глобальное обновление изменённых локально данных.

Важно отметить, что смысл понятий «байт в секунду» и «пакетов в секунду» на данном уровне существенно изменяется. Как было показано выше, протокол МАКС DSM подразумевает обмен множеством сообщений между несколькими узлами для выполнения любой операции над распределёнными данными. Во избежание путаницы, заменим понятие «пакетов в секунду» понятием «транзакций в секунду». Под транзакцией будем понимать совокупность всех сообщений между всеми узлами, необходимых для выполнения типичной операции уровня МАКС DSM – от входа в критическую секцию до выхода из неё включительно. Таким образом, в транзакцию попадут сообщения (между Клиентом, Сервером и Копией) для выполнения эксклюзивной блокировки и её снятия, сопровождающегося обновлением изменённых распределённых данных. Под «размером транзакции» будем понимать размер группы распределённых переменных – то есть это по-прежнему размер «полезных данных» в общем трафике, хотя понимание этих данных несколько изменилось.

Первичное представление результатов измерений можно увидеть на рис. 3.9.

Как и ожидалось, производительность существенно снизилась по сравнению с предыдущим этапом. Это объясняется уже изложенным выше фактом, что каждая транзакция требует нескольких операций уровня DSM (типа «выполнить эксклюзивную блокировку» и пр.), а каждая из операций приводит к необходимости отправки/приёма нескольких сообщений низлежащего уровня.

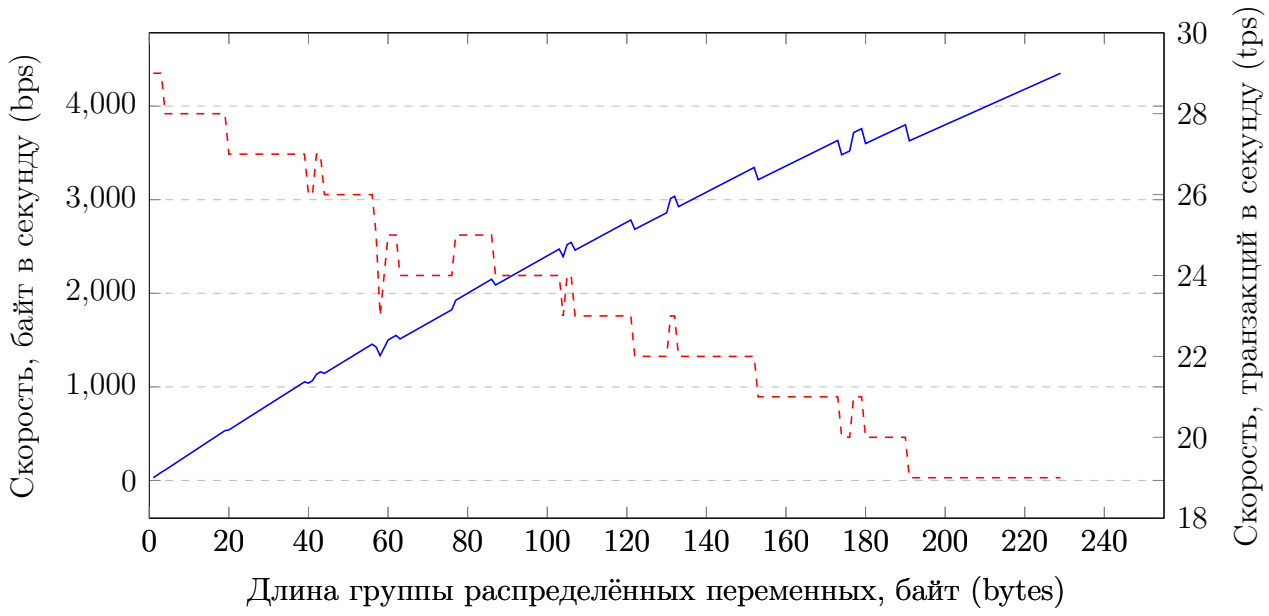


Рисунок 3.9 – Скорость DSM уровня

Скорость обмена «полезными» данными уверенно выходит на уровень около 4,000 bps, причём увеличение размера транзакции всегда положительно сказывается на общей пропускной способности системы.

Характер графика позволяет предположить, что выбранный масштаб на данном уровне становится недостаточен для понимания происходящего – действительно, количество транзакций в секунду измеряется единицами и десятками, и небольшие флуктуации могут сглаживаться дискретностью графика[1]. Приведем данные этого же эксперимента, указывая количество транзакций на 10-ти секундных интервалах – рис. 3.10.

Действительно, выраженная линейность с редкими скачкообразными изменениями теперь на графике отсутствует, однако из-за разных типов и размеров пакетов уровня DSM, а также сложного протокола взаимодействия узлов, локальные помехи в эфире, как и ожидалось, гораздо меньше влияют на общую скорость работы системы и график становится гораздо более линеен чем на предыдущем уровне протоколов. Кроме того, вероятно играет роль и тот факт, что на данном уровне между отправкой пакетов появляются задержки,

[1]А именно – использующимся округлением. Если за 1 секунду мы совершили 2.6 транзакции, используется значение 3, что при малом количестве транзакций оказывает существенное влияние на характер графика.

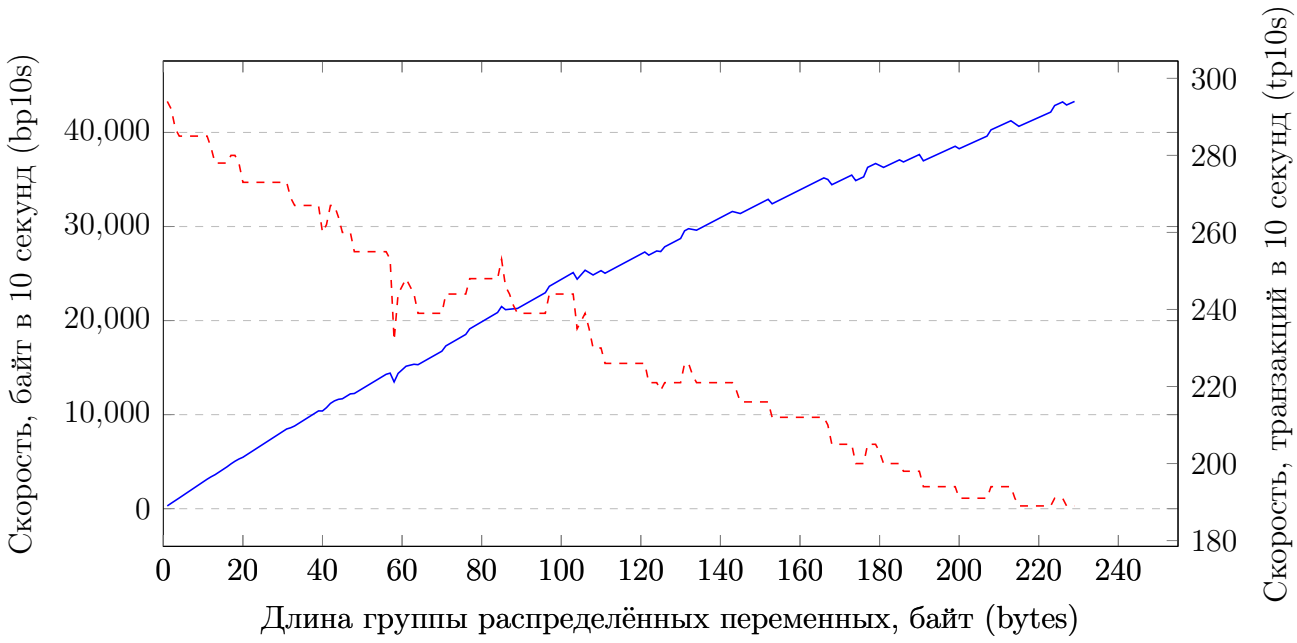


Рисунок 3.10 – Скорость DSM уровня (масштаб 10 секунд)

связанные с работой алгоритмов MAKC DSM, что также может привести к повышению стабильности. Данная гипотеза может быть проверена в будущих работах по развитию созданного решения.

3.7.2. Зависимость производительности от количества узлов

В разделе 2.1.1 мы определили требование, что создаваемая в рамках данного исследования система должна обеспечивать возможность совместной работы от одного до полутора десятков устройств. Однако в предыдущих разделах мы убедились в работоспособности созданного решения лишь для одного и двух узлов. Интерес также представляет не только факт работоспособности решения в максимальной конфигурации, но и зависимость производительности системы от количества устройств в ней. Текущий раздел посвящён поиску ответов на данные вопросы.

С целью минимизации финансовых и временных затрат было принято решение совместить испытания на реальном оборудовании с имитационным моделированием. Кроме сокращения трудозатрат данный подход позволяет в перспективе моделировать труднодостижимые на практике ситуации как по

количеству устройств так и по состоянию эфира. Частью программного имитационного комплекса стало штатное ПО МАКС DSM, дополненное моделью радиопередатчика и моделью эфира. Достоверность имитационной модели подтверждалась сравнением показателей имитационного и аппаратного комплексов в схожих условиях. С целью повышения достоверности имитационного ПО аппаратный комплекс был дополнен еще тремя устройствами, аналогичными описанным в разделе 3.7.1. Таким образом, программно-аппаратный стенд позволил проводить испытания для пяти устройств в максимуме, большее же число узлов в системе имитировалось программно.

Результаты испытаний имитационного комплекса на низлежащих относительно МАКС DSM сетевых уровнях в данном разделе опущены. Достаточно сказать, что были достигнуты сравнимые с полученными на реальном оборудовании результаты, за исключением несущественных артефактов, предположительно связанных с особенностями реализации радиопередатчика, прокомментированных ранее. Сосредоточимся на испытаниях уровня DSM. На рисунке 3.11 представлены результаты измерений производительности в зависимости от количества устройств в системе, произведённых на реальном оборудовании.

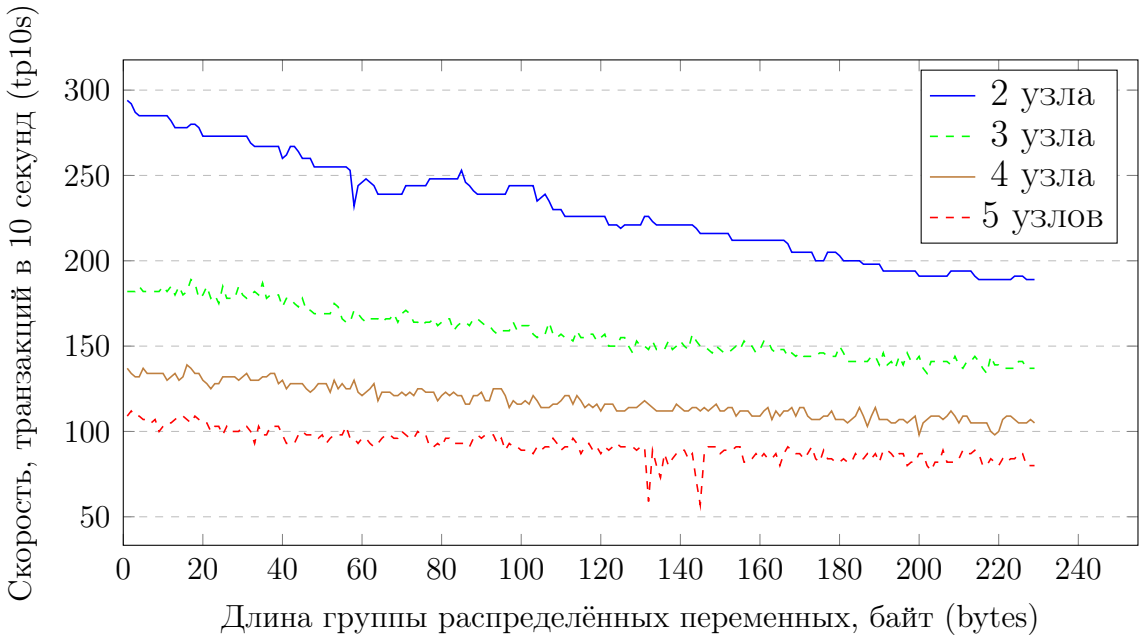


Рисунок 3.11 – Скорость DSM уровня (2–5 узлов)

Отметим, что результаты замеров для двух устройств в системе использованы те же, что были представлены пунктирной линией на рисунке 3.10 выше.

Графики на рис. 3.11 демонстрируют, что с ростом количества узлов в системе, её производительность ожидаемо снижается. Характер снижения – гиперболический (что в дальнейшем мы проверим на большем количестве устройств). Вместе с тем, мы знаем, что протокол МАКС DSM должен вносить околоконстантную задержку в производительность системы начина с момента, когда количество узлов в системе обеспечивает работу всех ролей (Клиент, Сервер, Копия) – то есть трёх. В связи с этим возникает гипотеза, что причина существенного влияния количества узлов на производительность системы – в низлежащем сетевом уровне. Проверим данную гипотезу, произведя аналогичные замеры для уровня «гарантированная доставка», рассмотренного ранее в конфигурации двух устройств. Результаты новых измерений изображены на рисунке 3.12.

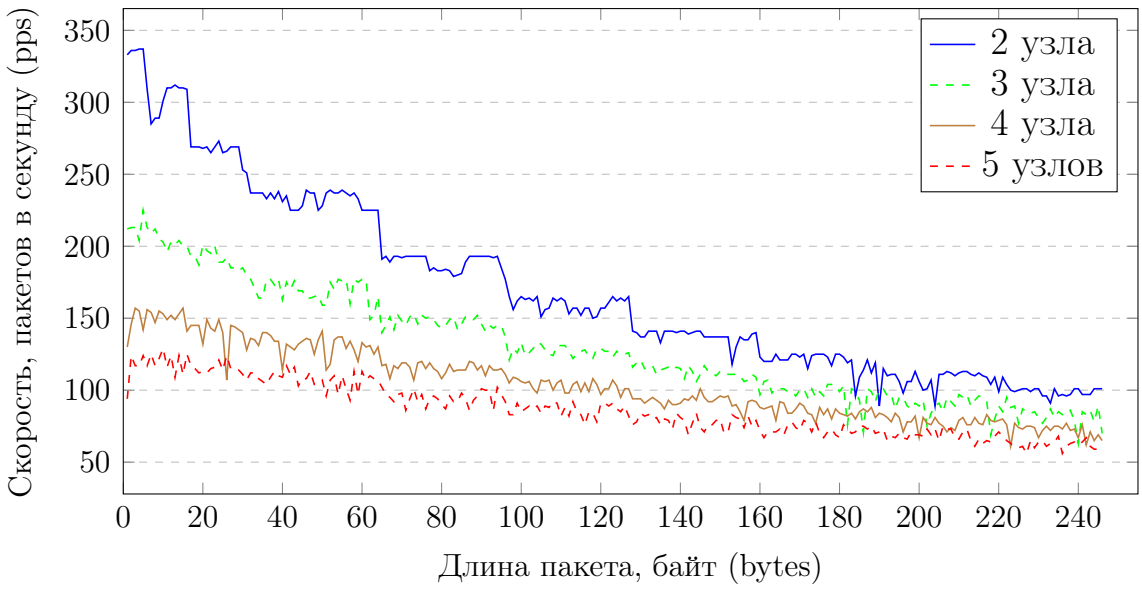


Рисунок 3.12 – Скорость уровня гарантированной доставки (2–5 узлов)

Гиперболическая зависимость производительности от количества устройств обусловлена реализацией данного уровня по методу из категории privilege-based algorithms [19] – специальный токен передается от устройства к устройству особым сообщением, и только узел, обладающий токеном в данный

момент, может вести передачу в эфире. Производительность подобной системы на данном уровне определяется формулой $1/N$, где N - количество узлов в системе[1]. Таким образом, для отправки пакета требуется полный оборот кольца из N устройств. Чем больше устройств, тем дольше данный оборот длится, но тем меньший вклад в величину задержки вносит включение в систему новых узлов.

При сравнении рисунков 3.11 и 3.12 может возникнуть вопрос – в связи с чем графики уровня DSM сходятся медленнее графиков низлежащего сетевого уровня. Имеется несколько причин:

1. Объем служебной информации на DSM уровне значительно выше (в среднем 8 пакетов низлежащего уровня на одну транзакцию, плюс, дополнительные служебные поля в каждом пакете). Это является основной причиной снижения производительности.
2. Постоянная составляющая общего замедления на DSM уровне также увеличена (только 3 из 8-ми пакетов транзакции включают в себя пользовательские данные). Это приводит к снижению влияния на производительность длины группы распределённых переменных.
3. Протокол МАКС DSM включает в себя операции вида «запрос-ответ», что требует дополнительно «оборота кольца» (в текущей реализации низлежащего уровня, основанной на методе передачи токена между узлами).
4. Низлежащий уровень «умеет» передавать сразу несколько пакетов с данными, но в случае DSM данная возможность задействуется только в случае одновременной работы с несколькими группами разделяемых переменных, что не происходит в имеющихся тестах производительности.

Таким образом полученные на двух последних рисунках графики соответствуют нашим ожиданиям. Производительность же МАКС DSM составляет,

[1]Вклад других компонент протокола, например, необходимость подтверждения приёма сообщений, мы для наглядности опускаем.

округленно, $X/10$, где X – скорость низлежащего сетевого уровня. Полученное на практике значение хорошо согласуется с нашим представлением о производительности, основанном на логике протокола MAKC DSM – в разделе 3.3.1 было показано, что эксклюзивная блокировка реализуется восемью сообщениями, поэтому восьмикратное замедление относительно низлежащего уровня неизбежно. Реальное замедление оказалось несколько выше, что объясняется теми же причинами, что были указаны выше в качестве влияющих на характер сходимости графиков.

Остаётся открытым вопрос производительности в случае присутствия в системе полутора десятка устройств. Как было описано выше (см. раздел 3.7.2), для проведения данного вида измерений использовалось моделирование. Результаты представлены на рисунке 3.13.

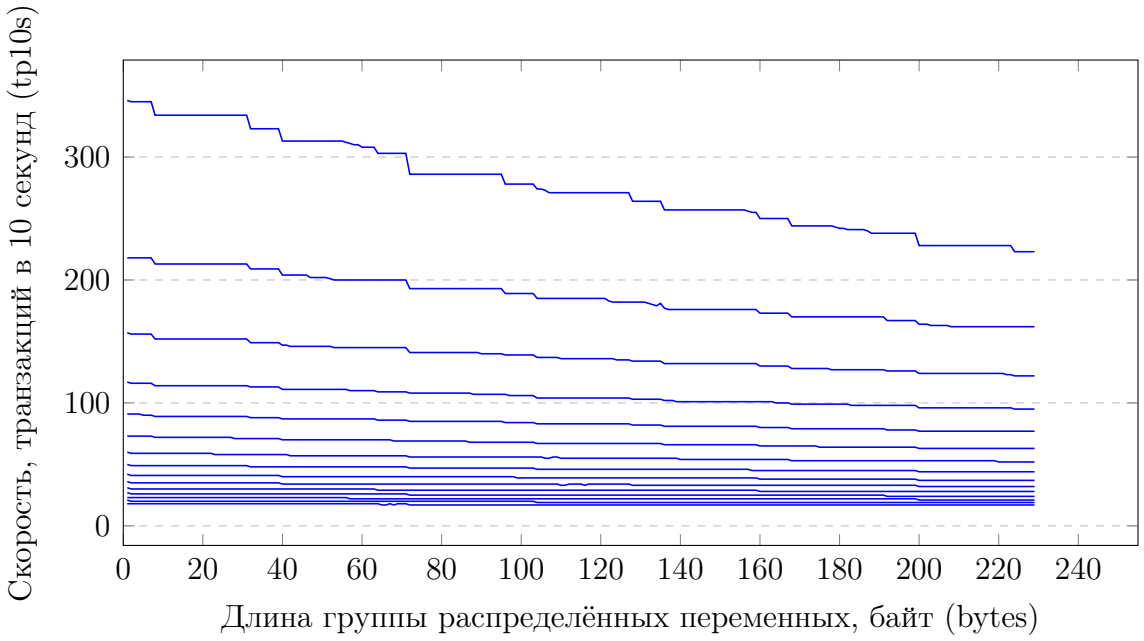


Рисунок 3.13 – Скорость DSM уровня (2–16 узлов)

При проведении данных замеров в модели была отключена имитация зашумлённости эфира, так как характер графиков в этом случае проявляется наиболее явно. Результирующие графики изображены в едином стиле, отсутствуют их расшифровка – так как в данном случае нам нет необходимости анализировать графики по отдельности. Чем ниже расположен график, тем

больше устройств присутствовало в системе, изменяясь от 2 до 16. Данные графики подтверждают высказанные ранее предположения и позволяют обнаружить нижнюю границу производительности, которая составила 17 транзакций в 10 секунд для системы из 16-ти устройств и размере группы распределённых переменных в 229 байт, что соответствует эффективной производительности 389 байт в секунду.

Нижняя граница производительности может показаться чрезмерно низкой, однако данная граница интересна лишь в смысле прикладного использования созданного решения, включая все сетевые уровни, где наибольший вклад в снижение производительности вносит уровень гарантированной доставки сообщений. Текущая реализация данного уровня (основанная на механизме круговой передачи токена) наиболее эффективна для систем, в которых каждый из узлов одинаково активен в эфире. В системах, где активных узлов меньше, более эффективны будут другие реализации. Поскольку в рамках данного исследования рассматривается только DSM уровень, его производительность следует оценивать относительно низлежащего сетевого уровня. Снижение производительности с переходом на уровень МАКС DSM, как было показано выше, приблизительно десятикратное.

3.8. Выводы

В данной главе была описана реализация прикладного интерфейса, принципы системы сообщений и ключевые фрагменты протокола, рассмотрена реализация отказоустойчивости, а также представлен пример программы, функционирующей на нескольких устройствах и демонстрирующей все возможности созданной системы. В завершение был рассмотрен испытательный стенд и приведены результаты измерений производительности решения.

Использование техники метапрограммирования (реализованной посредством языка Си++ и директив препроцессора, что отличается от ранее при-

менявшихся в DSM системах подходах), позволило достичь сочетания в одном решении качеств, ранее совместно не встречавшихся: простоты в использовании, проверки корректности на этапе компиляции и отсутствия зависимости от специфичных компонентов среды разработки (таких как специализированный для конкретной DSM системы компилятор или препроцессор). Ожидается, что данные свойства будут способствовать широкому распространению созданного решения.

Система ролей позволила выстроить лаконичный протокол обмена сообщениями, что повышает надежность решения. Вместе с принципом атомарности высокоуровневых сообщений, это позволило снизить сложность реализации требования отказоустойчивости. Отказоустойчивость реализуется рассмотрением всех возможных ситуаций выхода из строя устройств любой роли, в том числе одновременного выхода из строя нескольких устройств. Для каждой ситуации предусмотрена процедура восстановления системы. Хотя восстановление системы после сбоя отдельного узла или нескольких узлов происходит достаточно быстро, следует отметить, что в текущей реализации не обрабатывается ситуация, когда сбой происходит в процессе восстановления системы от предыдущего сбоя (каскадный сбой) – данный недостаток необходимо вынести на дальнейшую проработку вне рамок текущего исследования.

Была рассмотрена экспериментальная распределённая программа, функционирующая на нескольких устройствах и выполняющая некоторую вычислительную задачу. Было показано, что благодаря MAKC DSM, исходный код программы получился крайне лаконичен, но, несмотря на это, обеспечил синхронную работу нескольких устройств и работоспособность системы в случае выключения отдельных узлов. Таким образом, работоспособность созданного решения была доказана.

В заключение были проведены измерения – как на реальном, специально созданном оборудовании (для системы от двух до пяти устройств), так и на модели (от двух до 16 устройств). Для более глубокого понимания происхо-

дящих в системе процессов и более качественной интерпретации результатов, измерения были проведены для всех логических уровней сетевых протоколов. Округлённо, введение уровня МАКС DSM снижает производительность системы в десять раз (относительно низлежащего сетевого уровня), что в основном является следствием количества сообщений, необходимых для выполнения отдельной операции (транзакции) данного уровня.

Заключение

Целью данного диссертационного исследования была разработка моделей, алгоритмов и программных средств, реализующих концепцию распределённой общей памяти для мультиагентных систем в IoT и позволяющих существенно упростить и ускорить создание прикладных решений в данной области. Заявленная цель была достигнута.

Итоги выполненного исследования. Основные результаты диссертационной работы заключаются в следующем.

1. Разработана усиленная модель консистентности по выходу, позволяющая гарантировать согласованность данных в различных узлах распределённой мультиагентной системы и отличающаяся от существующих тем, что сочетает в себе свойства нескольких других моделей, в своей совокупности позволяющие минимизировать время существования уникальных данных в системе и обеспечить высокий уровень устойчивости к непреднамеренным ошибкам прикладного программиста, вместе с тем, не наследуя свойств, избыточных для обозначенной сферы применения.

2. Разработан алгоритм организации узлов мультиагентной системы в само-восстанавливающуюся структуру, устойчивую к выходу из строя отдельных узлов. Алгоритм основан на концепции ролей – каждый узел системы исполняет некоторую роль, динамически сменяющуюся в случае изменений в конфигурации системы. Проработаны возможные аварийные ситуации, способы их выявления и восстановления системы без потери функциональности и данных. Также определена ситуация, в которой восстановление системы в текущей реализации не обеспечивается (каскадный отказ нескольких узлов).

3. Разработаны принципы программного интерфейса для прикладного взаимодействия с созданным механизмом реализации концепции распределённой памяти. При формировании принципов были учтены недостатки прошлых решений, препятствующие широкому распространению соответствующих систем.

4. Разработано алгоритмическое и программное обеспечение, реализующее концепцию распределённой общей памяти для мультиагентных систем в сфере IoT. Созданное решение позволяет существенно упростить задачу организации взаимодействия устройств в MAC, что показано на примере использования данного решения.

5. Создан экспериментальный программно-аппаратный стенд из пяти устройств. С его помощью собраны характеристики разработанного решения для конфигураций от двух до пяти устройств в системе. В дополнение к стенду создан программный имитационный комплекс, позволяющий предсказать характеристики решения в более широких пределах вариантов использования. Достигнута точность модели, обеспечивающая сравнимые характеристики модели и программно-аппаратного стенда при одинаковом количестве устройств. На модели произведены замеры для количества устройств от двух до шестнадцати.

Рекомендации по применению результатов работы. При применении результатов данной работы в научных исследованиях или на производстве необходимо учитывать следующие аспекты.

1. При переносе МАКС DSM на другие платформы и среды разработки необходим анализ возможностей компилятора, используемого в конкретной ситуации. Система МАКС DSM реализована на языке C++ с использованием возможностей языка, разрешенных стандартом ISO/IEC 14882:2003, однако в области встраиваемых решений поддержка даже распространённых стандартов может быть ограничена.

2. При использовании решения необходимо учитывать, что устойчивость к каскадным сбоям, когда очередной сбой происходит в процессе восстановления системы от предыдущего сбоя, в данный момент решением не обеспечивается.

3. При использовании разработанной системы в практических задачах с целью максимизации производительности следует настраивать величину таймаутов в алгоритме смены роли узлом в соответствии с характеристиками кон-

кретной сети.

Перспективы дальнейшей разработки темы. В ходе проведения данного исследования было выявлено несколько моментов, нуждающихся в дополнительной проработке вне рамок текущей работы. Основные из них перечислены ниже.

1. Доработка решения в плане отказоустойчивости – необходимо предусмотреть устойчивость к каскадным сбоям, в ситуациях когда очередной сбой происходит в процессе восстановления системы от предыдущего сбоя.

2. Сбор метрик решения, связанных со скоростью восстановления системы после сбоев отдельных узлов.

3. Выявление причин аномалий, обнаруженных при проведении замеров на оборудовании. Описанные в данной работе аномалии выглядят незначительными, однако потенциально могут быть следствием неучтённых, но существенных явлений.

4. Оптимизация решения по производительности в стандартных режимах, сравнение вариантов по всем метрикам, включая метрики, связанные со скоростью восстановления системы после сбоев. К примеру, одним из способов существенной оптимизации может стать ликвидация роли «Копия» – необходимо оценить влияние данного решения на скорость восстановления, а также на возможности масштабирования решения.

5. Разработка моделей программирования, примеров для распространённых вариантов использования. Например, конечное ПО, предназначенное для группы устройств, каждое из которых порождает новые данные, может существенно отличаться от ПО для системы, в которой дополнительные устройства вводятся исключительно с целью резервирования основного узла. В обоих случаях МАКС DSM позволяет существенно упростить разработку конечного решения, однако для достижения оптимального результата нужно придерживаться принципов разработки, несколько различающихся для каждой из ситуаций.

Список сокращений и условных обозначений

АСКУЭ	—	Автоматизированная система коммерческого учёта электроэнергии
БПЛА	—	Беспилотный летательный аппарат
МАС	—	Мультиагентная система, многоагентная система (англ. multi-agent system)
ОС	—	Операционная система
ОСРВ	—	Операционная система реального времени
ОСРВ МАКС	—	ОСРВ для мультиагентных когерентных систем
ПО	—	Программное обеспечение
ЦП	—	Центральный процессор
API	—	Программный интерфейс приложения (англ. application programming interface)
DSM	—	Распределённая общая память (англ. distributed shared memory)
FIFO	—	Очередь сообщений типа «первый пришел – первый вышел» (англ. first in, first out)
IoT	—	Интернет вещей (англ. internet of things)
MMU	—	Блок управления памятью (англ. memory management unit)
OFDMA	—	Множественный доступ с ортогональным частотным разделением каналов (англ. orthogonal frequency division multiple access)
OSI	—	Базовая эталонная модель взаимодействия открытых систем (англ. open systems interconnection basic reference model)
POSIX	—	Переносимый интерфейс операционных систем (англ. portable operating system interface)

RPC — Удаленный вызов процедур (англ. remote procedure call)

TCP — Протокол управления передачей (англ. transmission control protocol)

UML — Унифицированный язык моделирования (англ. unified modeling language)

WiMax — Неофициальное название технологии, созданной по стандарту IEEE 802.16 для предоставления высокоскоростного беспроводного доступа к сети

Список литературы

1. Андреев, А. М. Многопроцессорные вычислительные системы: теоретический анализ, математические модели и применение [Текст] / А. М. Андреев, Г. П. Можаров, В. В. Сюзев. — Москва : Изд-во МГТУ им. Н. Э. Баумана, 2011. — ISBN: 9785703834398.
2. Бойко, П. В. МАКС DSM: Метаязык для организации взаимодействия групп автономных аппаратов [Текст] / П. В. Бойко // Навигация и гидрография. — 2017. — № 50. — С. 7–11.
3. Бойко, П. В. Подход к задаче обеспечения когерентности распределённых данных в мультиагентной системе [Текст] / П. В. Бойко // Инновации и инвестиции. — 2017. — № 2. — С. 206–208.
4. Бойко, П. В. Разработка прикладного API системы распределенной общей памяти МАКС DSM [Текст] / П. В. Бойко // Системный администратор. — 2017. — № 6. — С. 12–13.
5. Бойко, П. В. Распределённая общая память как способ организации взаимодействия в мультиагентных системах [Текст] / П. В. Бойко // Инновации и инвестиции. — 2017. — № 3. — С. 113–117.
6. Бойко, П. В. DSM в IoT: Усиленная модель консистентности по выходу [Текст] / П. В. Бойко // Инновации и инвестиции. — 2017. — № 7. — С. 134–136.
7. Бойко, П. В. ОСРВ МАКС (операционная система реального времени для мультиагентных когерентных систем) [Текст] / П. В. Бойко. — [Б. м.] : Свидетельство о государственной регистрации программы для ЭВМ № 2016617143, 28.06.2016 (Роспатент).
8. Бороздин, А. С. Гетерогенная система на основе сигнального процессора и процессора ARM под управлением одной операционной системы [Текст] / А. С. Бороздин // Электроника: Наука, Технология, Бизнес. — 2017. — № 8. — С. 94–97.

9. Таненбаум, Э. С. Современные операционные системы. 4-е изд. [Текст] / Эндрю С Таненбаум, Бос Херберт. — [Б. м.] : Издательский дом «Питер», 2015. — ISBN: 9785496013956.
10. Тарасов, В. Б. От многоагентных систем к интеллектуальным организациям: философия, психология, информатика [Текст] / В. Б. Тарасов. — Москва : Эдиториал УРСС, 2002. — ISBN: 5836003300.
11. Adve, S. V. Memory models: A case for rethinking parallel languages and hardware [Text] / Sarita V. Adve, Hans-J. Boehm // Commun. ACM. — 2010. — August. — Vol. 53, no. 8. — P. 90–101. — Access mode: http://doi.acm.org/10.1145/1787234.1787255.
12. Bal, H. E. Orca: a language for parallel programming of distributed systems [Text] / H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum // IEEE Transactions on Software Engineering. — 1992. — Mar. — Vol. 18, no. 3. — P. 190–205.
13. Baratloo, A. CALYPSO: a novel software system for fault-tolerant parallel processing on distributed platforms [Text] / A. Baratloo, P. Dasgupta, Z. M. Kedem // Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing. — [S. l. : s. n.], 1995. — Aug. — P. 122–129.
14. Bennett, J. K. Munin: Distributed shared memory based on type-specific memory coherence [Text] / J. K. Bennett, J. B. Carter, W. Zwaenepoel // Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming. — PPOPP '90. — New York, NY, USA : ACM, 1990. — P. 168–176. — Access mode: http://doi.acm.org/10.1145/99163.99182.
15. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors [Text] : Rep. : CMU-CS-91-170 / Carnegie Mellon University ; Executor: Brian N Bershad, Matthew J Zekauskas : 1991. — Sept.
16. Bershad, B. N. The Midway distributed shared memory system [Text] /

B. N. Bershad, M. J. Zekauskas, W. A. Sawdon // Digest of Papers. Compcon Spring. — [S. l. : s. n.], 1993. — Feb. — P. 528–537.

17. Carter, J. B. Design of the Munin distributed shared memory system [Text] / John B Carter // Journal of Parallel and Distributed Computing. — 1995. — Vol. 29, no. 2. — P. 219–227.
18. Carter, J. B. Implementation and performance of Munin [Text] / John B. Carter, John K. Bennett, Willy Zwaenepoel // Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles. — SOSP '91. — New York, NY, USA : ACM, 1991. — P. 152–164. — Access mode: http://doi.acm.org/10.1145/121132.121159.
19. Défago, X. Total order broadcast and multicast algorithms: Taxonomy and survey [Text] / Xavier Défago, André Schiper, Péter Urbán // ACM Comput. Surv. — 2004. — Dec. — Vol. 36, no. 4. — P. 372–421. — Access mode: http://doi.acm.org/10.1145/1041680.1041682.
20. Dubois, M. Memory access buffering in multiprocessors [Text] / M. Dubois, C. Scheurich, F. Briggs // Proceedings of the 13th Annual International Symposium on Computer Architecture. — ISCA '86. — Los Alamitos, CA, USA : IEEE Computer Society Press, 1986. — P. 434–442. — Access mode: http://dl.acm.org/citation.cfm?id=17407.17406.
21. Engelmore, R. Blackboard systems [Text] / Robert Engelmore, Tony Morgan. The insight series in artificial intelligence. — Wokingham (G.B.) : Addison-Wesley, 1988. — ISBN: 0201174316. — Includes index. Access mode: http://opac.inria.fr/record=b1080316.
22. Fine-grained mobility in the emerald system [Text] / E. Jul, H. Levy, N. Hutchinson, A. Black // SIGOPS Oper. Syst. Rev. — 1987. — Nov. — Vol. 21, no. 5. — P. 105–106. — Access mode: http://doi.acm.org/10.1145/37499.37511.
23. Fleisch, B. Mirage: A coherent distributed shared memory design [Text] / B. Fleisch, G. Popek // SIGOPS Oper. Syst. Rev. — 1989. — Nov. — Vol. 23,

no. 5. — P. 211–223. — Access mode: http://doi.acm.org/10.1145/74851.74871.

24. Gelernter, D. Generative communication in Linda [Text] / David Gelernter // ACM Trans. Program. Lang. Syst. — 1985. — Jan. — Vol. 7, no. 1. — P. 80–112. — Access mode: http://doi.acm.org/10.1145/2363.2433.
25. Cache consistency and sequential consistency [Text] : Rep. : 61 / SCI Committee ; Executor: James R Goodman : 1989. — Mar.
26. Grappa: A latency-tolerant runtime for large-scale irregular applications [Text] / Jacob Nelson, Brandon Holt, Brandon Myers [et al.] // International Workshop on Rack-Scale Computing (WRSC w/EuroSys). — [S. l. : s. n.], 2014.
27. Heterogeneous distributed shared memory [Text] / S. Zhou, M. Stumm, K. Li, D. Wortman // IEEE Transactions on Parallel and Distributed Systems. — 1992. — Sep. — Vol. 3, no. 5. — P. 540–554.
28. Hu, W. JIAJIA: A software DSM system based on a new cache coherence protocol [Text] / Weiwu Hu, Weisong Shi, Zhimin Tang // International Conference on High-Performance Computing and Networking / Springer. — [S. l. : s. n.], 1999. — P. 461–472.
29. Hutto, P. W. Slow memory: weakening consistency to enhance concurrency in distributed shared memories [Text] / P. W. Hutto, M. Ahamad // Proceedings of the 10th International Conference on Distributed Computing Systems. — [S. l. : s. n.], 1990. — May. — P. 302–309.
30. Keleher, P. Lazy release consistency for software distributed shared memory [Text] / Pete Keleher, Alan L. Cox, Willy Zwaenepoel // SIGARCH Comput. Archit. News. — 1992. — April. — Vol. 20, no. 2. — P. 13–21. — Access mode: http://doi.acm.org/10.1145/146628.139676.
31. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs [Text] / L. Lamport // Computers, IEEE Transactions on Computers. — 1979. — Sept. — Vol. C-28, no. 9. — P. 690–691.

32. Shared virtual memory on loosely coupled multiprocessors [Text] : Rep. / Yale Univ., New Haven, CT (USA) ; Executor: Kai Li : 1986.
33. Li, K. Memory coherence in shared virtual memory systems [Text] / Kai Li, Paul Hudak // ACM Trans. Comput. Syst. — 1989. — November. — Vol. 7, no. 4. — P. 321–359. — Access mode: http://doi.acm.org/10.1145/75104.75105.
34. Lipton, R. J. PRAM: A scalable shared memory [Text] / Richard J Lipton, Jonathan S Sandberg. — [S. l.] : Princeton University, Department of Computer Science, 1988.
35. Memory consistency and event ordering in scalable shared-memory multiprocessors [Text] / Kourosh Gharachorloo, Daniel Lenoski, James Laudon [et al.] // SIGARCH Comput. Archit. News. — 1990. — May. — Vol. 18, no. 2SI. — P. 15–26. — Access mode: http://doi.acm.org/10.1145/325096.325102.
36. Mosberger, D. Memory consistency models [Text] / David Mosberger // ACM SIGOPS Operating Systems Review. — 1993. — Vol. 27, no. 1. — P. 18–26.
37. Multiagent coordination in microgrids via wireless networks [Text] / Hao Liang, Atef Abdrabou, Bong Jun Choi [et al.] // IEEE Wireless Communications. — 2012. — Vol. 19, no. 3. — P. 14–22.
38. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence [Text] / Ed. by G. Weiss. — [S. l.] : The MIT Press, 1999. — ISBN: 0262232030.
39. Naplus: a software distributed shared memory for virtual clusters in the cloud [Text] / Lingfang Zeng, Yang Wang, Kenneth B Kent, Ziliang Xiao // Software: Practice and Experience. — 2017.
40. Protic, J. Distributed shared memory: concepts and systems [Text] / J. Protic, M. Tomasevic, V. Milutinovic // IEEE Parallel Distributed Technology: Systems Applications. — 1996. — Summer. — Vol. 4, no. 2. — P. 63–71.
41. Shoham, Y. Multiagent Systems: Algorithmic, GameTheoretic, and Logical

Foundations [Text] / Y. Shoham, K. LeytonBrown. — [S. l.] : Cambridge University Press, 2008. — ISBN: 0521899435.

42. Steen, M. v. Distributed Systems (3rd Edition) [Text] / Maarten van Steen, Andrew S. Tanenbaum. — [S. l.] : CreateSpace Independent Publishing Platform, 2017. — ISBN: 1543057381.
43. Steinke, R. C. A unified theory of shared memory consistency [Text] / Robert C Steinke, Gary J Nutt // Journal of the ACM (JACM). — 2004. — Vol. 51, no. 5. — P. 800–849.
44. Stumm, M. Algorithms implementing distributed shared memory [Text] / M. Stumm, S. Zhou // Computer. — 1990. — May. — Vol. 23, no. 5. — P. 54–64.
45. Stumm, M. Fault tolerant distributed shared memory algorithms [Text] / M. Stumm, Songnian Zhou // Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990. — [S. l. : s. n.], 1990. — Dec. — P. 719–724.
46. Tanenbaum, A. S. Distributed operating systems [Text] / Andrew S Tanenbaum. — [S. l.] : Pearson, 1994. — ISBN: 0132199084.
47. Treadmarks: Distributed shared memory on standard workstations and operating systems. [Text] / Peter J Keleher, Alan L Cox, Sandhya Dwarkadas, Willy Zwaenepoel // USENIX Winter. — Vol. 1994. — [S. l. : s. n.], 1994. — P. 23–36.
48. TreadMarks: shared memory computing on networks of workstations [Text] / C. Amza, A. L. Cox, S. Dwarkadas [et al.] // Computer. — 1996. — Feb. — Vol. 29, no. 2. — P. 18–28.
49. View-based consistency and false sharing effect in distributed shared memory [Text] / Zhiyi Huang, Chengzheng Sun, M Purvis, Stephen Cranefield // ACM SIGOPS Operating Systems Review. — 2001. — Vol. 35, no. 2. — P. 51–60.
50. VODCA: View-oriented, distributed, cluster-based approach to parallel computing [Text] / Z. Huang, W. Chen, M. Purvis, W. Zheng // Cluster Comput-

ing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on. — Vol. 2. — [S. l. : s. n.], 2006. — May. — P. 15–15.

51. Voronoi coverage of nonconvex environments with a group of networked robots [Text] / Andreas Breitenmoser, Mac Schwager, JeanClaude Metzger [et al.] // In processing of the IEEE International Conference on Robotics and Automation (ICRA). — Anchorage, Alaska, USA : [s. n.], 2010. — P. 4982–4989.
52. Wooldridge, M. An Introduction to MultiAgent Systems - Second Edition [Text] / M. Wooldridge. — [S. l.] : Wiley, 2009. — ISBN: 0470519460.
53. Yu, X. Tardis: Time traveling coherence algorithm for distributed shared memory [Text] / X. Yu, S. Devadas // 2015 International Conference on Parallel Architecture and Compilation (PACT). — [S. l. : s. n.], 2015. — Oct. — P. 227–240.

Список иллюстративного материала

Список таблиц

Приложение А. Результаты измерений

В данном приложении в виде таблицы представлены исходные (числовые) данные произведённых измерений. Информация может быть использована для уточнения графиков представленных в главе 4, а также визуализации данных в альтернативном виде.

Расшифровка названий колонок приведена ниже. При этом суффикс «-b» означает «байт в секунду» (от англ. bytes), «-p» – «пакетов в секунду» (от англ. packets).

- b – Количество байт в одном пакете данных.
- r – «Сырая» скорость канала (от англ. raw).
- c – Скорость для протокола с контролем целостности данных (от англ. check).
- g – Скорость для протокола с гарантированной доставкой (от англ. guarantee).
- d – Скорость DSM уровня для режима эксклюзивного доступа (от англ. DSM).

Так как в использующихся протоколах размер пакета представлен одним байтом, максимально возможный размер пакета – 255 байт. Соответственно, количество строк в таблице ограничено именно этим числом. Каждый новый уровень стека протоколов (за исключением уровня с контролем целостности, см. примечание в разделе замеров данного уровня в главе 4), добавляет несколько байт служебной информации к каждому пакету исходных, «пользовательских» данных. Гарантированная доставка – дополнительно 9 байт, DSM – ещё 17 байт. В связи с этим, замеры для каждого более высокого уровня стека протоколов прекращаются на размере пакета, меньшего на соответствующее количество байт.

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
1	1,796	1,525	1,525	333	333	29	29	294	294
2	3,526	2,982	1,491	672	336	58	29	583	292
3	5,199	4,401	1,467	1,008	336	87	29	861	287
4	6,806	5,760	1,440	1,348	337	112	28	1,139	285
5	8,360	7,075	1,415	1,685	337	140	28	1,424	285
6	9,866	8,340	1,390	1,854	309	168	28	1,709	285
7	11,319	9,569	1,367	1,995	285	196	28	1,993	285
8	12,721	10,752	1,344	2,312	289	224	28	2,278	285
9	14,081	11,907	1,323	2,601	289	252	28	2,563	285
10	15,406	13,000	1,300	3,010	301	280	28	2,848	285
11	16,663	14,036	1,276	3,410	310	308	28	3,132	285
12	17,912	14,940	1,245	3,720	310	336	28	3,390	282
13	19,112	15,912	1,224	4,056	312	364	28	3,613	278
14	20,273	16,870	1,205	4,340	310	392	28	3,891	278
15	21,389	17,790	1,186	4,650	310	420	28	4,169	278
16	22,520	18,624	1,164	4,944	309	448	28	4,447	278
17	23,573	19,448	1,144	4,573	269	476	28	4,763	280
18	24,603	20,142	1,119	4,842	269	504	28	5,043	280
19	25,623	20,976	1,104	5,111	269	532	28	5,280	278
20	26,593	21,640	1,082	5,360	268	540	27	5,467	273
21	27,534	22,386	1,066	5,649	269	567	27	5,741	273
22	28,452	22,924	1,042	5,830	265	594	27	6,014	273
23	29,355	23,690	1,030	6,187	269	621	27	6,287	273
24	30,251	24,240	1,010	6,552	273	648	27	6,561	273
25	31,149	24,700	988	6,625	265	675	27	6,834	273
26	31,894	18,954	729	6,916	266	702	27	7,107	273
27	32,750	19,413	719	7,263	269	729	27	7,381	273
28	33,540	19,768	706	7,532	269	756	27	7,654	273
29	34,278	20,474	706	7,801	269	783	27	7,927	273
30	35,049	20,910	697	7,590	253	810	27	8,201	273
31	35,774	21,204	684	7,781	251	837	27	8,474	273
32	36,567	21,728	679	7,584	237	864	27	8,602	269
33	23,090	21,978	666	7,821	237	891	27	8,795	267
34	23,573	22,746	669	8,058	237	918	27	9,062	267
35	24,150	23,100	660	8,295	237	945	27	9,328	267
36	24,619	23,436	651	8,388	233	972	27	9,595	267
37	25,161	23,828	644	8,769	237	999	27	9,861	267
38	25,623	23,978	631	8,854	233	1,026	27	10,128	267

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
39	26,133	24,453	627	9,282	238	1,053	27	10,395	267
40	26,593	24,560	614	9,240	231	1,040	26	10,388	260
41	27,070	25,174	614	9,635	235	1,066	26	10,741	262
42	27,610	25,284	602	9,450	225	1,134	27	11,194	267
43	28,019	25,800	600	9,675	225	1,161	27	11,461	267
44	28,534	25,784	586	9,900	225	1,144	26	11,627	264
45	28,953	26,370	586	10,260	228	1,170	26	11,686	260
46	29,442	26,266	571	10,994	239	1,196	26	11,946	260
47	29,829	26,790	570	11,139	237	1,222	26	12,206	260
48	30,251	26,736	557	11,376	237	1,248	26	12,247	255
49	30,753	27,391	559	11,025	225	1,274	26	12,502	255
50	31,149	27,600	552	11,400	228	1,300	26	12,757	255
51	31,529	27,846	546	12,087	237	1,326	26	13,012	255
52	31,996	28,236	543	12,428	239	1,352	26	13,267	255
53	32,347	27,454	518	12,561	237	1,378	26	13,522	255
54	32,750	28,242	523	12,798	237	1,404	26	13,777	255
55	33,180	27,720	504	13,145	239	1,430	26	14,032	255
56	33,558	28,560	510	13,272	237	1,456	26	14,288	255
57	33,925	28,785	505	13,395	235	1,425	25	14,413	253
58	34,396	25,288	436	13,746	237	1,334	23	13,477	232
59	34,738	25,665	435	13,747	233	1,416	24	14,381	244
60	35,066	25,980	433	13,500	225	1,500	25	14,761	246
61	35,507	26,047	427	13,725	225	1,525	25	15,146	248
62	35,902	26,412	426	13,950	225	1,550	25	15,253	246
63	36,192	26,460	420	14,175	225	1,512	24	15,356	244
64	36,567	26,816	419	14,400	225	1,536	24	15,308	239
65	28,250	26,780	412	12,415	191	1,560	24	15,547	239
66	28,553	27,060	410	12,738	193	1,584	24	15,787	239
67	28,849	27,068	404	12,663	189	1,608	24	16,026	239
68	29,139	27,404	403	13,124	193	1,632	24	16,265	239
69	29,393	27,186	394	13,317	193	1,656	24	16,504	239
70	29,757	27,720	396	13,440	192	1,680	24	16,743	239
71	30,027	27,619	389	13,703	193	1,704	24	17,306	244
72	30,289	28,080	390	13,896	193	1,728	24	17,550	244
73	30,638	27,959	383	14,089	193	1,752	24	17,793	244
74	30,925	27,602	373	14,282	193	1,776	24	18,037	244
75	31,168	28,125	375	14,475	193	1,800	24	18,281	244
76	31,403	27,816	366	14,668	193	1,824	24	18,525	244

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
77	31,731	28,259	367	14,091	183	1,925	25	19,119	248
78	31,996	28,080	360	14,430	185	1,950	25	19,368	248
79	32,314	28,519	361	14,457	183	1,975	25	19,616	248
80	32,570	28,320	354	14,640	183	2,000	25	19,864	248
81	32,769	28,755	355	14,904	184	2,025	25	20,112	248
82	33,070	28,536	348	15,006	183	2,050	25	20,361	248
83	33,307	27,971	337	14,857	179	2,075	25	20,609	248
84	33,653	28,644	341	15,120	180	2,100	25	20,857	248
85	33,823	28,305	333	15,385	181	2,125	25	21,493	253
86	34,160	28,982	337	16,254	189	2,150	25	21,158	246
87	34,434	28,188	324	16,791	193	2,088	24	21,206	244
88	34,644	27,720	315	16,984	193	2,112	24	21,249	241
89	34,849	28,480	320	17,177	193	2,136	24	21,288	239
90	35,172	26,820	298	17,370	193	2,160	24	21,527	239
91	35,421	26,663	293	17,563	193	2,184	24	21,766	239
92	35,609	28,060	305	17,756	193	2,208	24	22,005	239
93	35,920	27,993	301	17,856	192	2,232	24	22,245	239
94	36,097	26,790	285	18,142	193	2,256	24	22,484	239
95	36,401	26,695	281	17,575	185	2,280	24	22,723	239
96	36,567	28,512	297	16,992	177	2,304	24	22,962	239
97	30,550	27,160	280	16,005	165	2,328	24	23,643	244
98	30,753	26,950	275	15,288	156	2,352	24	23,887	244
99	30,953	28,809	291	16,038	162	2,376	24	24,131	244
100	31,149	28,700	287	16,500	165	2,400	24	24,375	244
101	31,363	27,573	273	16,463	163	2,424	24	24,618	244
102	31,575	27,336	268	16,728	164	2,448	24	24,862	244
103	31,787	29,355	285	16,686	162	2,472	24	25,106	244
104	31,996	29,328	282	17,160	165	2,392	23	24,402	235
105	32,205	28,350	270	15,855	151	2,520	24	24,876	237
106	32,412	28,302	267	16,536	156	2,544	24	25,354	239
107	32,618	30,388	284	16,799	157	2,461	23	25,106	235
108	32,823	30,348	281	17,712	164	2,484	23	24,848	230
109	33,027	29,321	269	17,658	162	2,507	23	25,079	230
110	33,229	29,260	266	18,040	164	2,530	23	25,309	230
111	33,430	31,413	283	17,982	162	2,553	23	25,033	226
112	33,630	31,360	280	17,136	153	2,576	23	25,258	226
113	33,728	30,284	268	17,741	157	2,599	23	25,484	226
114	33,925	30,324	266	17,898	157	2,622	23	25,710	226

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
115	34,121	32,315	281	17,480	152	2,645	23	25,935	226
116	34,316	32,364	279	18,212	157	2,668	23	26,161	226
117	34,510	31,239	267	18,369	157	2,691	23	26,386	226
118	34,702	31,270	265	17,700	150	2,714	23	26,612	226
119	34,894	31,297	263	17,969	151	2,737	23	26,837	226
120	35,084	33,360	278	18,840	157	2,760	23	27,063	226
121	35,274	32,307	267	18,997	157	2,783	23	27,288	226
122	35,359	29,890	245	19,642	161	2,684	22	26,958	221
123	35,546	31,488	256	20,295	165	2,706	22	27,179	221
124	35,731	30,752	248	20,088	162	2,728	22	27,400	221
125	35,916	30,750	246	20,500	164	2,750	22	27,336	219
126	36,100	30,870	245	20,412	162	2,772	22	27,842	221
127	36,282	30,861	243	20,955	165	2,794	22	28,063	221
128	36,464	32,512	254	18,048	141	2,816	22	28,284	221
129	31,770	31,734	246	18,060	140	2,838	22	28,505	221
130	31,938	31,850	245	17,810	137	2,860	22	28,726	221
131	32,105	31,833	243	17,947	137	3,013	23	29,543	226
132	32,271	31,944	242	18,612	141	3,036	23	29,769	226
133	32,436	31,920	240	18,753	141	2,926	22	29,691	223
134	32,600	33,634	251	18,894	141	2,948	22	29,609	221
135	32,764	32,940	244	17,955	133	2,970	22	29,830	221
136	32,927	32,912	242	19,176	141	2,992	22	30,051	221
137	33,089	32,743	239	19,317	141	3,014	22	30,272	221
138	33,250	32,844	238	19,458	141	3,036	22	30,493	221
139	33,330	32,804	236	19,460	140	3,058	22	30,714	221
140	33,490	34,580	247	19,740	141	3,080	22	30,935	221
141	33,649	34,545	245	19,881	141	3,102	22	31,156	221
142	33,807	34,648	244	19,738	139	3,124	22	31,377	221
143	33,964	33,891	237	20,020	140	3,146	22	31,598	221
144	34,120	33,840	235	20,304	141	3,168	22	31,491	219
145	34,276	34,075	235	20,445	141	3,190	22	31,379	216
146	34,431	34,164	234	20,002	137	3,212	22	31,596	216
147	34,585	34,251	233	20,139	137	3,234	22	31,812	216
148	34,739	35,076	237	20,276	137	3,256	22	32,029	216
149	34,892	35,164	236	20,413	137	3,278	22	32,245	216
150	35,044	35,100	234	20,550	137	3,300	22	32,462	216
151	35,195	35,183	233	20,687	137	3,322	22	32,678	216
152	35,346	35,264	232	20,824	137	3,344	22	32,894	216

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
153	35,496	36,873	241	18,054	118	3,213	21	32,414	212
154	35,645	34,188	222	20,020	130	3,234	21	32,626	212
155	35,794	33,480	216	21,235	137	3,255	21	32,837	212
156	35,942	33,540	215	21,060	135	3,276	21	33,049	212
157	36,089	33,598	214	21,195	135	3,297	21	33,261	212
158	36,236	33,654	213	21,962	139	3,318	21	33,473	212
159	36,382	33,867	213	22,260	140	3,339	21	33,685	212
160	36,527	33,920	212	19,680	123	3,360	21	33,897	212
161	32,589	33,971	211	19,320	120	3,381	21	34,108	212
162	32,725	35,316	218	19,440	120	3,402	21	34,320	212
163	32,860	35,371	217	19,560	120	3,423	21	34,532	212
164	32,995	35,588	217	20,500	125	3,444	21	34,744	212
165	31,152	34,815	211	19,965	121	3,465	21	34,956	212
166	33,264	34,860	210	20,086	121	3,486	21	35,168	212
167	33,464	35,070	210	20,207	121	3,507	21	34,999	210
168	33,530	35,112	209	21,000	125	3,528	21	34,443	205
169	33,663	35,152	208	21,125	125	3,549	21	34,648	205
170	33,795	36,550	215	20,910	123	3,570	21	34,853	205
171	33,926	36,594	214	21,375	125	3,591	21	35,058	205
172	34,057	35,948	209	19,780	115	3,612	21	35,263	205
173	34,187	36,157	209	21,452	124	3,633	21	35,468	205
174	34,317	36,018	207	21,750	125	3,480	20	34,881	200
175	34,446	36,050	206	21,875	125	3,500	20	35,081	200
176	34,575	36,080	205	22,000	125	3,520	20	35,282	200
177	34,515	36,108	204	21,771	123	3,717	21	36,289	205
178	34,831	36,312	204	20,826	117	3,738	21	36,494	205
179	34,958	37,769	211	22,375	125	3,759	21	36,699	205
180	35,085	37,800	210	22,500	125	3,600	20	36,494	203
181	35,212	37,829	209	22,263	123	3,620	20	36,284	200
182	35,269	37,128	204	21,658	119	3,640	20	36,484	200
183	35,394	37,149	203	22,143	121	3,660	20	36,685	200
184	35,519	37,352	203	17,664	96	3,680	20	36,885	200
185	35,643	37,370	202	20,165	109	3,700	20	37,086	200
186	35,767	35,154	189	21,204	114	3,720	20	36,863	198
187	35,890	35,343	189	22,627	121	3,740	20	37,061	198
188	36,013	35,344	188	21,056	112	3,760	20	37,259	198
189	36,135	35,532	188	22,491	119	3,780	20	37,457	198
190	36,257	35,530	187	16,910	89	3,800	20	37,655	198

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
191	36,379	35,526	186	21,965	115	3,629	19	36,983	194
192	36,500	35,712	186	21,120	110	3,648	19	37,177	194
193	33,216	35,705	185	21,423	111	3,667	19	37,371	194
194	33,331	35,696	184	21,534	111	3,686	19	37,564	194
195	33,445	35,880	184	20,670	106	3,705	19	37,758	194
196	33,559	35,868	183	19,208	98	3,724	19	37,951	194
197	33,673	36,642	186	20,882	106	3,743	19	38,145	194
198	33,729	36,630	185	20,988	106	3,762	19	38,339	194
199	33,899	36,815	185	22,487	113	3,781	19	38,532	194
200	33,954	36,800	184	21,200	106	3,800	19	38,270	191
201	34,066	36,783	183	20,100	100	3,819	19	38,462	191
202	34,177	36,966	183	20,402	101	3,838	19	38,653	191
203	34,288	36,946	182	21,721	107	3,857	19	38,844	191
204	34,399	37,128	182	18,156	89	3,876	19	39,036	191
205	34,510	37,105	181	22,755	111	3,895	19	39,227	191
206	34,620	37,080	180	22,866	111	3,914	19	39,419	191
207	34,729	38,502	186	23,391	113	3,933	19	39,610	191
208	34,839	38,688	186	23,296	112	3,952	19	40,275	194
209	34,948	38,456	184	22,990	110	3,971	19	40,469	194
210	35,056	38,010	181	23,520	112	3,990	19	40,662	194
211	35,165	37,980	180	23,843	113	4,009	19	40,856	194
212	35,272	38,160	180	23,956	113	4,028	19	41,050	194
213	35,380	38,127	179	23,643	111	4,047	19	41,243	194
214	35,487	38,306	179	23,540	110	4,066	19	40,949	191
215	35,594	38,270	178	24,080	112	4,085	19	40,651	189
216	35,700	38,232	177	23,760	110	4,104	19	40,840	189
217	35,806	38,409	177	23,653	109	4,123	19	41,029	189
218	35,912	36,406	167	22,672	104	4,142	19	41,218	189
219	36,018	36,354	166	23,871	109	4,161	19	41,407	189
220	36,123	36,520	166	21,780	99	4,180	19	41,596	189
221	36,227	36,465	165	24,089	109	4,199	19	41,785	189
222	36,332	36,630	165	23,088	104	4,218	19	41,974	189
223	36,436	36,572	164	22,300	100	4,237	19	42,164	189
224	36,539	36,512	163	22,176	99	4,256	19	42,863	191
225	33,630	37,800	168	22,500	100	4,275	19	43,054	191
226	33,729	37,968	168	22,826	101	4,294	19	43,246	191
227	33,828	38,136	168	22,927	101	4,313	19	42,920	189
228	33,927	38,076	167	22,572	99	4,332	19	43,109	189

b	r-b	c-b	c-p	g-b	g-p	d-b	d-p	d-b10	d-p10
229	34,025	37,556	164	23,129	101	4,351	19	43,298	189
230	34,123	37,720	164	23,230	101				
231	34,220	37,653	163	22,176	96				
232	34,318	37,816	163	22,272	96				
233	34,415	37,746	162	21,203	91				
234	34,511	37,908	162	23,634	101				
235	34,608	37,835	161	22,560	96				
236	34,704	37,996	161	22,892	97				
237	34,800	38,157	161	22,752	96				
238	34,895	38,080	160	23,086	97				
239	34,991	38,240	160	24,139	101				
240	35,086	39,360	164	24,240	101				
241	35,129	39,524	164	23,377	97				
242	35,224	39,688	164	23,474	97				
243	35,369	39,609	163	23,571	97				
244	35,412	39,040	160	24,644	101				
245	35,505	39,200	160	24,745	101				
246	35,599	39,114	159	24,846	101				
247	35,692	39,273	159						
248	35,784	39,184	158						
249	35,877	39,342	158						
250	35,969	37,250	149						
251	36,061	37,399	149						
252	36,153	37,296	148						
253	36,244	37,444	148						
254	36,336	37,592	148						
255	36,428	37,740	148						