© 2012 г. **С.В. ЗНАМЕНСКИЙ**, д-р физ.-мат. наук
(Институт программных систем им. А.К. Айламазяна РАН, Переславль-Залесский)

РЕТРОСПЕКТИВНАЯ ОСНОВА РАСПРЕДЕЛЁННОЙ ПАМЯТИ ДЛЯ ИЗМЕНЧИВОЙ ВЫЧИСЛИТЕЛЬНОЙ СРЕДЫ[1]

Ретроспективный подход не связан с какими-либо ограничениями на количество вычислительных узлов, особенностями оборудования или прикладной направленности задачи. Принципиальным ограничением эффективной применимости является лишь размер гранул параллелизма, при которых время параллельного вычисления должно быть больше, чем погрешность согласования физического времени в системе.
Для случая двух разделяемых узлов теоретически разобран пример, в котором ретроспективность обеспечивает сочетание высоких корректности и быстроты реакции при задержках и временных потерях связи, невозможное при стандартном подходе в силу известной теоремы CAP.
Описана методология использования ретроспективной памяти в качестве основы для безгранично перестраиваемого высокодоступного вычислительного сервиса для длительной эксплуатации.

RETROSPECTIVE BASED DISTRIBUTED MEMORY FOR CHANGING COMPUTING ENVIRONMENT / S.V. Znamenskij (Program Systems Institute of RAS Pereslavl-Zalessky Yaroslavl Region Russia, 152020, E-mail: svz@latex.pereslavl.ru).
Retrospective approach has no restrictions on the number of computing nodes, special hardware or application sphere. The only fundamental limitation is in parallelism granularity: the execution time must be greater than the physical time uncertainty in the sytem.
The case of two poor connected nodes is theoretically analyzed. Retrospective approach provides a combination of high correctness and availability in the case of delays and temporary network loss. In the standard approach it is impossible due to the well-known theorem CAP.
The new methodology of retrospective memory is presented as a basis for highly available infinitely adaptive computing services for the long run.

1. Введение

Целью работы является проработка архитектурных принципов высокоэффективных системы информационной поддержки сложного долговременного целенаправленного взаимодействия большого числа людей и вычислительных средств в будущем десятилетии.

[1]Работа выполнена при финансовой поддержке МОН.

Задачами такого рода систем будут разработка гладко обновляемой операционной системы и конструктивное разрешение содержательных социально-экономических и экологических проблем.

Имеется в виду, что мощность распределённой вычислительной системы в процессе бесперебойного обслуживания нарастает заменой устаревших кластеров на новые, которые могут принципиально отличаться по архитектуре и управлению. Речь о системе, которая не выключается и постоянно предоставляет внешний доступ к данным. Без остановок в ней же разрабатывается уникальное системное и прикладное программное обеспечение, отлавливаются и исправляются ошибки, сопоставляются результаты, полученные разными путями, оптимизируется выполнение задач, устраняются любые изолированные поломки.

Такая организация высокопроизводительного бесперебойного сервиса в долговременной перспективе является проблематичной: обновление оборудования не сводится к простой замене, а требует использования новых парадигм, качественно изменяются запросы и интерфейсы пользователей и средства разработки программного обеспечения.

Прогноз погоды даёт не единственный возможный пример будущей прикладной длительно функционирующей системы, в которой количественно и качественно растущие данные должны параллельно в тесном сотрудничестве обрабатываться в разных взаимосвязанных целях.

Повышению качества прогноза послужат реорганизация ввода информации от новых источников, наращивание вычислительных мощностей, совершенствование алгоритмов обработки и интерфейсов пользователя.

Единая информационно-вычислительная система должна обеспечивать не только вычисления по фиксированному алгоритму на основе заранее определённых структур входных данных, но и включение в обработку новых видов входной информации и новых алгоритмов обработки и совершенствование старых алгоритмов. Основой всего должна стать развёрнутая организация параллельных работ с общими данными на расширяющейся вычислительной основе. Единой организацией доступа к данным должны быть поддержаны работы по сопоставительному анализу разных подходов и алгоритмов обработки, выявлению возможностей ситуативного выбора наиболее адекватных алгоритмов обработки и архитектур локального оборудования. На этой же основе должны попутно развиваться и исследоваться алгоритмы интервальных оценок будущих погодных явлений.

На этом примере отчётливо видна потребность повторного использования полной информации о входных данных, результатах их обработки, использованных алгоритмах и конфигурации системы при вычислениях.

Видна на нём и потребность в длительно функционирующей системе с гибко изменяемой конфигурацией оборудования и программного обеспечения и организации работ. Длительное существование информационной системы связано с рисками принципиальной несовместимости оборудования. Так, устройства одноразовой записи WORM и flash-память, требующая периодической перезаписи во избежание потерь, могут эффективно использоваться лишь в разных парадигмах. Непримиримые несовместимости порождаются и при использовании процессоров разных типов, в том числе графических ускорителей [1]. Гибридные архитектуры привносят особую сложность, главным источником которой является «не дополнительный язык для программирования ускорителей, а наличие нескольких дополнительных уровней иерархии памяти, из-за кото-

рого сложность задания корректного и эффективного распределения и перемещения данных достигает критического уровня» [2].

Исследования, связанные с организацией высокопроизводительного бесперебойного сервиса в долговременной перспективе [3, 4, 5], не подсказывают ясного пути к решению поставленной задачи.

Особенности оборудования, программных интерфейсов или общей структуры управления не могут претендовать на архитектурную основу такой системы. Этой архитектурной основой становится ретроспективность: доступность в неискажённом виде всей ранее доступной информации, включая программные и пользовательские интерфейсы и сопроводительную документацию по указанию времени прежнего доступа.

2. Идея ретроспективности

Суть ретроспективной парадигмы в непризнании абстракции текущего состояния. На примере банковских систем это означает, что сделки совершаются не мгновенно, поэтому значение денег на счёте в момент незавершённой транзакции есть абстракция, не имеющая физического смысла. Поэтому не состояния счетов, а записи в истории изменений являются базовыми объектами. Точное значение на счету существует лишь для достаточно отдаленного прошлого, все транзакции из которого завершились.

Записи об изменении счетов только создаются и никогда не изменяются. Можно считать, что они никогда не исчезают. Поэтому ответ на любой разумный вопрос хорошо построенная система может дать быстро, и на логическом уровне в ней нет никакой конкуренции доступа и нужды в блокировках (записи ведь все разные), и неясно, действительно ли нужны откаты транзакций.

Использование старых данных опасно не потерей актуальности. В самом деле: если система работает быстро, то разница во времени ничтожна, а если время обработки значительно, то пока ответ по данным на момент запроса будет получен, момент запроса устареет на такое же время обработки.

Опасность в том, что обрабатывать старые данные можно лишь по их состоянию на *фиксированный* момент времени. Изменившееся значение противоречит прежнему.

Учитывая техническую сложность согласования времени с высокой точностью, такой подход требует, чтобы время выполнения подзадачи на отдельном узле ощутимо превосходило погрешность согласования времени в решающем задачу кластере. По-видимому, это ограничение не является практически значимым.

Мы приходим к требованию *ретроспективной согласованности данных информационной системы*:

1) в любой момент времени t в системе доступен прошедший *момент истины* $\mu(t) < t$, которым завершилась доступная история согласованных изменений данных системы;

2) для любого предыдущего момента $\tau < \mu(t)$ система возвращает согласованные реплики.

Разность $t - \mu(t)$ назовём *длительностью согласования*. При удачном разделении длительность согласования для части системы оказывается меньше, чем для целой системы.

Ретроспективная информационная система может не всегда иметь единую концептуальную модель. Более того, её концептуальная модель может изменяться не мгновенно и не одновременно во всех частях. Этим она принципиально отличается от *последова-*

тельных информационных систем, основанных на моделировании последовательности изменений глобального согласованного состояния системы.

Практически все существующие информационные системы *последовательные*. Это косвенно подтверждается общепринятостью доказанности теоремы CAP.

Использование старых данных опасно вовсе не потерей их актуальности.

Покажем, что ретроспективность позволяет резко повысить устойчивость распределённых систем к временным потерям и задержкам связи.

3. Устойчивость ретроспективных систем к задержкам и разделениям

Известная [7] своими небезупречными популяризациями *теорема* **CAP** [6], утверждает невозможность сочетания трёх свойств:

Consistency (*Согласованность*):
Ответы на запросы к системе логически непротиворечивы.

Availability (*Доступность*):
Ответ на любой запрос может быть получен незамедлительно.

Partition-tolerance (*Устойчивость к разделению*):
При восстановлении потерянной на время связи функциональность и внутренняя целостность восстанавливаются.

Формальное доказательство этой теоремы в [8] основано на прозрачной идее:

Как только в частях появились различия, то их доступность означает противоречие

Практику после разрыва связи с филиалом компании работа *должна* быть продолжена с адекватными ситуации ограничениями и после восстановления связи система *должна* полностью восстановиться без потерь информации, то логика доказательства очевидно противоречит потребностям практики.

Неприемлемость строго доказанного результата в науке встречается. Достаточно вспомнить классические безупречные доказательства невозможности рационального корня из числа 2, корней из отрицательных чисел и недифференцируемости функции Хевисайда. Их неприемлемость привела в своё время к развитию и широкому применению новых теорий, ставших основой современной математики — теории вещественного числа, теории комплексных чисел и теории обобщённых функций.

Рассмотрим возможность и целесообразность построения системы, сочетающей ретроспективность с *адекватностью* — безупречностью реакции на запросы и соразмерностью последствий временных поломок и сбоев.

3.1. Пробная задача

Банк имеет тысячу клиентов в двух городах A и B. Расчётное максимальное время задержки передачи данных между городами $T = 0.25$ секунды.

Города расположены так, что устранить эту задержки технически невозможно[2]

[2]При передаче со скоростью света через геостационарный спутник сигнал задерживается на 1/4 секунды. При недоступности геостационарного решения неизбежны задержки при переключениях на очередной спутник.

Для поддержки переводов денег со счёта на счёт банк нуждается в качественной информационной системе с по возможности быстрым временем отклика τ, рассчитанную на обработку не более десяти запросов в секунду с одновременной поддержкой ста сессий:

1) Результаты любых запросов гарантированно должны быть корректны и согласованы.

2) При любом обращении к сервису задержка обработки не должна превосходить τ для любого из следующих запросов:

а) Просмотр состояния счёта и истории его изменений.

б) Перевод имеющейся суммы на другой счёт.

в) Получение отчётов.

Для предельного упрощения задачи полагаем, что банк с наличностью не работает, кредитованием не занимается, вне городов никого не обслуживает, пользователи заведены с неизменными личными данными и для простоты не переезжают, а потерями и задержками внутри городов можно пренебречь.

3.2. Последовательный подход

Информация о текущем состоянии каждого счёта должна где-то храниться. Запрос о переводе на этот счёт из другого города, должен дойти и после этого ответ должен вернуться обратно, чтобы пользователь смог знать, что запрос принят. Поэтому реакция системы не может быть гарантирована быстрее, чем в течение $\tau = 2T = 0.5$ секунды.

3.3. Ретроспективный подход

В мегаполисах размещаются идентичные серверы. Каждый из них ведёт счета пользователей из своего мегаполиса.

Сервер отказывается выполнить перевод если денег на счету отправителя недостаточно. Если на счету достаточно денег, то сервер немедленно изменяет состояние локальных счетов и надёжно запоминает своё обязательство по возможности быстро выполнить перевод. Это должно делаться за ощутимо меньшее время, чем ожидаемое время поступления следующего запроса. Если повторный запрос от пользователя на перевод денег следует через доли миллисекунды, то это скорее повторное срабатывание и уж никак не осмысленные действия пользователя, поэтому гораздо безопаснее оставить лишь один из серии таких практически одновременных запросов.

После обработки серии запросов (например каждые пол-секунды) серверы передают друг другу полные списки накопившихся запросов и записей изменений в счетах и извещения о получении списков. Извещение о получении посылается только в том случае, когда пакет получен полностью и содержит момент времени, вплоть до которого все данные получены. Пакетная передача помогает гарантировать единый порядок и отсутствие пропусков в общей части информации серверов и корректно снизу оценить момент истины системы $\mu(t)$, вся история изменений вплоть до которого получена на обоих серверах.

Пользователь получает ответ на свой запрос с учётом реакции только ближайшего сервера. Реакция удалённого (например, уведомление о получении перевода в другой город) может быть учтена в отдельном последующем через секунду запросе (который может делаться автоматически либо клиентским приложением либо на основе

технологии comet, позволяющей локальному серверу самостоятельно передать такое уведомление в момент получения).

3.4. Сравнение по скорости и интерфейсу

Условия задачи не влекут никаких технических препятствий к получению быстрого (в течение десяти миллисекунд) ответа на запрос на основе имеющейся на ближайшем сервере информации. Повышение реактивности системы в сто раз здесь достигнуто по сути включением в реакцию системы на запрос пользователя только немедленно доступной информации.

Это позволяет обеспечить пользователя удобным интерфейсом, дающим возможность выбора, наблюдать за процессом обработки, любо спокойно выключить клиентское приложение либо заняться операциями с другими счетами.

Взаимодействие с внешней системой может породить сложную совместную транзакцию (например, перевод с одного счёта на другой в другой город, оттуда на третий и снова назад) и потери времени станут неизбежны. Для клиента, ожидающего поступлений на счёт, может быть создано клиентское приложение, ожидающее изменения состояния счёта для повторения запроса, отвергнутого из-за недостатка денег на счету.

Разумеется, при переводе денег упомянутая секунда задержки не составляет проблемы для клиента. Сюжет задачи выбран для ясности рассмотрения, а ничем не компенсированные задержки в других ситуациях могут быть критическими.

3.5. Сравнение по логичности и надёжности

В конкретной задаче общая сумма денег на счетах в реальности не константа. Ясно и привычно, что отправленные с одного счёта деньги поступают на другой не мгновенно. Стандартная реализация на основе транзакций прячет время транзакции в неповоротливость пользовательского интерфейса, искусственно создавая иллюзию мгновенности сделки.

Ретроспективное решение делает находящиеся в пути деньги доступными для точного последующего анализа.

История изменений счёта, полученная пользователем, полна до момента $\mu(t)$, а после $\mu(t)$ в ней могут временно отсутствовать деньги, «находящиеся в пути».

Запросы сводной информации на момент, предшествующий $\mu(t)$, не зависят от сервера. Для запроса сводных данных на последний момент у пользователя есть две возможности:

1) посмотреть сводную по всем доступным (неполным) данным с предупреждением о неполноте,

2) посмотреть *полную* сводную по всем данным на последний момент $\mu(t)$, для которого они доступны.

Если запросы следуют не чаще, чем может обработать традиционная система, то выдаётся в точности тот ответ, каким он был бы в традиционной системе на запрос, посланный из оптимально выбранного момента прошлого (неважно, был ли такой запрос в действительности).

Поломка связи между городами при стандартном подходе полностью блокирует операции в одном из них. При ретроспективном блокируются лишь операции, непосред-

ственно нуждающиеся в этой связи, остальное продолжает работать и сервис немедленно полностью восстанавливается при восстановлении связи.

Кроме упомянутых, ретроспективная организация имеет и другие полезные особенности:

Поломка или неожиданная потеря одного сервера базы данных (попадание метеорита) при стандартном подходе должно быть заранее предупреждено специальными мерами, например, чёткой организацией резервного копирования или репликации базы. Иначе потери могут оказаться фатальны.

При ретроспективном подходе без каких-либо дополнительных мер исчезновение сервера приводит к потере лишь части обновлений за последние доли секунды, что оставляет шансы на спасение.

3.6. Итоги сравнения

Теорема 1. Существует возможность реализации прикладной информационной системы с ретроспективным доступом, вполне адекватной функциональностью при разделении, быстрым автоматическим восстановлением после восстановления связи и с временем отклика в сто раз ниже, чем у любой реализации на традиционной основе.

Эта теорема разумеется противоречит не самой теореме CAP, а её популярным интерпретациям.

Рассмотренный пример вырос из неоднократно обсуждавшихся на конференции [9, 10, 11] ретроспективных систем. Он ставит ряд вопросов:

1) Ведёт ли он к общей технологии разработки ретроспективных информационных систем?

2) Не чрезмерны ли требования ретроспективной парадигмы к объёмам памяти и вычислительных ресурсов?

3) Сможет ли ретроспективная технология кардинально упростить исправление концептуальных ошибок и создание эволюционирующих систем?

4) Сможет ли ретроспективная технология резко уменьшить потери и неудобства пользователей, связанные с обновлением систем?

5) Сможет ли положительный эффект ретроспективного подхода проявиться при организации многопроцессорных распределённых систем и высокопроизводительных вычислений с большими гранулами параллелизма либо с задержками переконфигурирования FPGA [12]?

4. Ретроспективная парадигма программной инженерии

Проектирование системной архитектуры предполагает разделение системы на наиболее крупные составные части и принятие конструктивных решений, которые после их принятия с трудом поддаются изменению.

В рассматриваемой общей постановке неограниченно изменяются концептуальная модель, используемое оборудование и приложения (сервисы). Поэтому они не могут быть общей основой системной архитектуры для ретроспективной системы. Такой основой может быть только стабильная подсистема идентификации структур процессов и данных, обеспечивающая безусловную доступность неподдельной истории при произвольных изменениях.

Прежде, чем её описывать, следует разобраться с понятиями модульности и информационных объектов, которые в системе с историей и изменчивостью имеют свои особенности.

4.1. Сохранение полной истории

История изменений необходима

— для эффективного согласования распределённых данных,

— для быстрого восстановления при поломках,

— для ускорения отладки нового кода,

— для упрощения расследования вторжений и других происшествий,

— для повышения ответственности персонала.

Последние две цели требуют гарантий неподдельности истории. Эти гарантии нуждаются нуждаются в комплексной поддержки истории изменений *на более низком уровне*, чем механизмы разрешение конфликтов доступа и механизмы обеспечения прикладной функциональности.

Экономное сохранение истории достигается разделением данных на *небольшие независимо изменяемые части*. Например, для веб-сайта организации, управляемого CMS, должны сохраняться история изменений шаблонов фрагментов страниц, версии фрагментов заполнений, версии исполняемых процедур, обеспечивающих формирование страницы на этой основе (об истории активности пользователей разговор особый).

Пример: *За два года существования сайта каждый из этих элементов изменяется в среднем пять раз (редкие 100, многие 1), а средний размер элемента превышает длину штампа времени и текущий размер информации сайта 100 Mb.*

Общий объём всех версий исходных файлов, помеченных моментом сохранения, составляет

$$(100 + 100) \times 5 = 1000 \text{ Mb}.$$

Объём выданных пользователям веб-страничек, которые получаются разными сочетаниями исходных файлов и шаблонов, может на порядки превосходить эту цифру.

Для быстрого доступа к любой страничке из прошлого нужно лишь добыть актуальное на этот момент сочетание файлов и шаблонов, что несложно организовать с логарифмически растущими с объёмом издержками [10]. Много резервных копий системы не потребуется, поскольку вся история остаётся нетронутой и сможет пропасть только вместе с используемыми данными.

Пример показывает возможность хранения высоко доступной полной истории изменений при вполне умеренном росте ресурсозатрат при двух условиях

1) Выделение независимо изменяющихся «первичных» фрагментов информации в отдельные информационные объекты.

2) Несохранение объектов, просто однозначно реконструируемых из первичных.

Если ненужные версии данных занимают слишком много места, часть давно неактуальной информации может автоматически удаляться [13], создавая *белые пятна* истории на оси времени — промежутки, состоящие из моментов времени, информация на которые недоступна. Этот процесс мог бы проводиться автоматически субоптимально [14], оставляя относительный размер каждого белого пятна близким к минимально возможному при заданном ограничении общего объёма хранилища.

4.2. Модульность

Модульность освобождает разработчиков прикладных программ от сложностей организации исполнения. Она традиционно понимается как разделение на изолированные модули со своими кодом и данными, обменивающиеся сообщениями. Рассмотрим, как такое понимание может при структурных изменениях приводить к неконтролируемому росту сложности организации обмена сообщений.

Сложность эта порождается двумя принципиальными дефектами неуправляемого обмена сообщениями:

1) Новому исполнителю недоступна информация, разосланная участникам совместно выполняющейся работы.

2) Сообщения теряются по разным причинам.

3) Сообщение читается с неизвестным запозданием, за это время оно может потерять адекватность, а отменить непрочитанное сообщение автор не может.

Отсюда опасность использования изоляции модулей с организацией обмена сообщениями в коллаборативном приложении: после успешного тестирования малейшее изменение структуры взаимодействия или возникновение задержек может неожиданно выявить необходимость корректирующих сообщений. В итоге безупречная работа обеспечивается лишь до замены модулей, и откатов неправильных действий, а запуск новых исполнителей на замененном оборудовании и (или) с новыми алгоритмами исполнения требует серьёзного рефакторинга системы.

Альтернативой обмена сообщениями является публикация данных с подпиской на изменения, и, в частности, стандартное решение DDS (Data Distribution Service) [15].

Механизм обработки запросов на подписку и на публикацию авторы не регламентируют. Во избежание коллизий запросы вероятно должны обрабатываются строго, то есть проходить через общий вычислительный узел, что принципиально ограничивает масштабируемость. В любом случае это не снижает ценности DDS как перспективного новейшего технического решения для общей памяти распределённой системы, которое по-видимому может эффективно масштабироваться при устойчивых структурах данных с достаточно редко меняющимися настройками.

Подписка производится с настройками, устанавливающими количество хранимых версий, время жизни данных, количество реплицируемых копий, планово допустимые задержки, учитывающими период изменения данных и приоритетность передачи. DDS ничем не ограничивает свободы программиста по выбору этих настроек.

Получая полный доступ к гибкому распределению вычислительных ресурсов, прикладной программист к сожалению практически лишён разумных ориентиров. Это создаёт сложнейшую проблему избыточной гибкости настроек, которые для долгоживущей системы должны быть ещё и динамически управляемыми.

Решение этой проблемы требует системной организации для гибкого регулирования минимальным количеством прозрачных настроек.

Построив дерево всех объектов, настройки будем считать вектор-функцией от узла. Если предположить, что настройки чаще должны совпадать для смежных узлов, то задание настроек сводится к разрезанию дерева и заданию настроек для каждой связной части. Поскольку каждая связная часть имеет ближайший к корню элемент, то именно в нём можно задавать или менять настройки. Тогда настройки, сделанные в любом узле, наследуются во всей ветке, за исключением подветок со своими настройками.

Это означает, что проблема настроек разделяемой памяти сводится к разумному

представлению всех данных системы в виде дерева объектов.

Примерами таких веток в рассмотренном примере являются данные о изменениях в счёте отдельного пользователя и данные по городу. В общем случае такое разделение означает иерархию управления изменениями в системе.

4.3. Контекстная автономная архитектура

Архитектура контекстно-автономной системы [9] формируется на основе иерархии управления изменениями. Используется идея процессного подхода, рекомендованного стандартами ИСО серии 9000. Однако вместо термина *процесс* мы будем использовать более общий термин *активность*, не предполагающий обязательности входных данных и не ассоциирующийся с процессами операционной системы. Так, множество всех ИСО9000-процессов является подмножеством всех активностей системы.

Другие примеры активностей:

— нажатие пользователем кнопки, переход по ссылке или ввод текста;

— обнаружение события, нуждающегося в обработке.

— обработка данных, полученных от разных пользователей, при совместном редактировании.

Мониторинг сложных активностей осуществляют активности управления качеством, в частности корректирующие взаимных приоритетов дочерних активностей.

Системная активность управляет качеством системы в целом через такие дочерние активности, заведомо неполный список которых включает:

— мониторинг наличия и состояния физических устройств и каналов связи;

— мониторинг информационного обмена между активностями и между физическими узлами и перемещение активностей.

— мониторинг взаимных приоритетов активностей с общим родителем;

— мониторинг качества обновления информации;

— документирование кода и данных системы;

— уточнение приоритетов информационного обслуживания активностей;

Большинство задач системной активности в начальной фазе существования системы может осуществляться человеком, а затем переводиться на автоматическое управление. Автоматизация управления требует постановок задач поиска эффективных алгоритмов. Эти алгоритмы должны будут разумным образом перестраивать размещения активностей по физическим устройствам и оптимизировать коммуникации, используя в качестве входных данных изменения физической конфигурации системы, изменения приоритетов и статистику мониторинга активностей.

4.4. Логическая идентификация данных

Элемент данных (например, атрибут объекта) может сохраняться и реплицироваться в разных узлах системы. Изменение данного создаёт его новую версию. Прежде чем идентифицировать версии в различных местах системы, мы должны идентифицировать то, версии чего в ней сохраняются. Идентификатор самого элемента (например, поле конкретной формы, заполненное конкретным пользователем, содержащее код исполняемого в системе скрипта), не учитывающий версии и места хранения, будем называть *логическим идентификатором элемента данных*. Прозрачность идентификации данных крайне важна для доработок системы.

Для каждой активности A определим её *контекст* $\mathcal{D}(A) = [d_0(A), d_1(A)] \subset U$ как сегмент в линейно упорядоченном универсуме U всех возможных логических идентификаторов данных [13].

Универсум U будем считать *неисчерпаемым* в следующем смысле:

$$\forall x, y \in U \exists z \in U : x < z < y$$

Неисчерпаемость обеспечивает возможность размещения в любом контексте множества данных любой конечной мощности. Например, универсум строк символов неограниченной длины с отношением лексикографической упорядоченности неисчерпаем.

Для вложенных контекстов активность с более широким контекстом будем называть родительской, а с более узким — дочерней.

Принцип разделения ответственности и принцип единоначалия приводят к следующим требованиям:

1) *Контексты любых двух активностей либо вложены, либо не пересекаются.*

2) *Активность с более широким контекстом не модифицирует контексты дочерних активностей.*

3) *Перемещение активности в новую область резервирует старый контекст для возможного возврата и изоморфно переносит все дочерние активности.*

Добавляя совокупную активность, поддержанную системой, получаем в каждый момент времени дерево активностей. Оно изменяется — активности появляются, замирают, глубина вложенности активности может изменяться. Это обеспечивает произвольные структурные перестройки и гарантирует доступ к неискажённому прошлому.

4.5. Организация исполнения

Иерархия активностей упрощает проблему регулирования разделения ресурсов. Приоритеты допустимости задержек, уровня защищённости и ценности истории устанавливаются (и могут в любой момент быть пересмотрены) для общих активностей верхнего уровня.

1) Состояние изменяющегося объекта однозначно определено лишь в достаточно давнем прошлом.

2) Безупречный ответ может быть дан лишь на запросы об устоявшемся (линейно упорядоченном) прошлом, предшествующим моменту истины множества данных, нужных доля обработки запроса.

3) *Момент истины* (ранее которого прошлое устоялось) быстро определяется по времени и идентификатору.

4) Обновление информации в автономном контексте данных происходит с ограниченной частотой при появлении изменений, необработанных на момент, когда все входные данные устоялись.

5) Изменения помечаются комбинированным временем обработки, интерпретируемым и как частично упорядоченное логическое (линейно упорядоченное до момента истины), и как приближение к физическому времени.

Ограниченность частоты изменений обеспечивает локализацию последствий ошибок и открывает возможность оперативного автоматического выявления проявившихся в работающей системе противоречий. Локализация последствий ошибок вместе с автоматической диагностикой особенно важна для ретроспективной системы, поскольку перестройку «на ходу» невозможно осуществить без временных рассогласований и исправляемых ошибок.

Поясним сказанное.

Пример: один процесс обеспечивает $a = -b$, а другой — $b = a + 1$.

Подобное нарушение логики, проскользнувшее в обычную работающую систему, либо останется незамеченным, либо заставит систему бесконечно пересчитывать a и b.

Фиксация частоты изменений ограничивает любые негативные эффекты, в том числе и этот.

При появлении нескольких кандидатов для значения в близком времени возникает конфликт, требующий разрешения. Иногда для разрешения могут использоваться либо выбор значения, поступившего последним, либо выбор наибольшего (рекордного) из поступивших значений.

Остальные значения игнорируются или удаляются при записи. Если конфликты сложнее, то могут вводиться дополнительные активности для разных источников значений с целью разрешения конфликтов.

Если, например, от пользователя поступает более десяти форм в секунду, то это возможная попытка взлома. Классическое эффективное средство борьбы — это игнорирование промежуточных запросов от одного пользователя в малом промежутке времени. При этом запросы от всех разных пользователей должны быть сохранены и корректно обработаны.

Уменьшение частоты изменений контекстов ресурсозатратных активностей, таких как профиля пользователя или объёмная статистика, экономит ресурсы.

4.6. Ретроспективная СУБД как основа физической реализации

Доступ к физическим ресурсам (процессоры, память) обеспечивается ретроспективной СУБД [11]. На каждом вычислительном узле она хранит лишь часть общей истории изменений объектов, а именно:

1) данные активностей, осуществляющихся на узле,

2) наличие и давность необработанных изменений во входных данных,

3) данные активностей, используемых в качестве входных данных, в частности:

— исполняемый код осуществляющихся в узле активностей,

— относительные приоритеты исполнения, шаги дискретизации и моменты истины,

— списки активностей-подписчиков[3] с их шагами дискретизации,

— список узлов, на которых осуществляются активности-подписчики,

4) состояния процессов репликации данных на узлы активностей-подписчиков, давность непереданных изменений.

Ретроспективная СУБД реализует на узле следующую функциональность:

1) Быстрое сохранение изменённой версии объекта с штампом времени изменения.

2) Очень быстрое чтение версии объекта на заданный момент времени.

3) Передача изменений на узлы с активностями-подписчиками.

4) Сохранение изменений входных данных, полученных по подписке.

5) Поддержка ссылок на согласованные версии данных активности с изменениями в отдельных объектах.

Подробности конструкции ретроспективной СУБД описаны в [11] и [13].

[3] Активности-подписчики используют результаты осуществляющихся в узле активностей в качестве входных данных

5. Выводы:

Рассмотрен базирующийся на сохранении полной истории способ организации информационной системы.

Показано, как он обеспечивает резкое повышение доступности и защищённости системы.

В общих чертах описана архитектура подобной системы, основанная на общей ретроспективной памяти, поддерживаемой на узлах ретроспективными СУБД. Она позволяет гибко расширять и заменять оборудование, обновлять структуры, алгоритмы и пользовательские интерфейсы без приостановок обслуживания.

Описаны решения, позволяющие полностью контролировать накладные расходы на сохранение полной истории.

СПИСОК ЛИТЕРАТУРЫ

1. С. С. Андреев, А. А. Давыдов, С. А. Дбар, А. О. Лацис, Е. А. Плоткина. О моделях и технологиях программирования суперкомпьютеров с нетрадиционной архитектурой // Научный сервис в сети Интернет: суперкомпьютерные центры и задачи. М.: Изд-во МГУ, 2010. С. 186–187.

2. Ю. А. Климов, А. Ю. Орлов, А. Б. Шворин. Программный инструментарий для трафаретных вычислений на гибридных суперкомпьютерах // Прогр. Сист. Теор. Прил. 2012. Т. 3 №2(11). С. 23–49.

3. Ji Hong Yan, Chun Hua Feng. Sustainability-Oriented Product Modular Design Using Design Structure Matrix (DSM) // Method. Appl. Mechan. and Mater. 2011. P. 1468–1471.

4. Software Engineering for Self-Adaptive Systems: A Research Roadmap B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525. 2009. P. 1–26.

5. A. Metzger, K. Pohl, M. Papazoglou, E. Di Nitto, A. Marconi, D. Karastoyanova. Research challenges on adaptive software and services in the future internet: Towards an scube research roadmap // ICSE 2012.

6. Eric Brewer, Towards Robust Distributed Systems // Proc. XIX Ann. ACM Symposium on Principles of Distributed Computing. 2000. P. 7.

7. Кузнецов С.Д. Транзакционные параллельные СУБД: новая волна // http://citforum.ru/database/articles/kuz_oltp_2010/.

8. Seth Gilbert, Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services // ACM SIGACT News. **33** №(2). 2002. P. 51–59.

9. С.М. Абрамов, С.В. Знаменский, Н.С. Живчикова, А.В. Котомин, Е.В. Титова. Информационная система для разработки технологий организации сложной совместной деятельности // RCDL2009. С. 186–192.

10. С. В. Знаменский. Гибкая основа информационной системы для обучения // RCDL2010. С. 451–460.

11. С. В. Знаменский. Ретроспективная основа совместной реорганизации сложных информационных ресурсов // RCDL2011. С. 93–101

12. L. Bauer, C. Braun, M.E. Imhof, M.A. Kochte, H. Zhang, H.-J. Wunderlich, J. Henkel. OTERA: Online Test Strategies for Reliable Reconfigurable Architectures // AHS12. 2012. P. 38–45.

13. С. В. Знаменский. Глобальная идентификация данных в долговременной перспективе // Прогр. Сист. Теор. Прил. 2012. Т. 3 №2(11). С. 77–88.

14. С. В. Знаменский. Показатели эффективности расписания резервного копирования // Прогр. Сист. Теор. Прил. 2012. Т. 3 №2(11). С. 51–60.

15. Angelo Corsaro, Douglas C. Schmidt. The Data Distribution Service – The Communication Middleware Fabric for Scalable and Extensible / System of Systems, 2012.