АРХИТЕКТУРА РАСПРЕДЕЛЕННОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ E1

Леонид Рыжик, Антон Бурцев
{leonid, anton}@grafin.kiev.ua

7 июня, 2003

Современные распределенные операционные системы предоставляют пользователям прозрачный доступ ко всем ресурсам вычислительной сети посредством абстракции распределенного объекта. При этом, надежность и эффективность доступа определяется внутренней реализацией данной абстракции. Существующие архитектуры основываются на централизованном хранении состояния объекта в одном из узлов сети, либо на распространении состояния между несколькими узлами при помощи механизма распределенной разделяемой памяти. Указанные подходы не позволяют учесть семантику конкретного объекта. Как следствие, для большинства объектов эффективность доступа в распределенной среде значительно ниже, чем в локальном случае.

Авторы предлагают альтернативную архитектуру распределенной операционной системы, основанную на концепции репликации распределенных объектов. В каждом узле, где используется распределенный объект, размещается полная или частичная копия его состояния. Когерентность копий обеспечивается алгоритмами репликации. При этом, для каждого объекта может применяться наиболее эффективный алгоритм доступа, учитывающий его семантику. Все подсистемы ОС изначально спроектированы с учетом механизма репликации, что делает E1 удобной платформой для разработки надежных распределенных приложений.

Введение

В современных операционных системах поддержка распределенных вычислений, как правило, ограничена стеком сетевых протоколов. Однако, для построения распределенных приложений необходимы более развитые средства взаимодействия, такие как удаленные вызовы процедур, распределенные примитивы синхронизации и распределенная разделяемая память. Увеличивающаяся сложность программных систем формирует необходимость в новом уровне программного обеспечения, предоставляющем разработчикам эффективный, надежный и защищенный доступ к ресурсам сети.

На сегодняшний день, такой уровень наиболее часто реализуется системами промежуточного программного обеспечения. Промежуточное ПО определяется как программная прослойка между операционной системой и прикладными программами, предоставляющая единый набор абстракций в различных узлах распределенной системы. Например, в системах распределенной обработки данных широко используется компонентно-ориентированный промежуточный уровень, поддерживающий единую объектную модель в различных узлах вычислительного комплекса [35, 33, 53, 50].

Альтернативный подход заключается в том чтобы интегрировать поддержку распределенных вычислений в операционную систему. На сегодняшний день, развитые средства распределенного взаимодействия становятся необходимой базовой составляющей программного обеспечения, подобно файловой системе или механизму межпроцессного взаимодействия. Реализация на уровне ОС позволяет построить наиболее эффективную архитектуру, поддерживающую единый набор примитивов для доступа к локальным и удаленным ресурсам.

Распределенная ОС – это программная платформа, предоставляющая приложениям единую в рамках распределенной компьютерной системы среду выполнения, включающую средства доступа к аппаратным и программным ресурсам системы, а также механизмы взаимодействия приложений.

В данной работе представлен проект архитектуры распределенной операционной системы E1. Такая ОС, по мнению авторов, должна отвечать трем основным группам требований:

1. **Удобство интерфейса.** Природа распределенной среды такова, что для пользователей и разработчиков ПО работать в ней сложнее, чем в централизованной. Среди факторов сложности можно назвать: неоднородность доступа к локальным и удаленным ресурсам, высокую вероятность отказов, асинхронность доставки сообщений, отсутствие общей памяти. Для того чтобы в такой среде можно было выполнять вычисления, распределенная ОС должна поддерживать набор абстракций, изолирующих разработчика от перечисленных аспектов сложности и предоставляющих удобный интерфейс ко всем ресурсам вычислительного комплекса.
2. **Эффективность.** Эффективность ОС определяется главным образом временными характеристиками доступа к различным ресурсам. В распределенной среде низкая, по сравнению со скоростью доступа к оперативной памяти, скорость сетевых соединений становится узким местом производительности. Поэтому распределенная ОС должна максимально нейтрализовать влияние удаленного взаимодействия на эффективность работы программного обеспечения.
3. **Надежность.** В отсутствие механизмов отказоустойчивости сбой в работе одного из узлов или разрыв сетевого соединения может привести к потере работоспособности всей распределенной системы и уничтожению информации. Поэтому важным требованием к распределенной ОС является наличие механизмов надежности, обеспечивающих избыточное хранение данных и выполнение, а также восстановление после сбоев.

1. Концепции E1

В данном разделе представлен принятый в E1 подход к реализации каждого из указанных требований.

Удобство интерфейса.

Распределенная ОС должна предоставлять приложениям удобный интерфейс ко всем ресурсам компьютерной сети. С этой целью в E1 реализована абстракция образа единой системы. Это означает, что для прикладного ПО распределенная вычислительная система выглядит как централизованная. Данное свойство позволяет разработчику абстрагироваться от физического расположения ресурсов, с которыми взаимодействует приложение и сосредоточиться на функциональности, которую они предоставляют.

Реализация образа единой системы в E1 основана на абстракции **распределенного объекта**. Состояние и функциональность всех компонентов ОС инкапсулируется распределенными объектами. Каждый объект предоставляет набор четко определенных интерфейсов, которые могут вызываться другими объектами. Объекты глобально, единообразно доступны посредством своих интерфейсов из всех узлов вычислительного комплекса.

В E1 единая объектная модель используется как для подсистем ОС, так и для прикладного ПО. Т.е., приложения E1 также разрабатываются на основе распределенных объектов. С точки зрения прикладного программиста сеть ЭВМ представляет собой единый виртуальный компьютер, программное обеспечение которого структурировано в виде набора объектов. Доступ к аппаратным и программным ресурсам, а также взаимодействие между компонентами программной системы сводится к вызовам методов интерфейсов соответствующих объектов.

Эффективность.

Распределенные программные системы состоят из набора взаимодействующих компонентов, расположенных в различных узлах сети. Поскольку операции, выполняемые в каждом узле, часто зависят от команд и данных, получаемых от удаленных компонентов, задержки, возникающие при распределенном взаимодействии, в конечном итоге снижают эффективность функционирования всей системы. Для преодоления данного эффекта, в распределенных приложениях часто используются такие приемы как замена удаленного взаимодействия локальными операциями и вынесение удаленного взаимодействия за рамки критических путей выполнения программы. Замена удаленного взаимодействия локальным предполагает, что состояние объекта-сервера кэшируется в узлах, где находятся клиенты. В этом случае операции чтения могут выполняться локально над кэшированной копией объекта. Операции записи также иногда могут выполняться локально с последующей доставкой пакета изменений серверу. Метод вынесения удаленного взаимодействия за рамки критических путей направлен на то, чтобы сократить время, затрачиваемое основными вычислительными потоками на ожидание получения удаленного сообщения. Для этого используются вспомогательные потоки, спекулятивным образом получающие данные, требуемые для выполнения основных вычислений.

Обобщением указанных подходов является **репликация объектов**. В E1 состояние распределенного объекта может быть полностью или частично доступно локально в каждом узле, где данный объект используется. Состояние объекта синхронизируется (реплицируется) между узлами. Каждый вызов метода объекта вначале обрабатывается его репликой, находящейся в том узле, где производится данный вызов. Взаимодействие с удаленными репликами привлекается только в тех случаях, когда этого требует протокол репликации, например, когда необходимо получить недостающую часть состояния объекта.

Таким образом, распределенное взаимодействие в E1 перемещается вовнутрь распределенного объекта. Следовательно, эффективность доступа к объекту определяется эффективностью стратегии репликации. Очевидно, что не существует стратегии репликации, одинаково эффективной для всех типов объектов. Поэтому архитектура E1 не навязывает использование какой-либо конкретной стратегии или набора стратегий. Вместо этого, ОС предоставляет механизмы для построения реплицированных объектов. При этом, для каждого класса объектов может использоваться алгоритм репликации, обеспечивающий максимально эффективный доступ с учетом семантики объекта. Такой алгоритм может быть выбран из набора существующих стратегий репликации, либо разработан специально для данного класса объектов.

Надежность

E1 поддерживает два механизма надежности: репликация и персистентность. Репликация может выступать не только как средство обеспечения эффективного доступа к объекту, но и как механизм избыточности. Например, поддерживая когерентные копии объекта в *n* различных узлов, возможно достичь устойчивости объекта к выходу из строя любых *n-1* узлов [46]. Таким образом, репликация позволяет использовать аппаратную избыточность распределенной компьютерной системы для обеспечения устойчивого функционирования приложений. При этом, E1 предоставляет механизмы для построения реплицированных объектов, а собственно алгоритм избыточности реализуется стратегией репликации.

Механизм персистентности обеспечивает объектам неограниченное время существования, независимо от того, функционирует система непрерывно или нет. Для этого

копия объекта хранится в долговременной памяти и может синхронизироваться с активной копией. Сохраненное состояние объекта всегда корректно, даже при сбоях в работе узлов[1].

Еще один важный принцип, лежащий в основе архитектуры E1 – **поддержка компонентной модели**. В соответствии с данным принципом, модель реплицированных объектов расширяется до полноценной компонентной модели. Такая архитектура делает E1 удобной платформой для разработки распределенных приложений.

Прежде чем перейти к обсуждению использования компонентных моделей в распределенных ОС, скажем несколько слов о компонентной парадигме программирования в целом.

В основе компонентно-ориентированного подхода к разработке программного обеспечения лежит идея создания программных систем на основе повторного использования готовых оттестированных компонентов. Компонент должен представлять собой самостоятельный продукт, который может использоваться третьей стороной, не участвующей в проектировании и имплементации данного компонента.

Компонент определяется как единица композиции с определяемыми договоренностью интерфейсами и явными контекстными зависимостями [54]. Компоненты являются объектами, т.е. обладают свойствами инкапсуляции, полиморфизма и доступности посредством интерфейсов. Однако, компоненты обладают дополнительными свойствами, не присущими объектам языка программирования. В отличие от объекта, компонент является программным продуктом. Это, в частности, означает, что разработка компонента и его использование могут осуществляться независимо различными сторонами. Компонент – это исполняемый модуль, а не сущность языка программирования. Поэтому для компонентов не поддерживается наследование имплементации. Повторное использование компонентов достигается путем композиции и агрегации. Для того чтобы два компонента были интероперабельными необходимо и достаточно, чтобы они были разработаны в соответствии с требованиями компонентной модели. При этом не имеет значения, разрабатывались ли они при помощи одного или разных языков программирования. Компоненты обладают большей самостоятельностью, чем объекты, и, как следствие, им свойственна большая гранулярность. Как правило, компонент строится из нескольких взаимосвязанных объектов языка программирования.

Компонентная модель определяет среду функционирования компонентов включающую: механизм защищенного взаимодействия, средства именования компонентов, механизм динамической загрузки классов, механизм сборки мусора, средства разработки компонентов, а также набор дополнительных сервисов, таких как механизм персистентности, транзакции, средства репликации компонентов, объектный трейдинг и т.д. (см., например, [36]).

Единая компонентная модель может поддерживаться в нескольких узлах сети. Такая среда удобна для разработки распределенных приложений, поскольку, помимо других преимуществ компонентно-ориентированной архитектуры, обеспечивает прозрачность сети, т.е., компоненты, расположенные в различных узлах, могут вызывать методы друг друга так же как в локальном случае. Данный подход реализован в системах промежуточного ПО, таких как COM [33], Corba [35], EJB [53].

Любопытно, что многие современные распределенные ОС предоставляют набор абстракций и сервисов, близких по содержанию к распределенным компонентным моделям промежуточного ПО. Вероятно, это объясняется тем, что их разработчики преследовали

[1] На сегодняшний день архитектура механизма персистентности E1 недостаточно проработана. Поэтому в данной работе она не освещается.

сходные цели: построить среду, обеспечивающую удобную и эффективную поддержку распределенных приложений. Для унификации доступа к ресурсам распределенные ОС, как правило, предоставляют объектно-ориентированный интерфейс. В некоторых реализациях объект является одним из основных примитивов ОС ([55], [21], [11]), в то время как другие системы предоставляют более простые примитивы, такие как порты сообщений в Mach[4] и Chorus[43] или порталы в Opal[10], на основе которых абстракция объекта реализуется объектно-ориентированной средой выполнения приложений. Для управления распределенными объектами ОС предоставляет сервисы, аналогичные ключевым сервисам компонентных моделей. В первую очередь, это механизм защищенного взаимодействия, позволяющий единообразно вызывать методы любого объекта при наличии соответствующих прав. Кроме того, все распределенные ОС поддерживают механизм глобального именования объектов, позволяющий получить доступ к объекту посредством некоторого уникального идентификатора. Некоторые распределенные ОС поддерживают также персистентность объектов [11, 10, 12, 49, 15].

Несмотря на указанные сходства, на сегодняшний день нельзя говорить о полноценной поддержке распределенными ОС компонентных моделей. Абстракция объекта в них рассматривается скорее как удобное средство межпроцессного взаимодействия, а не способ структурирования приложений. Как сервисы ОС, так и приложения строятся в виде набора серверных процессов, предоставляющих точки входа для взаимодействия с другими программами. Посредством точки входа сервер предоставляет операции для доступа к некоторому ресурсу или группе ресурсов. Для вызова этих операций используется объектная семантика. Клиент указывает идентификатор точки входа, требуемую операцию и набор параметров. После выполнения вызова сервер возвращает одно или несколько значений. Таким образом объект служит в основном коммуникационной абстракцией.

В то же время, компонентная парадигма программирования рассматривает объекты как самостоятельные программные сущности со своим собственным состоянием, контекстными зависимостями и четко определенной функциональностью. Такое представление плохо укладывается в архитектуру современных распределенных ОС. Это не означает, что в них исключается использование компонентной модели, но для ее реализации требуется промежуточный уровень программного обеспечения, подобный используемому в традиционных (не распределенных) ОС.

Авторы убеждены, что потенциально реализация полноценной распределенной компонентной модели на уровне ОС обладает существенными преимуществами по сравнению с использованием промежуточного ПО. Разработчик компонентно-ориентированного промежуточного слоя неизбежно приходит к реализации некоторой абстрактной машины поверх примитивов, предоставляемых ОС, что, естественно, приводит к снижению эффективности. В то же время, если ОС проектируется с учетом распределенной компонентной модели на всех уровнях, то существует возможность построить максимально эффективную и надежную имплементацию. Такой подход принят в архитектуре E1, имплементирующей распределенную компонентную модель на основе реплицированных объектов. Все объекты, как прикладные, так и системные, существуют в рамках этой модели, т.е. являются компонентами.

Рассмотрим, чем компонентная модель E1 отличается от объектных моделей существующих распределенных ОС. Наиболее существенным отличием являются примитивы выполнения E1. В традиционных ОС основной абстракцией выполнения является процесс или задача, представляющая собой экземпляр программы, загруженной в память. Задача выполняется в изолированном адресном пространстве. С задачей может быть связано несколько потоков выполнения. Такая модель плохо адаптируется для поддержки взаимодействующих объектов средней гранулярности [18]. Поэтому мы отказываемся от нее в пользу новой модели выполнения, ориентированной на поддержку компонентных систем.

В E1 весь исполняемый код и данные принадлежат объектам. Все объекты расположены в едином 64-битном адресном пространстве. Для защиты объектов используется механизм страничной защиты памяти процессора. В E1 используется модель мигрирующих потоков[18], в которой поток, выполняющий вызов метода объекта, переносит выполнение в вызываемый объект. Такая модель позволяет отойти от клиент-серверного подхода к разработке объектов, когда каждый объект должен запускать один или несколько потоков для обработки вызовов его методов.

Второй особенностью компонентной модели E1 является то, что она основана на реплицированных объектах. Способность к репликации является неотъемлемым свойством каждого объекта. E1 содержит развитые средства поддержки репликации, включающие мощный и очень гибкий механизм удаленного взаимодействия между репликами распределенного объекта, инструменты для управлением локальным состоянием объекта в каждом узле, а также расширяемую библиотеку стратегий репликации.

Помимо механизма репликации объектов, компонентная модель E1 включает следующие элементы:

- Механизм защищенного взаимодействия, позволяющий выполнять вызовы методов любого объекта из любого узла системы. В E1 все вызовы выполняются над локальной репликой вызываемого объекта. Правомерность каждого вызова проверяется распределенным сервером контроля доступа (СКД). Архитектура СКД обеспечивает большую степень свободы при выборе конкретной модели защиты.
- Репозиторий классов и механизм динамической загрузки классов.
- Механизм глобального именования, обеспечивающий отображение уникального идентификатора объекта в список контактных точек данного объекта.
- Механизм сборки мусора, обнаруживающий и удаляющий неиспользуемые реплики объектов на основании анализа графа ссылок.
- Механизм персистентности, обеспечивающий контроль времени существования объектов путем надежного хранения целостного состояния объекта в долговременном хранилище.
- Средства поддержки разработки компонентов, включающие компилятор языка описания интерфейсов E1 и компилятор стратегий репликации.

В рамках единой компонентной модели E1 разрабатываются как объекты ОС, так и всё прикладное ПО. Поэтому важнейшими требованиями к реализации модели являются минимизация связанных с ней накладных расходов, высокая гибкость и эффективность. Эти требования легли в основу архитектуры компонентных сервисов E1, которая описана в следующих разделах данной работы.

2. Сравнение с существующими подходами

С точки зрения способа доступа к ресурсам распределенной системы современные распределенные ОС можно разбить на два класса: ОС архитектуры клиент-сервер и ОС, основанные на распределенной разделяемой памяти. В E1 принят отличный подход, основанный на реплицированных объектах. В этом разделе мы дадим краткую характеристику существующих архитектур и сравним их с архитектурой E1.

В ОС архитектуры клиент-сервер, как и в E1, все ресурсы вычислительного комплекса представлены объектами, единообразно доступными из всех узлов. Однако, объекты не являются физически распределенными. Каждый объект находится в одном из узлов системы под управлением серверного процесса. Глобальная доступность объектов обеспечивается механизмом удаленного вызова, скрывающим от клиента распределенный характер

взаимодействия. К системам типа клиент-сервер относятся, например, такие распределенные ОС как Amoeba[55], Mach[1], Chorus[43]. Преимуществом такой архитектуры является относительная простота реализации. Однако, она не обеспечивает локальность доступа к ресурсам и, следовательно, не исключает снижение производительности системы под влиянием сетевых задержек. Другой недостаток архитектуры клиент-сервер заключается в том, что она не обеспечивает надежного доступа к ресурсам. Выход из строя любого узла может вызвать волну программных сбоев по всей системе. Это явление получило название “эффекта домино”. Кроме того, архитектура клиент-сервер плохо масштабируется, так как не позволяет перераспределять нагрузку между узлами системы.

ОС на основе распределенной разделяемой памяти [10, 21, 13, 11] используют принципиально иной подход к реализации модели распределенных объектов. Основная идея такой архитектуры заключается в том, чтобы эмулировать общую память в распределенной вычислительной системе. Код и данные объекта представляют собой участки виртуальной памяти, глобально доступные из каждого узла по своему адресу. При первом обращении к объекту ОС создает локальные копии его страниц. Для синхронизации копий используются алгоритмы когерентности распределенной памяти [29]. Такие алгоритмы являются универсальными стратегиями репликации, применимыми для любых типов объектов. Но при этом они зачастую не обеспечивают эффективность доступа. Для реализации эффективного доступа к объекту алгоритм репликации должен выбираться с учетом семантики данного объекта. Естественно, репликация на уровне страниц памяти не может учитывать семантику объекта. Таким образом, мы наблюдаем закономерное противоречие между свойствами универсальности и эффективности стратегии репликации.

Для иллюстрации низкой эффективности распределенной разделяемой памяти рассмотрим гипотетический объект Очередь Сообщений. Объекты в различных узлах сети могут добавлять и читать сообщения из очереди. Каждый объект, имеющий доступ к очереди должен получить все сообщения, которые были в нее добавлены, т.е., в каждом узле в конечном счете должен быть получен одинаковый набор сообщений. Однако, порядок получения сообщений в различных узлах может отличаться – единственным требованием является каузальная согласованность доставки сообщений [27]. Предположим, что очередь реализована в виде простого односвязного списка, а размер сообщений фиксирован и равен нескольким байтам. Предположим, что для обеспечения распределенного доступа к Очереди Сообщений используется алгоритм когерентности распределенной памяти [29]. В этом случае каждая операция добавления сообщения требует монопольного блокирования минимум одной страницы объекта (содержащей указатель на последний элемент очереди). Для получения сообщения требуется произвести немонопольное блокирование одной страницы объекта. Каждая операция блокирования требует отправки сообщения всем узлам, содержащим копию объекта и получения подтверждения. Если очередь используется достаточно интенсивно, т.е. запросы на добавление и получение сообщений из очереди выполняются одновременно в различных узлах, то выполнение практически каждого запроса требует синхронизации локальной копии объекта с одной или несколькими удаленными копиями. Иными словами, каждое обращение к очереди требует обмена несколькими сообщениями с удаленными узлами.

В рамках подхода, основанного на распределенной разделяемой памяти, существенно улучшить эффективность доступа к Очереди Сообщений в принципе невозможно, поскольку алгоритмы когерентности памяти не учитывают семантику выполняемых над очередью операций, а также их гранулярность. В то же время, существуют алгоритмы репликации, позволяющие читать и записывать сообщения в очередь локально [26, 19]. Такие алгоритмы не гарантируют полного упорядочивания сообщений, но обеспечивают их доставку в каузально согласованном порядке. Чтение сообщения выполняется над локальной копией очереди. Операция добавления сообщения записывает новое сообщение в локальную

очередь, после чего управление возвращается вызывающему объекту. Доставка сообщения удаленым репликам производится асинхронно. Для обеспечения каузальной согласованности к каждому сообщению присоединяется метка логического времени [27]. Данный пример показывает, что стратегия репликации, учитывающая свойства конкретного объекта, может обеспечить значительно более эффективный доступ к нему, чем механизм разделяемой памяти.

Таким образом, как системы архитектуры клиент-сервер, так и системы на основе распределенной разделяемой памяти используют универсальные методы доступа к распределенным объектам (соответственно, удаленные вызовы методов и алгоритмы когерентности памяти). Эффективность этих методов ограничена, поскольку они не учитывают семантику каждого конкретного типа ресурсов.

Архитектура E1 позволяет использовать для каждого объекта стратегию репликации, обеспечивающую наиболее эффективный и надежный доступ к нему с учетом семантики данного объекта. Это, конечно, не означает, что для каждого класса объектов необходимо разрабатывать специальные стратегии репликации. В состав E1 входит обширная библитека готовых стратегий репликации, из которой можно выбрать эффективную стратегию практически для любого типа объектов. Так, например, описанный выше алгоритм репликации Очереди Сообщений – это один из вариантов активной репликации (раздел 6.1), которая может эффективно применяться для весьма широкого класса объектов.

Еще два примера стандартных стратегий репликации – стратегия клиент-сервер и стратегия репликации объектов памяти. Эти стратегии реализуют протоколы доступа к распределенным объектам, аналогичные соответственно ОС архитектуры клиент-сервер и ОС на основе распределенной разделяемой памяти. Таким образом, E1 можно рассматривать как обобщение этих архитектур.

3. Архитектура E1 в первом приближении

3.1. Архитектура распределенного объекта

Распределенный объект является основной абстракцией E1. Все сервисы, предоставляемые ОС, а также прикладное ПО строятся из распределенных объектов.

Все объекты в E1 находятся в едином виртуальном 64-битном адресном пространстве, архитектура которого обсуждается в разделе 4.1. Каждый объект предоставляет один или несколько интерфейсов, состоящих из набора методов. Каждый интерфейс распределенного объекта доступен по своему уникальному 64-битному адресу. Любой объект, владеющий этим адресом, может вызывать методы данного интерфейса из любого узла сети. Все интерфейсы в E1 содержат стандартный метод навигации, позволяющий получить указатель на любой другой интерфейс того же распределенного объекта.

Объекты в E1 могут быть физически распределенными, т.е. хранить частичную или полную копию состояния в нескольких узлах. Часть состояния объекта, хранящаяся в узле вычислительной системы, называется **репликой** распределенного объекта. Распространение состояния объекта между репликами и синхронизация реплик называется **репликацией** объекта.

Архитектура распределенного объекта E1 направлена на разделение семантики объекта и стратегии репликации. Разработчик объекта реализует только его семантику, т.е. функциональность в локальном, нереплицированном, случае, а алгоритм репликации разрабатывается отдельной стороной – поставщиком стратегии репликации. При этом стратегия репликации может быть универсальной, т.е. применяться для различных классов объектов. В то же время, для объектов одного класса могут применяться различные стратегии репликации.

Следуя данной цели, авторы предлагают архитектуру распределенного объекта, в которой семантика объекта и стратегия репликации реализуются различными структурными элементами. Распределенный объект представляет собой набор взаимодействующих **локальных объектов**. Локальный объект ограничен одним узлом распределенного вычислительного комплекса. Семантика распределенного объекта и его стратегия репликации реализуются различными локальными объектами, размещаемыми в узлах, в которых существуют реплики распределенного объекта. Отметим, что подобная архитектура распределенного объекта имплементирована в системе объектно-ориентированного промежуточного ПО Globe [50].

Структура локального объекта подобна структуре объекта языка C++ [52]. Он состоит из части фиксированной длины, содержащей поля данных и указатели на интерфейсы (таблицы методов), и структур данных, динамически выделяемых объектом в куче или стеке. В терминах C++, интерфейсы локального объекта представляют собой чисто виртуальные базовые классы, от которых пронаследован объект. Заметим, что аналогичным образом абстракция интерфейса реализована в компонентной модели COM[33].

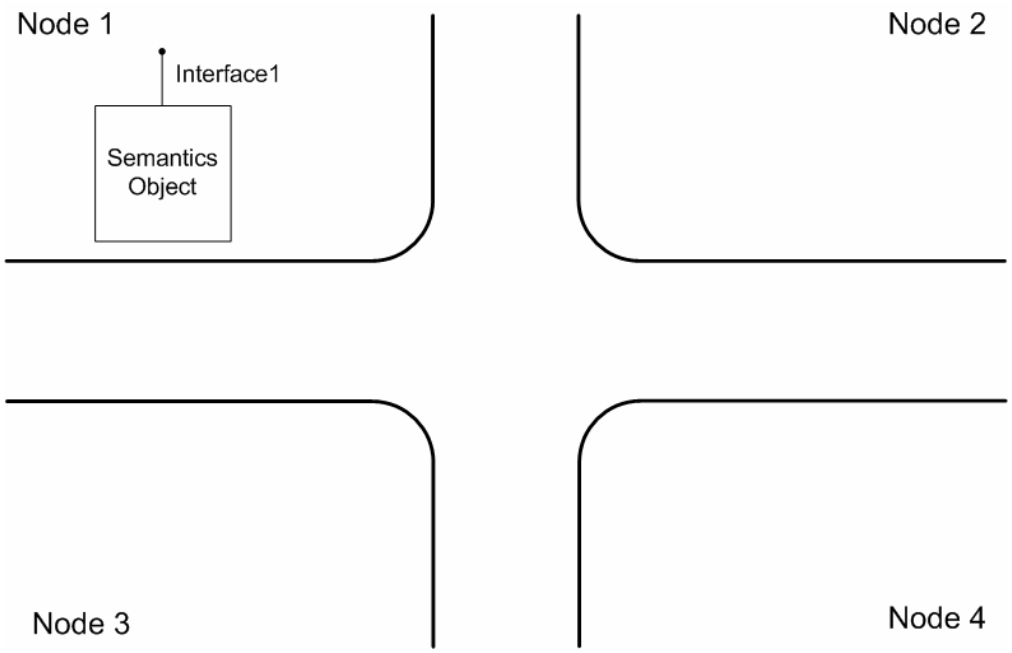
Архитектура распределенного объекта показана на рисунке 1.

В тривиальном случае, когда распределенный объект имеет только одну реплику (рис.1a), он отождествляется с единственным локальным объектом, **объектом семантики**. Объект семантики содержит состояние распределенного объекта, предоставляет интерфейсы распределенного объекта и реализует его функциональность.

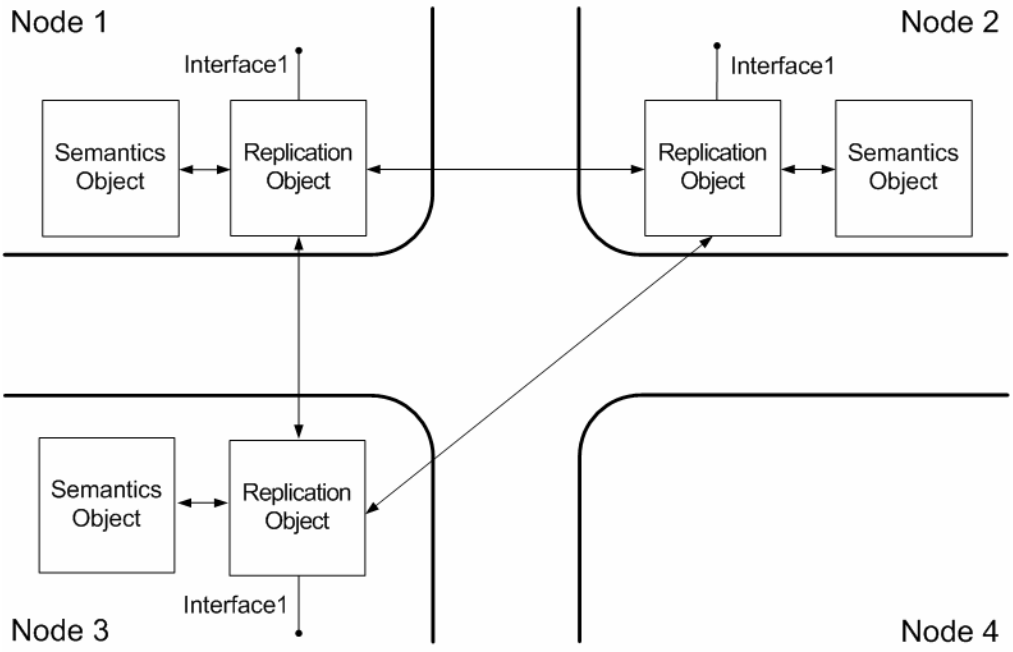
Во время создания ссылки на распределенный объект (см. раздел 5.4) из узла, в котором отсутствует реплика данного объекта, в этом узле создается новая реплика. Структура распределенного объекта с несколькими репликами показана на рисунке 1b. В каждый узел, в котором представлен распределенный объект, помещается экземпляр объекта семантики. Во всех узлах объекты семантики данного распределенного объекта располагаются по одному и тому же виртуальному адресу, что обеспечивает глобальную доступность интерфейсов распределенного объекта по их адресам в глобальном 64-битном адресном пространстве. Для поддержания целостности распределенного объекта, объекты семантики в каждом узле дополняются локальными **объектами репликации**, реализующими протокол репликации распределенного объекта.

Для этого объект репликации подменяет реализации интерфейсов объекта семантики собственными реализациями, что позволяет ему обрабатывать вызовы методов распределенного объекта[2]. При обработке вызова объект репликации может обращаться к объекту семантики для выполнения необходимых операций над локальным состоянием объекта, а также взаимодействовать с удаленными объектами репликации для синхронизации реплик и удаленного выполнения операций. Подмена интерфейсов прозрачна для других объектов и может мыслиться как агрегирование объектом репликации объекта семантики. Такая архитектура позволяет избежать накладных расходов, связанных с поддержкой объекта репликации, для объектов, которые представлены только в одном узле, т.е. фактически не являются распределенными. Если впоследствии у распределенного объекта снова остается одна реплика, объект репликации может быть уничтожен.

[2] Техника подмены интерфейсов не обсуждается подробно в данной работе



(a)



(b)

Рисунок 1. Архитектура распределенного объекта. a. Распределенный объект с единственной репликой; b. Распределенный объект с несколькими репликами

Описанная архитектура распределенного объекта обладает двумя важными свойствами. Во-первых, она эффективно разделяет семантику объекта и стратегию репликации. Во-вторых, она не накладывет каких-либо существенных ограничений на используемые алгоритмы репликации. Для каждого объекта может применяться стратегия репликации, обеспечивающая при заданных характеристиках надежности максимально эффективный доступ к объекту с учетом его семантики.

3.2. Объекты классов

Классы локальных объектов в E1 описываются объектами специального типа – **объектами классов**. Инкапсуляция свойств класса в объекте позволяет реализовать динамическую загрузку классов. Для этой же цели служат фабрики классов в COM [33] и Corba [37].

Объект класса хранит имплементации интерфейсов и предоставляет методы для создания и уничтожения объектов-экземпляров данного класса.

Каждый класс описывается уникальным идентификатором. Классы хранятся в специальном хранилище - **репозитории классов**. Для создания экземпляра класса в памяти необходимо предварительно загрузить объект класса с соответствующим идентификатором из репозитория. Как и другие системные сервисы E1, репозиторий классов является распределенным объектом. Общий репозиторий используется всеми узлами системы.

Каждый распределенный объект состоит из локальных объектов двух типов – объектов семантики и объектов репликации. В E1 нет понятия класса для распределенных объектов. При создании нового распределенного объекта указывается класс соответствующего объекта семантики, который может отождествляться с классом всего распределенного объекта, поскольку именно объект семантики инкапсулирует полезную функциональность распределенного объекта. При этом для распределенных объектов одного класса могут использоваться различные классы объектов репликации (реализующие, соответственно, различные стратегии репликации).

3.3. Обзор архитектуры E1

На рисунке 2 показана обобщенная архитектура E1. ОС E1 состоит из микроядра и множества распределенных объектов, функционирующих на прикладном уровне. Микроядро поддерживает минимальный набор примитивов, необходимых для построения операционной системы: адресные пространства, потоки выполнения, средства межпроцессного взаимодействия, диспетчеризация прерываний. Функции ОС и прикладное программное обеспечение реализуются объектами.

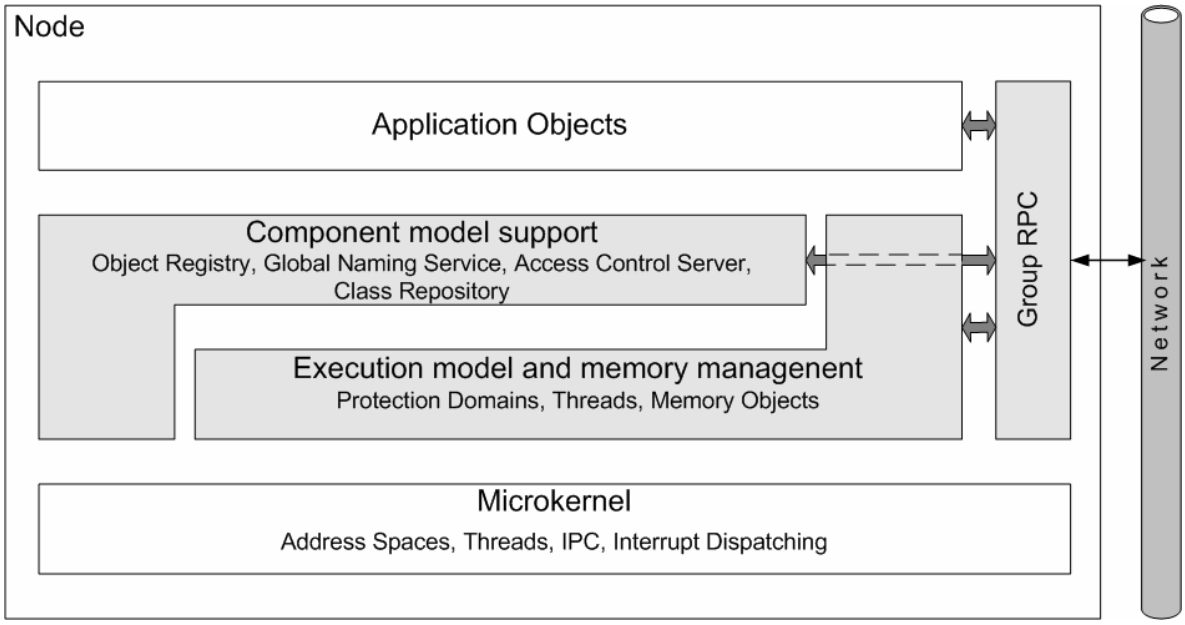


Рисунок 2. Обобщенная архитектура E1

Архитектура на основе микроядра обладает рядом преимуществ. Во-первых, она потенциально более надежна, чем традиционная монолитная архитектура, поскольку позволяет перенести большую часть кода ОС из привилегированного ядра на прикладной уровень. При этом, система становится более простой, более модульной и, как следствие, уменьшается количество ошибок и повышается устойчивость системы к ошибкам в отдельных подсистемах. Во-вторых, разработка ОС на основе готового микроядра значительно проще, чем разработка “с нуля”, поскольку микроядро предоставляет удобный набор примитивов, которые обеспечивают высокий уровень аппаратной абстракции, не накладывая при этом значительных ограничений на архитектуру ОС. Кроме того, поскольку сервисы ОС выполняются на прикладном уровне, а не в привилегированном ядре, существует возможность заменять, модифицировать, отлаживать отдельные сервисы непосредственно во время работы системы или даже запускать одновременно несколько версий некоторых сервисов. Наконец, в-третьих, как показывают исследования в данной области [30], архитектура на основе микроядра может быть более эффективной, чем монолитная архитектура. На сегодняшний день существуют микроядра, характеризующиеся очень низкими задержками при передаче управления между адресными пространствами [32]. Этот показатель является ключевым для систем, ориентированных на интенсивное взаимодействие компонентов, каковой является E1. К таким микроядрам относятся микроядра семейства L4 [31, 14, 39, 25]. Поэтому L4 было выбрано в качестве микроядра ОС E1.

Собственно компоненты E1 можно условно разбить на три группы (выделенные на рисунке 2 светло-серым фоном):

1. Примитивы выполнения, защиты и управления памятью.
2. Сервисы, реализующие поддержку компонентной модели.
3. Средства поддержки репликации (Group RPC на рисунке).

4. Взаимодействие и защита объектов

В E1 функции ОС и прикладного программного обеспечения инкапсулируются распределенными объектами. Для выполнения полезных действий объекты взаимодействуют посредством вызовова методов. Безопасное выполнение приложений обеспечивается механизмом защиты, гарантирующим, что взаимодействие объектов ограничивается четко определенной политикой контроля доступа.

Модель защиты объектов E1 основана на трех предположениях:

1) Методы объекта не имеют непосредственного доступа к состоянию других объектов (требование изоляции объектов).
2) Объекты могут взаимодействовать только посредством вызова методов.
3) Вызовы методов контролируются ОС, которая может проверить соответствие каждого вызова действующей политике контроля доступа.

Эти предположения обеспечиваются, соответственно, тремя механизмами: **домены защиты**, **междоменные вызовы** и **механизм контроля доступа**. В данном разделе рассматриваются домены и междоменные вызовы, а также связанная с ними модель потоков. Механизм контроля доступа описан в разделе 5, посвященном компонентной модели E1.

4.1. Домены защиты

Модель защиты ОС основывается на механизмах, предоставляемых аппаратной платформой. В современных микропроцессорах это, в первую очередь, механизм

виртуальной памяти. Поэтому защита объектов тесно связана с организацией виртуальной памяти.

В E1 все объекты расположены в едином виртуальном адресном пространстве. Интерфейсы распределенного объекта могут вызываться непосредственно по их адресам в памяти, подобно тому как в языке C++ методы объекта вызываются по указателю на объект. Главным преимуществом такой структуры виртуальной памяти является удобная модель программирования, максимально упрощающая организацию взаимодействия между объектами. Подчеркнем, что единое адресное пространство поддерживается в рамках всего распределенного вычислительного комплекса. Таким образом, распределенный объект может обращаться к любым данным или объектам в системе по их уникальным виртуальным адресам из любого узла сети.

Очевидно, что 4-гигабайтное адресное пространство 32-битных процессоров не позволяет вместить весь код и данные распределенной вычислительной системы. Для реализации ОС на основе единого адресного пространства требуется аппаратная платформа с широким пространством виртуальных адресов. Поэтому E1 ориентирована на использование микропроцессоров с 64-битной адресацией памяти.

Заметим, что в принципе модель распределенных объектов E1 может быть успешно реализована и в рамках модели изолированных адресных пространств. В этом случае для организации взаимодействия объектов могут использоваться косвенные ссылки [47], позволяющие осуществлять передачу управления между адресными пространствами. Тем не менее, единое адресное пространство позволяет построить более простую и удобную реализацию. Анализ преимуществ ОС на основе единого адресного пространства можно найти в работах [10, 22].

Вернемся к задаче защиты распределенных объектов. Напомним, что в основе модели защиты E1 лежит требование изоляции объектов, предполагающее, что состояние объекта недоступно для методов других объектов. Для достижения такой изоляции внутри единого адресного пространства, необходимо связать с каждым объектом отдельный контекст защиты, которому принадлежит код и состояние объекта и в рамках которого выполняются методы данного объекта. Виртуальная память вне контекста защиты недоступна для методов объекта. Такая схема обеспечивает корректную изоляцию объектов, однако приводит к значительным накладным расходам. Во-первых, модули управления памятью современных процессоров обеспечивают страничную гранулярность защиты. Это означает, что контекст защиты должен состоять из целого числа страниц. Так, например, для объекта размером несколько сот байт потребуется целая страница физической памяти. Наличие в системе большого количества маленьких объектов приводит к крайне неэффективному использованию памяти. Кроме того, при использовании указанного способа изоляции объектов, каждый вызов метода сопровождается сменой контекста защиты. Эта операция требует большого количества циклов процессора. При достаточно интенсивном взаимодействии объектов на смену контекста расходуется значительная часть процессорного времени.

Для того чтобы все же построить эффективный механизм изоляции объектов, в E1 вводится абстракция **домена защиты**. Домен представляет собой часть единого виртуального адресного пространства, внутри которой находится один или несколько распределенных объектов. Каждый объект в E1 принадлежит некоторому домену. С каждым доменом связан отдельный контекст защиты изолирующий, внутренние объекты домена от других объектов в системе. Однако, объекты внутри домена не защищены друг от друга. Вызовы методов внутри домена не требуют смены контекста защиты. Таким образом, домены позволяют достичь компромисса между эффективностью взаимодействия и степенью защищенности объектов.

При объединении объектов в домены следует учитывать следующие факторы:

- размещение объектов в различных доменах позволяет защитить их от случайных или преднамеренных попыток несанкционированного доступа;
- вызов метода объекта внутри домена эффективнее, чем междоменный вызов;
- объекты, находящиеся в общем домене, более экономно используют физическую память;

В соответствии с этими факторами, в один домен следует помещать интенсивно взаимодействующие объекты, реализующие общую функциональность.

Домены обеспечивают глобальную изоляцию объектов в рамках всей распределенной системы. Если объект имеет несколько реплик, то в каждом узле его реплики находятся в одном и том же домене, по одному и тому же виртуальному адресу. Следовательно, если два объекта изолированы друг от друга, т.е. находятся в различных доменах, то их реплики будут находиться в различных доменах во всех узлах. Как и другие примитивы E1, домен является распределенным объектом. Реплика домена помещается в каждый узел, в котором существует реплика по крайней мере одного объекта, принадлежащего данному домену.

4.2. Междоменные вызовы

Объекты в E1 взаимодействуют посредством вызова методов. Данный вид взаимодействия является синхронным. Каждый вызов сопровождается передачей набора аргументов и возвратом нуля или более значений. Передаваемые аргументы и возвращаемые значения описываются разработчиком объекта средствами Языка Описания Интерфейсов (IDL).

В E1 все вызовы методов выполняются локально. Т.е., при вызове распределенного объекта происходит обращение к его реплике в локальном узле. Для того чтобы во время работы с объектом такая реплика существовала и не была уничтожена системой сборки мусора, необходимо предварительно создать сильную ссылку на данный объект (см. раздел 5.4).

Для вызова методов объекта используется указатель на один из его интерфейсов. Поскольку все объекты в E1 находятся в едином виртуальном адресном пространстве, этот указатель является действительным в любом узле сети и в любом домене. Объект, владеющий указателем на интерфейс другого объекта, может вызывать его методы одинаковым образом, независимо от того, находится ли вызываемый объект в том же или в другом домене. Все интерфейсы в E1 содержат стандартный метод навигации, позволяющий получить указатель на любой другой интерфейс того же распределенного объекта.

Рассмотрим, каким образом выполняются вызовы методов в E1. В пределах одного домена взаимодействие объектов происходит так же как в языке C++. Вызов метода представляет собой простую передачу управления по адресу, записанному в таблице методов вызываемого объекта. При этом аргументы вызова помещаются в стек и в регистры процессора.

С точки зрения взаимодействующих объектов, междоменные вызовы ничем не отличаются от локальных. Однако, их реализация на уровне ОС несколько сложнее. При обращении к объекту, расположенному вне локального домена, возникает исключительная ситуация. Обработка данного исключения осуществляется специальным объектом – *Адаптером Междоменных Вызовов(AMB)*, находящимся в том же домене, что и объект, вызвавший исключение. Задача AMB – подготовить стек, который будет отображен в домен вызываемого объекта и на котором будет выполнен вызов метода. В этот стек необходимо скопировать аргументы вызова. В E1 поддерживаются аргументы двух типов: передаваемые по значению и по ссылке. Аргументы, передаваемые по значению, копируются в стек непосредственно. Данные, передаваемые по ссылке, также необходимо скопировать в стек и соответствующим образом настроить указатель на них. Механизм междоменных вызовов не

позволяет передавать большие массивы данных без копирования. Для этого следует использовать аппарат разделяемой памяти E1.

В каждом узле находится только один AMB, который отображается во все домены. Таким образом, AMB выполняет роль универсального прокси, обрабатывающего все междоменные вызовы в системе. Подготовка стека для междоменного вызова должна производиться на основе IDL-описания вызываемого метода. Поэтому для копирования параметров в стек вызова AMB обращается к объекту класса вызываемого объекта, который предоставляет для этого специальные методы.

Для того чтобы не создавать отдельный сегмент стека для выполнения каждого междоменного вызова, AMB использует стек, на котором поток, производящий данный вызов, выпонялся в исходном домене. Вершина стека выравнивается по границе страницы и полученный адрес интерпретируется как дно нового стека (рис.3). Таким образом, после перехода в целевой домен поток продолжает выполняться на том же стеке, что и в исходном домене, но при этом содержимое исходного стека изолировано от вызываемого объекта.

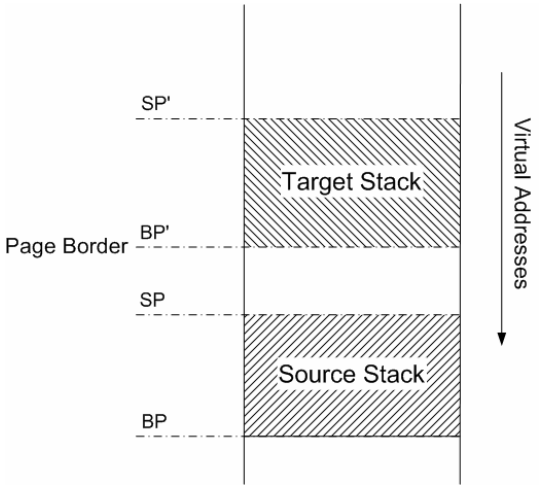


Рисунок 3. Управление стеком при междоменном вызове. BP и SP – указатели дна и вершины стека в исходном домене. BP′ и SP′ – указатели дна и вершины стека в целевом домене.

После подготовки стека AMB передает управление микроядру для завершения междоменного вызова. Ядро обращается к Реестру Объектов (раздел 5.1) для проверки права вызывающего объекта на выполнение данного вызова, после чего отображает сформированный стек в целевой домен и передает управление вызываемому объекту. Возврат из междоменного вызова происходит аналогичным образом, через AMB целевого домена.

4.3. Потоки

Модель выполнения E1 основана на концепции мигрирующих потоков [18]. В каждый момент времени поток выполняется в контексте некоторого объекта. При вызове метода выполнение потока переносится в вызываемый объект. Таким образом, поток не связан перманентно с каким-либо объектом или доменом. В работе [18] показано, что мигрирующие потоки лучше, чем традиционные статические потоки адаптированы для работы в объектной среде. Поскольку каждый метод выполняется тем же потоком, который его вызвал, нет необходимости запускать отдельный поток для обработки каждого вызова или буферизовать вызовы для их последовательной обработки. Как следствие, модель мигрирующих потоков

обеспечивает более эффективную передачу управления между объектами, а также упрощает разработку объектов и делает их более легковесными.

При создании нового потока указывается объект и метод, в контексте которого данный поток начнет свое выполнение. При выполнении этого метода поток может производить вызовы других объектов. Степень вложенности вызовов не ограничена. Поток может быть уничтожен после того как он завершит выполнение метода, в контексте которого был создан, либо при возврате из вложенного вызова, если объект, который производил данный вызов, был удален.

Если поток выполняет недопустимую операцию, то реплика объекта, в контексте которой данный поток выполнялся, уничтожается, а сам поток возвращается на один шаг назад в стеке вложенных вызовов, так же как если бы он завершил выполнение метода с кодом ошибки.

Поскольку в E1 вызов метода распределенного объекта – это вызов его локальной реплики, то он никогда не приводит к переносу выполнения потока в другой узел. Существует, однако, ситуация, когда поток может переместить выполнение в удаленный узел. Такая необходимость возникает тогда, когда стратегия репликации объекта предусматривает миграцию одной из его реплик между узлами системы (раздел 6.1). Тогда состояние объекта переносится вместе с выполняющимися в нем потоками. После завершения выполнения метода поток возвращается на исходный узел.

С каждым потоком связана специальная конструкция – *стек активаций*. Стек активаций описывает последовательность вложенных вызовов, производимых потоком. В него записываются как междоменные вызовы, так и вызовы объектов внутри одного домена. Элемент стека активаций содержит адрес объекта, который произвел вызов. Для междоменных вызовов стек активаций хранит также значения регистров, которые восстанавливаются после возврата из вызова.

Стек активаций необходим для того, чтобы поток мог корректно вернуться из междоменного вызова и возобновить выполнение в контексте вызывающего объекта. Кроме того, с его помощью ОС может управлять поведением потока при возврате из метода: заблокировать его, перенести на удаленный узел, либо уничтожить. Для этого в элементы стека активаций могут записываться специальные команды, которые обрабатываются в момент их извлечения из стека. Таким образом, выполнение операции над потоком откладывается до тех пор, когда он возвращается в исходный объект, закончив возможные модификации состояния вызываемого объекта.

5. Сервисы компонентной модели

В данном разделе рассматриваются системные сервисы E1, расширяющие модель распределенных объектов до полноценной компонентной модели. К ним относятся Реестр Объектов, Сервер Контроля Доступа, Сервер Глобального Именования, а также система сборки мусора[3].

5.1. Реестр Объектов

Центральным сервисом компонентной модели E1 является **Реестр Объектов**. Реестр хранит информацию о всех репликах распределенных объектов, существующих в локальном узле, в том числе, адреса объектов и их интерфейсов, размещение объектов в доменах, информацию о ссылках между объектами. Реестр играет координирующую роль при

[3] За рамками данной работы остается подробное описание механизмов динамической загрузки классов и персистентности

выполнении таких операций как создание и удаление распределенных объектов и их реплик, междоменные вызовы, сборка мусора.

Рассмотрим каждую из указанных функций Реестра Объектов подробнее

Создание и удаление объектов

Распределенные объекты в E1 создаются при помощи метода CreateObject Реестра. Аргументами метода являются идентификатор класса и домен, в котором должен быть создан объект. Кроме того, может указываться стратегия репликации, которая будет для него применяться. Метод CreateObject выполняет следующую последовательность действий:

1. Если в целевом домене отсутствует требуемый объект класса, Реестр обращается к Репозиторию Классов, который загружает данный объект класса в домен.
2. Реестр вызывает объект класса для создания объекта семантики. Поскольку на данный момент новый распределенный объект представлен только в одном узле, то объект репликации для него не нужен.
3. Информация об объекте сохраняется в структурах данных Реестра. В частности, объект, вызвавший метод CreateObject, регистрируется как владелец единственной сильной ссылки на новый объект.
4. Объект регистрируется в сервисе глобального именования.
5. Метод CreateObject возвращает указатель на один из интерфейсов нового распределенного объекта.

Как правило, распределенный объект удаляется, когда все его реплики обращаются в мусор (см. раздел 5.4). Кроме того, Реестр предоставляет метод DeleteObject для принудительного удаления объектов.

Создание реплики существующего распределенного объекта

Вначале каждый распределенный объект состоит из единственной реплики. Впоследствии реплики могут создаваться и уничтожаться. Как будет описано в разделе 5.4, для создания новой реплики существующего распределенного объекта в некотором узле, необходимо создать в данном узле сильную ссылку на этот объект. Когда реестр обнаруживает, что объект, на который создается ссылка, не представлен в локальном узле, он выполняет следующий алгоритм создания реплики объекта:

1. Обращается к сервису глобального именования за информацией об указанном объекте: класс, контактные точки, используемая стратегия репликации.
2. При необходимости, обращается к Репозиторию Классов для загрузки классов объекта семантики и объекта репликации в целевой домен.
3. Вызывает объекты классов для создания объекта семантики и связанного с ним объекта репликации.
4. Инициализирует объект репликации списком контактных точек, на основании которого будет запущен распределенный алгоритм вхождения новой реплики в состав распределенного объекта. Этот алгоритм является частью стратегии репликации.

Контроль междоменных вызовов

В системе междоменного взаимодействия Реестр Объектов отвечает за проверку корректности и правомерности вызовов. При выполнении междоменного вызова микроядро обращается к Реестру для того чтобы убедиться, что в локальном узле существует реплика вызываемого объекта и вызов является правомерным, т.е. вызывающий объект имеет право обращаться к указанному интерфейсу данного объекта. Кроме того, Реестр сообщает ядру, в каком домене находится вызываемый объект. Для взаимодействия с микроядром Реестр Объектов предоставляет интерфейс *IAccessValidator* (рис. 4).

Реестр не реализует самостоятельно политику контроля доступа. Вместо этого, для прорверки правомерности вызова Реестр обращается к другому системному объекту – **Серверу Контроля Доступа**, который обсуждается в следующем разделе.

Для повышения эффективности механизма междоменного взаимодействия информация об объектах может кэшироваться в микроядре, что позволяет избежать обращения к Реестру при каждом вызове. На рисунке красным пунктиром показан оптимизированый путь выполнения междоменного вызова. Кэш ядра должен постоянно поддерживаться в корректном состоянии, отражающем изменения, происходящие на верхнем уровне. Например, если удаляется локальная реплика или изменяются права на доступ к некоторому объекту, это должно приводить к удалению соответствующих записей из кэша. Однако, само микроядро не располагает информацией, позволяющей корректно управлять кэшем объектов. Поэтому микроядро предоставляет интерфейс *IAccessCache*, посредством которого Реестр Объектов и Сервер Контроля Доступа могут управлять содержимым кэша.

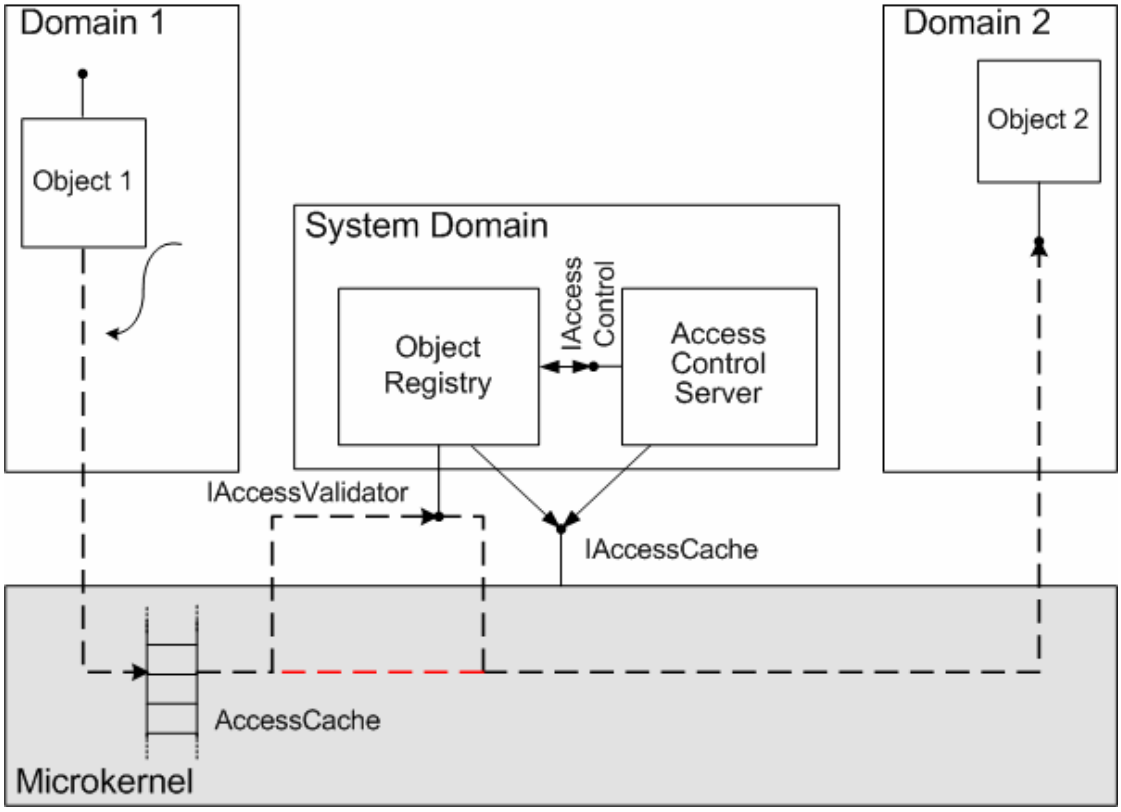


Рисунок 4. Взаимодействие микроядра и Реестра Объектов при выполнении междоменного вызова. Пунктиром показан нормальный и оптимизированный (красным) путь выполнения междоменного вызова.

Еще одна важная функция Реестра Объектов – управление ссылками и сборка мусора –будет рассмотрена в разделе 5.4.

5.2. Сервер Контроля Доступа

Сервер Контроля Доступа (СКД) – это распределенный сервис, реализующий единую политику ограничения доступа в рамках всего вычислительного комплекса. Для каждого вызова метода в системе СКД должен ответить на вопрос о правомерности данного вызова.

Выбор конкретной модели контроля доступа на этапе проектирования ОС – сложная задача. Ни одна из существующих на сегодняшний день моделей защиты не является универсальной и не может в общем случае считаться оптимальной. Поэтому архитектура E1 не определяет конкретную политику контроля доступа, которую должен имплементировать СКД. Никакие ограничения не накладываются и на стратегию репликации СКД, а также на структуры данных, используемые для описания прав. Вместо этого, СКД должен реализовывать интерфейс *IAccessControl*, посредством которого Реестр Объектов проверяет правомерность междоменных вызовов. Главный метод *ValidateAccess* данного интерфейса разрешает или запрещает выполнение вызова на основании идентификатора потока, производящего вызов, объекта, в контексте которого инициируется вызов, а также, вызываемого объекта и метода.

СКД может также предоставлять дополнительные интерфейсы, которые определяются конкретной моделью контроля доступа и реализуют специфичный для нее протокол распространения прав. Так, например, для реализации модели полномочий Take/Grant [7, 8], СКД может предоставлять интерфейс, содержащий методы *Take*, *Grant* и *Revoke*, а для эмуляции модели защиты UNIX необходимы методы *Chmod* и *Chown*.

Глобальное выполнение правил ограничения доступа в рамках распределенной системы обеспечивается репликацией Сервера Контроля Доступа. Так, например, при отзыве некоторого полномочия, стратегия репликации СКД доставляет соответствующую нотификикацию всем узлам, в которых хранится устаревшая информация. Накладные расходы, связанные с репликацией СКД, – один из важных факторов, которые необходимо принимать во внимание при выборе модели ограничения доступа.

Описанная архитектура СКД позволяет имплементировать разнообразные модели защиты, в том числе, различные модели полномочий (capabilities) [56, 20, 24, 7] и списков контроля доступа (ACL) [42]. При этом, как объект, так и субъект защиты могут выбираться различным образом. Например, в роли объекта может выступать распределенный объект, интерфейс или метод, а в роли субъекта – распределенный объект, домен или пользователь. Обратим внимание на последнюю возможность. До сих пор абстракция пользователя в E1 не вводилась. Тем не менее, модели, в которых права принадлежат пользователям или ролям имеют широкое применение [44, 42]. В такой модели каждый поток выполняется от имени и с полномочиями некоторого пользователя. Следовательно, хотя понятие пользователя не является основным для E1, существует возможность реализовать данную абстракцию на уровне Сервера Контроля Доступа, отождествляя пользователей с группами потоков.

5.3. Сервер Глобального Именования

Сервер Глобального Именования (СГИ) E1 реализует протокол локализации объектов, позволяющий по виртуальному 64-битному адресу объекта получить список его контактных точек, т.е. узлов сети, в которых находятся реплики данного объекта, а также класс объекта и используемую стратегию репликации. Реестр Объектов обращается к СГИ для того чтобы инициировать создание новой реплики распределенного объекта.

Конкретный протокол локализации объектов должен выбираться, исходя из масштаба распределенной системы и частоты удаления и добавления узлов. Для небольших систем может использоваться централизованный протокол с одним или несколькими серверами имен. Для крупномасштабных систем с постоянной структурой, как правило, используется иерархическая система доменных серверов [34], а для динамически изменяющихся систем наиболее эффективны децентрализованные протоколы именования[51].

5.4. Сборка мусора

Назначение механизма сборки мусора E1 – обнаружение и удаление неиспользуемых реплик распределенных объектов.

В большинстве ОС нет необходимости выделять сборку мусора в отдельную подсистему. Вместо этого, каждый компонент операционной системы использует собственный механизм учета ресурсов, основанный, как правило, на простом подсчете ссылок. Такой механизм легко реализуется и не влечет значительных накладных расходов. Однако, в распределенной ОС ненадежность узлов и сетевых соединений, а также асинхронный характер взаимодействия существенно усложняют задачу управления ссылками [38]. Кроме того, в E1 каждый объект может иметь несколько реплик в различных узлах, что также затрудняет определение момента, когда объект превращается в мусор. Для организации сборки мусора в такой системе необходимы достаточно сложные распределенные структуры данных и алгоритмы. Разрабатывать их отдельно для каждой подсистемы ОС нецелесообразно, поэтому E1 реализует единый механизм для всех объектов.

Механизм сборки мусора E1 основан на построении и анализе графа ссылок между репликами распределенных объектов. Две разновидности ссылок соответствуют двум видам взаимодействия объектов: локальное взаимодейтствие между репликами распределенных объектов, находящимися в одном узле, и удаленное взаимодействие между репликами одного распределенного объекта в рамках стратегии репликации. Соответственно, выделяются **локальные ссылки** между различными распределенными объектами и **удаленные ссылки** между репликами одного объекта. В обоих случаях, как владельцем, так и объектом ссылки является реплика распределенного объекта.

Будем также разделять **сильные** и **слабые ссылки**. Слабая ссылка – это адрес интерфейса объекта или RPC-указатель на удаленную реплику, посредством которых можно выполнять, соответственно, локальные или удаленные вызовы. Слабые ссылки не отслеживаются системой сборки мусора и не учитываются при обнаружении неиспользуемых объектов. Для того чтобы превратить слабую ссылку в сильную, необходимо выполнить над ней операцию создания сильной ссылки *AddRef*. Заметим, что в E1 *AddRef* – это метод системы сборки мусора, а не объекта, на который создается ссылка, в отличие, например, от COM. В результате, в системе сборки мусора будет зарегистрирована новая сильная ссылка. Если в данном узле реплика объекта, на который указывает ссылка, еще не существует, то, как было описано в разделе 5.1, она будет создана Реестром Объектов. Система сборки мусора ассоциирует с каждой ссылкой счетчик, значение которого инкрементируется и декрементируется, соответственно, операциями *AddRef* и *Release*. Когда значение счетчика становится равным нулю, сильная ссылка уничтожается. После уничтожения последней сильной ссылки на реплику распределенного объекта, она обращается в мусор и автоматически удаляется.

В каждом узле сети система сборки мусора хранит информацию о ссылках, связанных с репликами распределенных объектов в данном узле. Для каждой реплики хранится список сильных ссылок на данную реплику, а также список ссылок, ей принадлежащих. При этом, учитываются как локальные, так и распределенные ссылки. Этих данных достаточно для того чтобы отследить любые изменения графа ссылок, в том числе, вызванные выходом из строя некоторых узлов или сетевых соединений, в то время как простой подсчет ссылок не позволяет корректно обрабатывать такие ситуации.

В управлении ссылками участвует Реестр Объектов, а также расположенные в каждом домене локальные **Мониторы Ссылок**. Реестр хранит основную часть информации о ссылках и обрабатывает такие события как создание первой и уничтожение последней сильной ссылки на реплику распределенного объекта. Кроме того, при создании и удалении

распределенных ссылок, Реестры в различных узлах взаимодействуют для внесения соответствующих изменений в свои структуры данных. Мониторы Ссылок осуществляют подсчет ссылок внутри доменов, что позволяет минимизировать количество обращений к Реестру, связанных с управлением ссылками. Монитор предоставляет интерфейс *IRefMonitor*, содержащий методы *AddRef* и *Release*. Таким образом, прикладные объекты взаимодействуют с Монитором Ссылок, который, при необходимости, вызывает методы управления ссылками Реестра.

Для сборки циклического мусора, в том числе, распределенного, в E1 используется частичная трассировка графа ссылок. Данная процедура проверяет достижимость некоторой реплики из **Корневого Набора Объектов** (КНО)[58]. К корневому набору относятся основные системные объекты E1, не являющиеся мусором по определению. Мусором также не являются все объекты, достижимые из КНО. Все остальные объекты считаются мусором. Для выполнения частичной трассировки при помощи некоторой эвристической процедуры выбирается ”подозрительная” реплика, которая становится исходной точкой обхода графа. В результате либо будет доказана достижимость исходной реплики из корневого набора, либо будет обнаружен набор реплик, образующих цикл мусора.

На рисунке 5 изображен фрагмент графа ссылок. Сильные ссылки показаны сплошными стрелками, а слабые – пунктирными. Для распределенного объекта *A* используется стратегия репликации клиент-сервер с одним основным и одним вторичным сервером. Клиентская реплика A_2 владеет сильной ссылкой на серверную реплику A_4, которая хранит состояние объекта и выполняет операции над ним. В свою очередь, A_4 владеет сильной ссылкой на вторичный сервер A_1, хранящий резервную копию объекта. Реплика A_2 взаимодействует с A_4 для выполнения операций над объектом. A_4 взаимодействует с вторичным сервером A_1 для своевременного обновления резервной копии.

Для объекта *B* используется активная репликация, при которой вызов метода рассылается всем репликам объекта, после чего каждая реплика локально модифицирует состояние объекта. Для выполнения удаленных вызовов каждая реплика владеет слабыми ссылками на все остальные реплики (в данном случае у объекта только две реплики). Сильные ссылки в данном случае не нужны. Реплика существует до тех пор, пока она используется в своем локальном узле. После этого она может быть удалена без нарушения функционирования других реплик.

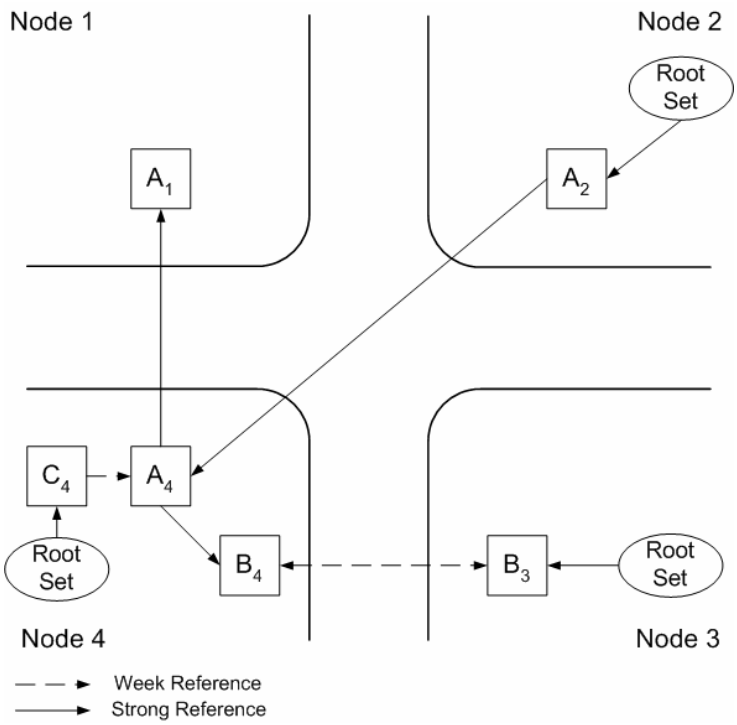


Рисунок 5. Фрагмент графа ссылок

Рассмотренный пример проясняет семантику сильных и слабых ссылок. Сильная ссылка отражает отношение зависимости между репликами, означающее, что корректное функционирование некоторой реплики или всего распределенного объекта зависит от другой реплики (того же или другого объекта). Слабая ссылка – это просто средство взаимодействия объектов. При этом, взаимодействующие реплики не зависят друг от друга и уничтожение одной из них не приводит к нарушению работы другой.

6. Репликация

В E1 эффективность доступа к объекту определяется эффективностью стратегии репликации. Для того чтобы сделать возможным использование для каждого объекта стратегии, учитывающей его семантику, архитектура E1 не накладывает каких-либо ограничений на внутреннюю организацию объекта репликации и используемые им алгоритмы. Вместо этого, ОС предоставляет набор механизмов, помогающих разработчику решить наиболее сложные задачи, возникающие при реализации большинства стратегий репликации.

6.1. Обзор стратегий репликации

В данном разделе кратко описывается несколько широко распространенных классов стратегий репликации, составляющих основу библиотеки стратегий репликации E1. Цель данного обзора – дать представление о возможностях подхода, основанного на репликации объектов. Подробное описание различных стратегий репликации читатель найдет в работах [9, 5, 28, 16, 2, 57, 19, 26].

Клиент-сервер

Клиент-сервер – простейшая стратегия репликации. Единственная копия состояния объекта хранится в *серверной* реплике (рис. 8). Остальные реплики являются *клиентами*. Все вызовы методов реплик-клиентов направляются серверу.

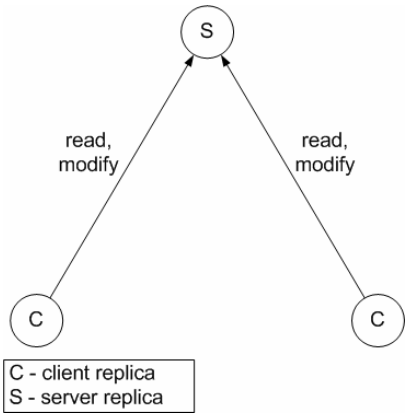


Рисунок 6. Стратегия репликации клиент-сервер

Данная стратегия в большинстве случаев является неэффективной, поскольку не обеспечивает локальность доступа к ресурсам. Другой недостаток стратегии клиент-сервер – низкая надежность как результат централизованного хранения и доступа к объекту. Существуют, однако, объекты, для которых клиент-сервер является единственной возможной стратегией репликации. В первую очередь, это объекты, представляющие различные аппаратные устройства, например, терминалы или принтеры.

Пассивная репликация

В случае пассивной репликации [9, 5] каждая реплика хранит копию состояния объекта (рис. 9). Одна из реплик назначается *первичной*. Операции чтения выполняются локально во всех узлах. Операции, модифицирующие состояние объекта, направляются первичной реплике, которая, после выполнения метода, обновляет все остальные реплики.

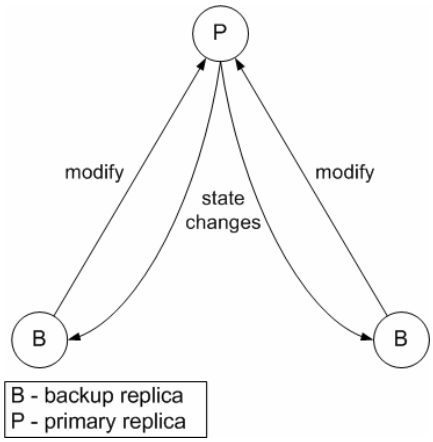


Рисунок 7. Пассивная репликация

Активная репликация

Каждая реплика хранит копию состояния объекта (рис.10). Операции чтения и модификации выполняются локально в каждом узле. Для поддержания когерентности реплик

операции модификации рассылаются всем репликам объекта, которые выполняют их над локальной копией состояния.

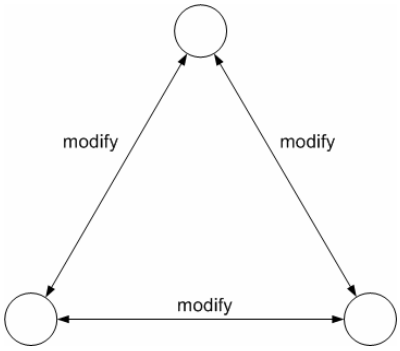


Рисунок 8. Активная репликация

К активным стратегиям репликации относится широкий класс алгоритмов, обеспечивающих различную степень согласованности реплик распределенного объекта (последовательная [28, 16], каузальная [2], временная [57], слабая [19], пассивная (lazy) [26]) согласованность.

Миграция

Под миграцией в E1 понимается перенос реплики объекта между узлами. Миграция не является самостоятельной стратегией репликации и используется в сочетании с другими стратегиями для балансировки нагрузки или обеспечения более эффективного доступа к ресурсу.

Наиболее просто мигрировать реплику, когда в ней не выполняется ни один поток. В этом случае достаточно скопировать состояние объекта в новый узел.

Иногда необходимо мигрировать реплику объекта, один или несколько методов которого выполняются длительное время или даже в течение всего времени существования объекта. Примером может являться UNIX-программа, портированная под E1. Метод main такого объекта вызывается сразу после его создания и выполняется в течение всего времени существования объекта. Для миграции данного объекта необходимо перенести из исходного узла в целевой его состояние вместе со всеми выполняющимися в объекте потоками. Как упоминалось в разделе 4.3, такая возможность предоставляется потоковой моделью E1.

6.2. Взаимодействие реплик распределенного объекта

Любая стратегия репликации предполагает наличие некоторых коммуникационных примитивов, обеспечивающих взаимодействие реплик распределенного объекта. В E1 такие примитивы предоставляются механизмом *группового удаленного вызова процедур*, позволяющим обращаться непосредственно к методам удаленных объектов репликации. В основе удаленного вызова процедур лежит механизм *группового взаимодействия*, обеспечивающий обмен однонаправленными (unicast) и широковещательными (multicast, broadcast) сообщениями между репликами распределенного объекта с определенными гарантиями надежности, упорядоченности и времени доставки.

Механизм группового взаимодействия

Система группового взаимодействия E1 включает два основных сервиса: *сервис формирования группы* и *сервис доставки сообщений*. В данном случае, группа – это множество реплик распределенного объекта. Сервис формирования группы позволяет

динамически добавлять в группу новые реплики и удалять существующие, а также отслеживает такие изменения в составе группы, вызванные сбоями в работе узлов и сетевых соединений, как уничтожение отдельных реплик, временная или перманентная фрагментация группы. Заметим, что сохранение целостности группы после сбоев – достаточно нетривиальная задача, поскольку в асинхронной среде невозможно отличить отказ узла от его временной недоступности, вызванной задержкой при доставке сообщений [41]. Следовательно, необходим распределенный алгоритм, выявляющий доступных участников группы и достигающий соглашения относительно нового состава группы среди ее выживших членов[45]. Данный алгоритм реализуется специальным компонентом в составе сервиса формирования группы – *Детектором Сбоев(ДС)*. В случае, если некоторые члены группы оказались недостижимы в результате разрыва сетевых соединений, а не отказа узлов, происходит фрагментация группы. При этом, ДС в каждом фрагменте формирует новую группу. Позднее фрагменты могут воссоединиться.

Сервис доставки сообщений предоставляет примитивы для обмена однонаправленными и широковещательными сообщениями между членами группы. Для каждого потока сообщений могут специфицироваться различные свойства протокола доставки, в первую очередь, надежность доставки и упорядоченность сообщений. В таблице 1 приведены некоторые возможные значения этих характеристик.

Свойство	**Описание**
НАДЕЖНОСТЬ ДОСТАВКИ	
Ненадежная доставка	Не предоставляет каких-либо средств контроля успешной доставки сообщения.
Атомарная доставка	Гарантирует, что каждое сообщение будет либо доставлено во все пункты назначения, либо ни в один из них
ПОРЯДОК ДОСТАВКИ	
Неупорядоченная доставка	Не накладывает каких-либо ограничений на порядок доставки сообщений
FIFO-доставка	Сообщения, отправленные одним членом группы, будут доставлены во все общие пункты назначения в порядке, соответствующем порядку отправки
Каузальный порядок	Сохраняет возможные причинно-следственные отношения [27] между сообщениями.
Полное упорядочивание	Любые два сообщения, доставляемые нескольким членам группы, будут доставлены каждому из них в одинаковом порядке

Таблица1. Свойства доставки сообщений

Разработка системы группового взаимодействия с нуля – весьма сложная задача, включающая имплементацию ряда распределенных алгоритмов доставки сообщений и управления составом группы. Поэтому механизм группового обмена сообщениями Е1 планируется построить на основе одной из существующих имплементаций. На сегоднящний день описанные выше сервисы реализованы в ряде систем группового взаимодействия [6, 40, 3]. Такие системы изначально ориентированы на использование в составе более сложного программного комплекса, в первую очередь, в качестве средства поддержки реплицированных сервисов. Поэтому они могут быть легко интегрированы в Е1. Кроме того, они обладают высокой модульностью, позволяющей легко расширять их поддержкой новых свойств доставки сообщений [40].

Групповой удаленный вызов процедур

Примитивы обмена сообщениями являются основой взаимодействия реплик распределенного объекта. Однако, для разработчика стратегии репликации более удобной и естественной является процедурная модель, позволяющая обращаться непосредственно к методам удаленного объекта репликации. В случае двухточечного взаимодействия для выполнения операций над удаленными объектами используется механизм удаленного вызова процедур (RPC). Обобщением RPC на случай группового взаимодействия является *групповой удаленный вызов процедур(GRPC)*. На основе группового обмена сообщениями GRPC реализует единый примитив, позволяющий вызывать методы сразу нескольких удаленных объектов и обрабатывать результаты вызова. Архитектура механизма GRPC E1 показана на рисунке 6.

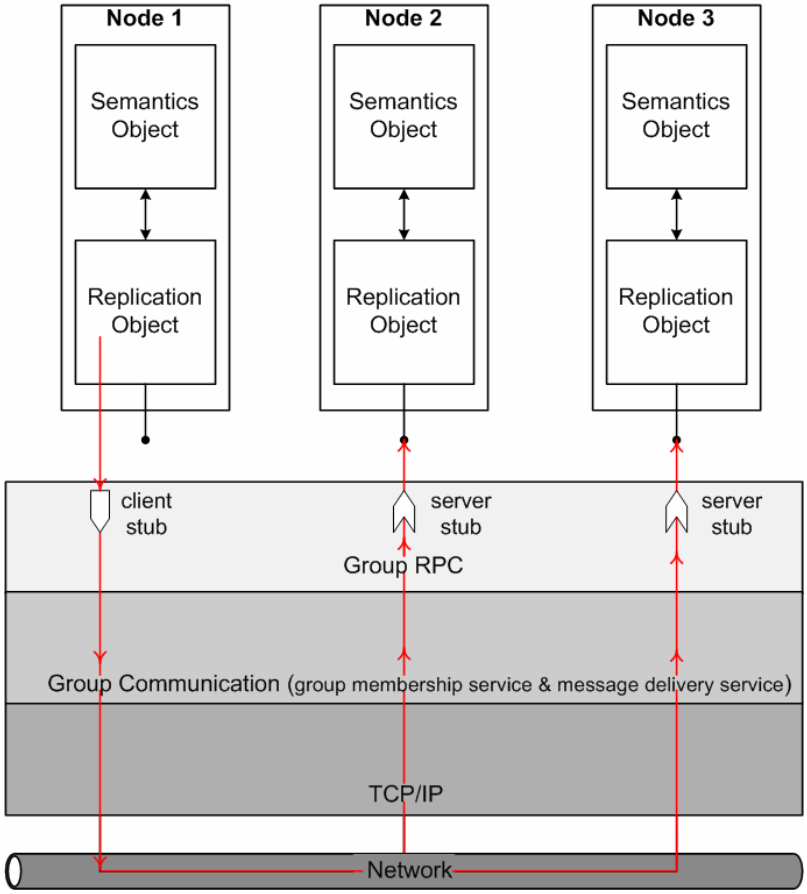


Рисунок 9. Выполнение удаленного вызова посредством механизма GRPC.

Как и в обычном RPC, в GRPC объекты взаимодействуют посредством клиентских и серверных стабов. Стабы компилируются автоматически из IDL-описания объекта (см. раздел 7.1). Клиентские стабы локально предоставляют интерфейсы удаленных объектов репликации. Каждый вызов клиентского стаба преобразуется в сообщение, которое при помощи механизма группового взаимодействия отправляется одной или нескольким удаленным репликам. Сообщение доставляется серверному стабу, который преобразует его в вызов соответствующего метода объекта репликации. Результат вызова отправляется вызывающей реплике, где он анализируется клиентским стабом. После получения необходимого числа ответных сообщений (определяемого семантикой вызова), клиентский стаб возвращает управление вызывающему объекту.

6.3. Изменение состава группы

Частью любой стратегии репликации является протокол взаимодействия реплик распределенного объекта в момент изменения его структуры. Такое изменение может быть инициировано добавлением или удалением реплики из состава распределенного объекта, а также фрагментацией объекта вследствие нарушения работы узлов или сетевых соединений. Как обсуждалось выше, на уровне системы обмена сообщениями такие ситуации обрабатываются сервисом формирования группы, который доставляет всем репликам согласованную информацию о составе распределенного объекта. Далее, система группового взаимодействия уведомляет объекты репликации об изменении структуры объекта посредством интерфейса обратного вызова IReplicaGroup. Обработка данного события может включать сложное распределенное взаимодействие между объектами репликации, в том числе, синхронизацию состояния объекта, создание и удаление распределенных ссылок и даже создание или удаление реплик объекта. В результате формируется новая конфигурация распределенного объекта, отвечающая требованиям целостности используемой стратегии репликации.

Рассмотрим, например, сценарий восстановления распределенного объекта после фрагментации сети (рис. 10). Исходная конфигурация объекта (рис. 10a) включает клиентскую реплику *C*, первичный сервер *P* и вторичный сервер *B*. Стрелками показаны сильные ссылки между репликами. В результате выхода из строя сетевого соединения объект распадается на три фрагмента (рис. 10b). Если стратегия репликации не предусматривает возможности воссоединения фрагментов в будущем и не предполагает каких-либо специальных действий, направленных на сохранение целостности объекта, то все его реплики будут удалены. Действительно, на серверные реплики (*P* и *B*) не остается ни одной сильной ссылки, а клиентская реплика не владеет копией сотояния объекта и не может успешно обрабатывать вызовы методов. Предположим, однако, что в данном примере используется вариант стратегии репликации, учитывающий возможность фрагментации. Первичная серверная реплика искусственно создает на себя сильную ссылку из корневого набора для того чтобы не быть уничтоженной системой сборки мусора (рис. 10c). Кроме того, поскольку данная стратегия репликации предусматривает наличие вторичного сервера, хранящего копию состояния объекта, первичный сервер создает дополнительную вторичную реплику. Все обращения к клиентской реплике возвращают ошибку, указывающую на фрагментацию объекта. Таким образом, только вторичный сервер *B* будет удален как мусор. Если впоследствии сетевое соединение будет восстановлено, выжившие реплики объекта могут воссоединиться (рис. 10d) и продолжить нормальное функционирование.

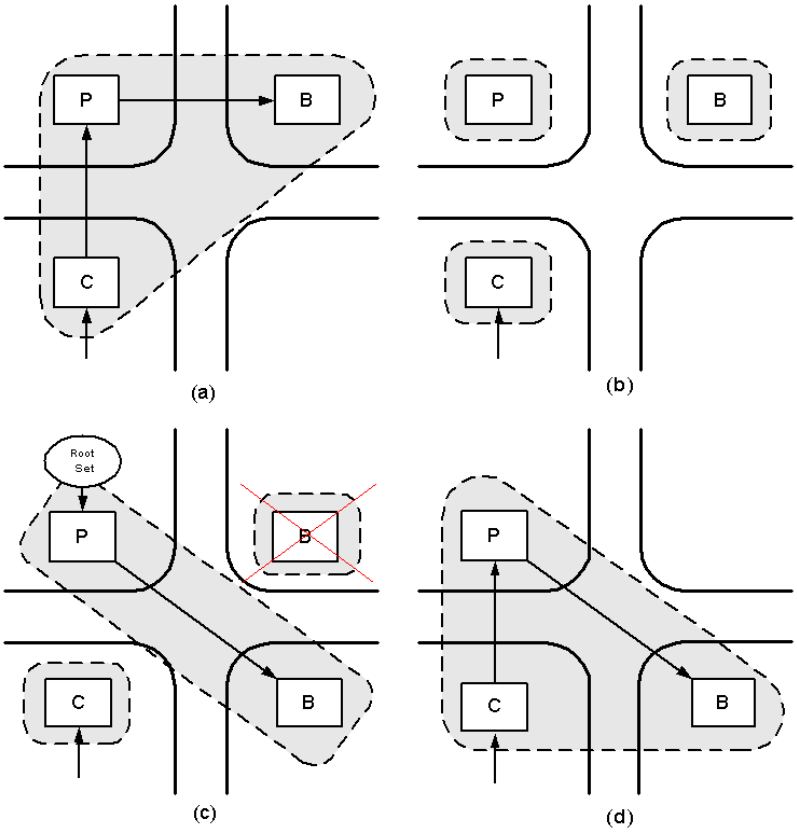


Рисунок 10. Фрагментация распределенного объекта

Заметим, что сразу после фрагментации (рис. 10b) реплики *P* и *B* распределенного объекта могли быть уничтожены как мусор. Для того чтобы стратегия репликации имела возможность корректно обрабатывать подобные ситуации, вводится понятие *переходного состояния* объекта. В тот момент, когда система группового взаимодействия нотифицирует объект репликации об изменении состава группы, реплика переводится в переходное состояние, в котором она не может быть уничтожена системой сборки мусора. Находясь в переходном состоянии, объект может произвольным образом изменять свою структуру, например, настраивать сильные ссылки между репликами или, как в данном примере, создавать внешние ссылки на свои реплики. После завершения всех необходимых действий, объект возвращается в нормальное состояние.

6.4. Интерфейс сериализации

Рассмотрим еще один важный аспект разработки распределенных объектов. Любая стратегия репликации использует некоторый набор операций, которые объект репликации может выполнить, только взаимодействуя с реплицируемым объектом семантики. Наиболее важными примерами являются сериализация и десериализация состояния объекта. Эти операции используются практически всеми стратегиями репликации при копировании состояния объекта из существующей реплики в новую, а также для синхронизации реплик. Объект репликации не располагает информацией о структуре объекта семантики, поэтому каждый объект семантики в E1 должен предоставлять интерфейс *ISerializable*, содержащий методы сериализации и десериализации. ISerializable является аналогом интерфейса Checkpointable в CORBA, также предназначенного для поддержки репликации объектов [35].

Таким образом, задача сериализации состояния распределенного объекта возлагается на разработчика объекта семантики. Заметим, что, вообще говоря, процедуры сериализации/десериализации могут оказаться весьма громоздкими, особенно для объектов сложной структуры. Поэтому желательно генерировать их автоматически. Это сложная задача, не имеющая универсального решения, которое было бы достаточно эффективно для любых типов объектов и при этом могло применяться для различных языков программирования.

Существуют классы языков, для которых возможно реализовать автоматическую сериализацию/десериализацию объектов. К ним относятся языки, хранящие во время выполнения информацию о типах, например, Java и C#. Авторы предвидят, что такие языки будут широко использоваться для прикладного программирования в E1.

Однако, наряду с ними, должны поддерживаться и другие языки, в частности, C++, для которого ни на этапе компиляции программы, ни на этапе выполнения невозможно в общем случае автоматически проанализировать структуру объекта и произвести его сериализацию [48]. Поэтому хотелось бы разработать не зависящий от языка программирования метод генерации интерфейса ISerializable. В E1 поддержка автоматической сериализации объектов предоставляется системой управления памяти. Каждый локальный объект состоит из статической части, а также динамически выделяемых данных. Для динамического выделения памяти используются объекты типа *Куча* (heap). Куча описывает непрерывный участок виртуального адресного пространства и предоставляет примитивы для динамического выделения и высвобождения памяти внутри этого участка. Каждый домен предоставляет локальную кучу, используемую по умолчанию всеми находящимися в нем объектами. Кроме того, любой объект может создать отдельную кучу и выделять память только из нее. Для сериализации такого объекта достаточно запаковать в некоторую структуру данных все участки памяти, выделенные данным объектом из кучи, с указанием их виртуальных адресов. При десериализации объекта создается точная копия исходной кучи в новом узле. Таким образом, сериализация/десериализация объекта семантики сводится к сериализации/десериализации кучи. В отличие от подхода, основанного на свойствах языка программирования, такой способ является универсальным. Однако, использование отдельной кучи для объекта семантики означает, что в физической памяти динамические данные объекта занимают целое число страниц, что понижает эффективность использования памяти.

Рисунок 11 иллюстрирует различия между двумя представленными подходами. На рисунке 11a показана сериализация объекта, разработанного при помощи языка программирования, хранящего во время выполнения информацию о типах. Для динамического выделения памяти такой объект использует общую кучу. Во время сериализации средства поддержки выполнения языка анализируют граф ссылок внутри объекта и запаковывают его состояние. При десериализации структура объекта восстанавливается в локальной куче целевого узла, при этом отдельные структуры данных могут располагаться по адресам, отличным от их положения в исходном узле, но ссылочная целостность объекта сохраняется. На рисунке 11b показана сериализация объекта, написанного на языке C++ и использующего собственную кучу. После десериализации все динамически выделенные структуры данных располагаются по тем же адресам, что и в исходном узле.

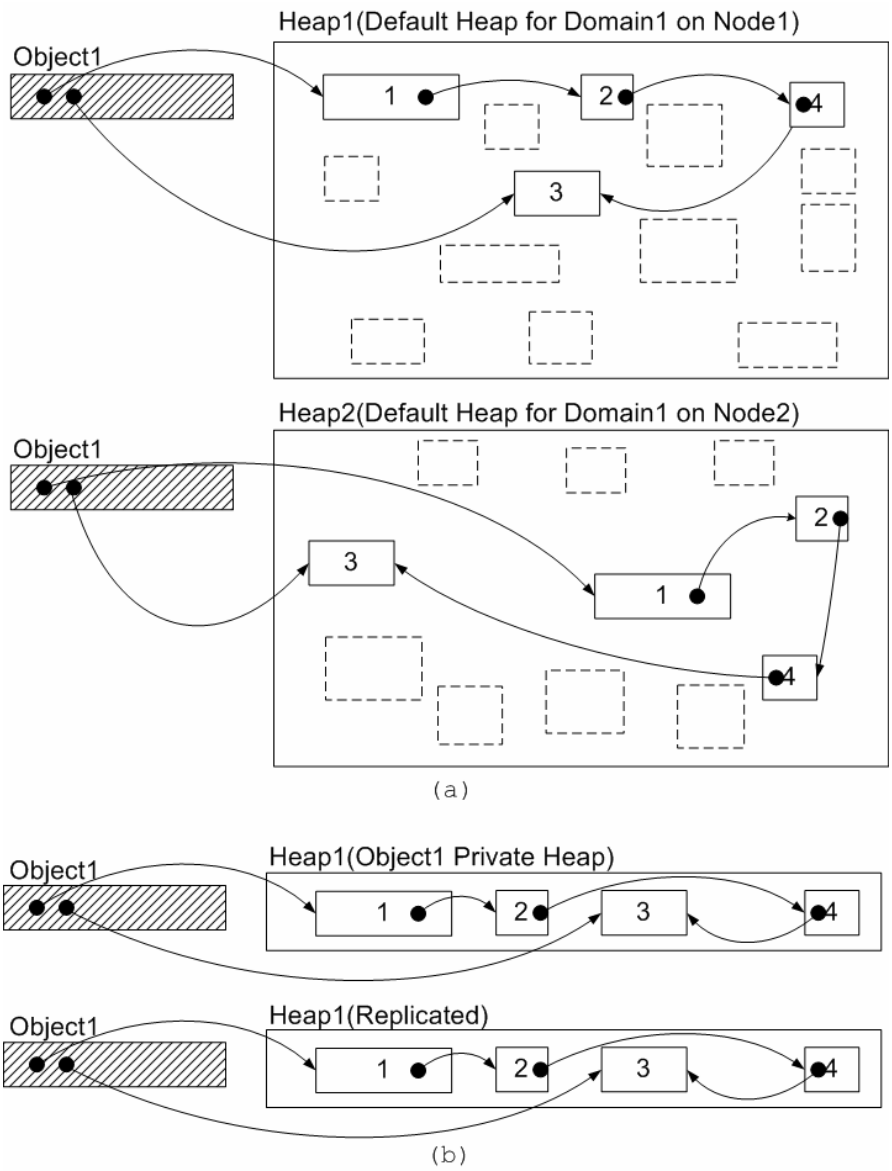


Рисунок 11. Два способа автоматической сериализации/десериализации объекта семантики. а. сериализация средствами языка программирования б. использование собственной кучи для динамического выделения памяти.

В случаях, когда язык программирования не позволяет реализовать автоматическую сериализацию состояния объекта, а использование собственной кучи для каждого объекта приводит к недопустимому расходу физической памяти, разработчик объекта самостоятельно имплементирует интерфейс ISerializable. По-видимому, данный подход будет использован для ряда системных объектов E1.

7. Программирование в E1

Программное обеспечение E1 состоит из распределенных реплицированных объектов. В данном разделе представлен взгляд на E1 со стороны разработчика объектов. Мы в общих чертах рассмотрим методологию создания распределенных объектов, а также необходимые для этого инструменты.

Распределенный объект состоит из объектов семантики и объектов репликации, разрабатываемых, как правило, различными сторонами. В большинстве случаев, разработчик объекта имплементирует объект семантики и указывает одну или несколько стандартных стратегий репликации, которые могут для него использоваться. Реже стратегия репликации разрабатывается специально для конкретного объекта семантики.

E1 поддерживает компонентную модель разработки ПО. Для этого в состав ОС включены сервисы, расширяющие модель распределенных объектов до компонентной модели: Реестр Объектов, Сервер Контроля Доступа, Сервер Глобального Именования, система сборки мусора. Частью компонентной модели E1 является также **язык описания интерфейсов (IDL)**, делающий возможной бинарную интероперабельность компонентов, написанных на различных языках программирования.

На рисунке 12 схематически показан цикл разработки распределенного объекта с использованием языка C++. В разработке участвует две стороны – сторона, реализующая стратегию репликации, и сторона, реализующая локальный объект семантики, т.е. собственно разработчик компонента. Разработчик объекта семантики составляет IDL-описание интерфейсов объекта, на основании которых компилятор IDL генерирует абстрактные классы C++, от которых наследуется имплементация объекта. IDL-описание объекта содержит следующую информацию:

- описание структур данных, используемых в качестве параметров или возвращаемых значений методами объекта;
- уникальный идентификатор класса объекта, а также идентификаторы его интерфейсов;
- описание методов каждого интерфейса: имя метода, типы аргументов и возвращаемых значений, направление передачи аргументов, а также дополнительная метаинформация, которая может использоваться при компиляции объекта репликации. Например, для метода может быть указан один из атрибутов: [read],[modify].

Реализация стратегии репликации может выполняться независимо от объекта семантики. Очевидно, однако, что объект репликации для конкретного объекта семантики может быть скомпилирован только при наличии информации об интерфейсах последнего. Это противоречит существованию универсальных стратегий репликации, таких как активная репликация, которые могут использоваться для самых различных объектов. Разрешение противоречия заключается в том, чтобы описывать стратегию репликации на специальном языке сценариев, использующем некоторые абстрактные конструкции для обозначения интерфейсов и методов реплицируемого объекта семантики. Собственно объект репликации может генерироваться на основании такого описания и IDL-спецификации объекта семантики. Компиляция объекта репликации может осуществляться автоматически, непосредственно в тот момент, когда данная стратегия репликации впервые применяется для некоторого класса объектов семантики. Для этого используется т.н. *Компилятор Стратегий Репликации*, являющийся частью E1. Вместе с объектом репликации Компилятор Стратегий Репликации генерирует GRPC-стабы, необходимые для удаленного взаимодейсивия реплик распределенного объекта.

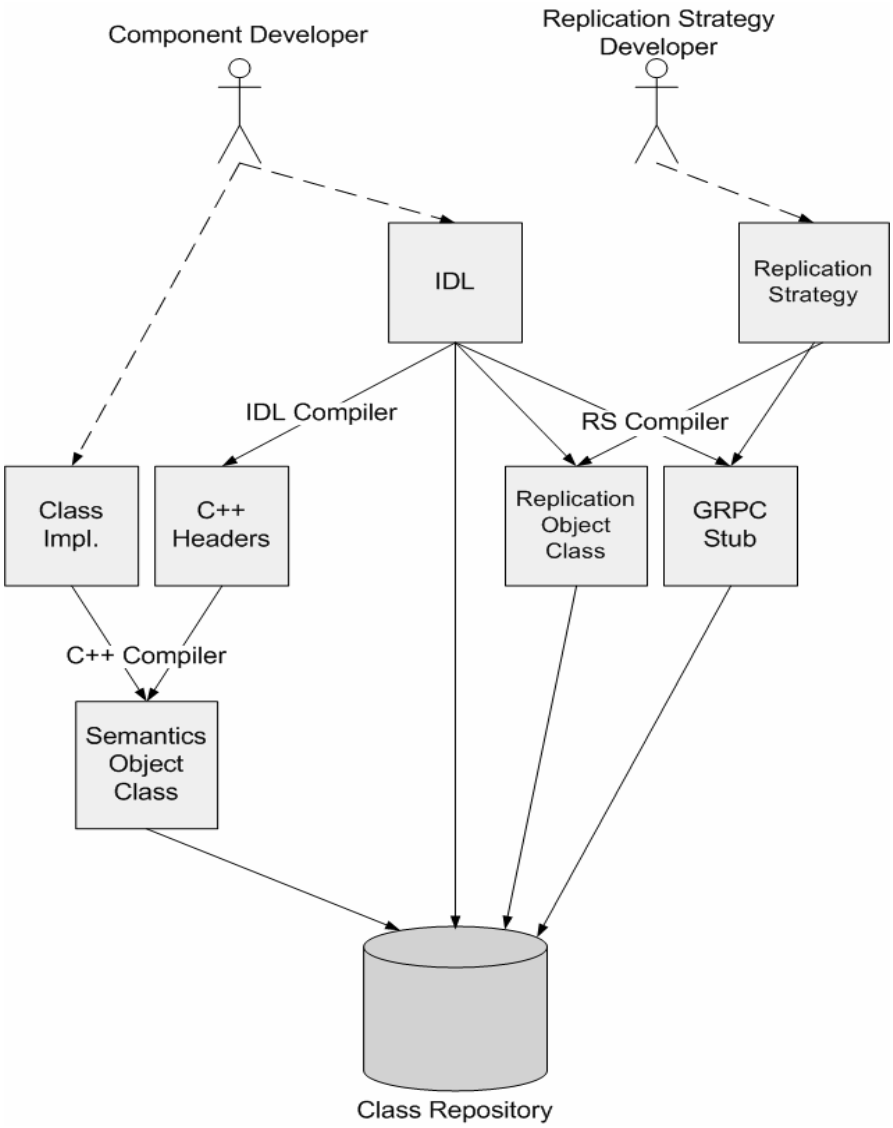


Рисунок 12. Цикл разработки распределенного объекта

Распределенные объекты могут разрабатываться не только при помощи языка C++, но и любых других языков программирования, для которых может быть построено отображение в объектную модель E1 и реализованы соответствующие компиляторы. Как и в других компонентных системах, для разработки объектов E1 могут применяться компонентно-ориентированные языки программирования. Эти языки лучше, чем C++, адаптированы для разработки компонентного ПО.

Заключение

В работе представлена архитектура распределенной операционной системы E1. Главной особенностью E1 является то что все компоненты ОС, а также прикладное программное обеспечение состоит из распределенных реплицированных объектов. Интерфейсы распределенного объекта глобально доступны из всех узлов сети, что делает распределенную природу вычислительной системы прозрачной для прикладных программистов и конечных пользователей. Возможность выбора для каждого объекта стратегии репликации с учетом его семантики позволяет достичь максимальной

эффективности доступа при требуемом уровне надежности. Внутренняя архитектура распределенного объекта позволяет эффективно разделить семантику объекта и алгоритм репликации, благодаря чему задача разработки реплицированного объекта оказывается эквивалентной разработке аналогичного централизованного объекта.

Реализация распределенных протоколов доступа к объекту осуществляется разработчиками стратегий репликации. Большинство стратегий являются универсальными, т.е. могут применяться для широких классов объектов. Однако, стратегия репликации может разрабатываться и для объектов конкретного типа, что позволяет в полной мере учесть особенности данного объекта и обеспечить максимально эффективный доступ к нему. E1 предоставляет средства, облегчающие разработку объектов репликации, в том числе, механизм группового взаимодействия реплик распределенного объекта и механизм автоматической сериализации/десериализации состояния объекта.

На нижнем уровне архитектуры E1 находится микроядро, поддерживающее минимальный набор примитивов, необходимых для построения операционной системы: адресные пространства, потоки выполнения, базовый механизм межпроцессного взаимодействия, диспетчеризация прерываний. Все системные сервсисы и приложения состоят из распределенных объектов, функционирующих на прикладном уровне. Такая архитектура позволяет повысить модульность и надежность системы, а также значительно уменьшить задержки при передаче управления через ядро, что особенно важно для систем, ориентированных на интенсивное взаимодействие компонентов.

Все объекты в E1 размещаются в едином 64-битном адресном пространстве, разбитом на изолированные области – домены защиты. Виртуальный адрес объекта является его глобально уникальным идентификатором, посредством которого другие объекты могут с ним взаимодействовать. Таким образом, использование единого адресного пространства позволяет реализовать простую и удобную модель взаимодействия объектов.

Модель выполнения E1 основана на концепции мигрирующих потоков, в соответствии с которой поток не связан перманентно с каким-либо объектом или доменом, а перемещается между объектами посредством вызова методов интерфейсов. По сравнению с использованием статических потоков, модель мигрирующих потоков упрощает разработку объектов, позволяет сделать их более компактными и обеспечивает более эффективную передачу управления между объектами.

На сегодняшний день для разработки распределенных приложений чаще всего применяется компонентный подход, заключающийся в том чтобы строить программные системы из готовых бинарных компонентов. Использование такого подхода предполагает наличие компонентной модели, описывающей набор сервисов, интерфейсов и соглашений, определяющих требования к компонентам, а также среду функционирования и взаимодействия компонентов. Как правило, реализация компонентной модели выполняется в виде промежуточного ПО. В E1 компонентная модель может быть построена без использования дополнительных программных уровней, путем расширения модели распределенных объектов необходимыми сервисами и инструментами. Помимо простоты и эффективности, важным преимуществом такой архитектуры является удобство разработки и использования компонентов, ведь для E1 распределенный объект (или компонент) является таким же фундаментальным понятием как, например, файловая абстракция в UNIX. Компонентная модель E1 предоставляет средства глобального именования и защиты объектов, механизм динамической загрузки классов, систему сборки мусора, а также инструменты разработки компонентов, такие как компилятор языка IDL и компилятор стратегий репликации.

Дальнейшая работа над E1 включает расширение описанной архитектуры механизмом персистентности распределенных объектов, после чего планируется перейти к ее имплементации и последующему анализу полученных результатов.

Библиография

1. M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, Jr. A. Tevanian, M. W. Young. Mach: A New Kernel Foundation for UNIX Development. *Proc. of the Summer 1986 USENIX Conference* pp.93-113, July 1986.
2. Ahamad, M. Raynal, G. Thia-Kime. An Adaptive Protocol for Implementing Causally Consistent Distributed Services. *Proc. 18th International Conference on Distributed Computing Systems*, Amsterdam, pp.86-93, 1998.
3. Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A communication sub-system for high availability. *Proc. 22nd IEEE Fault -Tolerant Computing Symposium (FTCS)*, July 1992.
4. J. S. Barrera. A fast Mach network IPC implementation. *Proc. of the Second USENIX Mach Symposium*. pp.1-12, 1991.
5. K. P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.
6. K. P. Birman, T. A. Joseph, Exploiting Virtual Synchrony in Distributed Systems, *Proc. 11th ACM Symp. on Operating Systems Principles*, pp. 123-138, Austin, TX, November 1987.
7. M. Bishop, L. Snyder. The transfer of information and authority in a protection system. *Proc. 17th ACM Symposium on Operating Systems Principles*, 1979.
8. J. Biskup. Some variants of the take-grant protection model. *Information Processing Letters*. 3, №19. 1984.
9. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, Distributed Systems, ACM Press Books, chapter 8, pages 199–216, Addison-Wesley, second edition, 1993.
10. J. S. Chase. An Operating System Structure for Wide-Address Architectures. PhD Thesis, Department of Computer Science and Engineering, University of Washington, August 1995.
11. P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. Jr. LeBlanc, W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, C. J. Wileknloh. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems Journal*, Vol 3, *USENIX*, Winter 1990.
12. A. Dearle, R. di Bona, J. M. Farrow, F. A. Henskens, A. Lindström, J. Rosenberg, F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System, Technical Report, Departments of Computer Science, Universities of Adelaide and Sydney, 1993.
13. P. Dechamboux, J.-P. Fassino, D. Hagimont, J. Mossière, X. Rousset. The ARIAS Distributed Shared Memory: an Overview. *SOFSEM Seminar*, Prague, November 1996.
14. K. Elphinstone, G. Heiser, L4 Reference Manual, Technical Report UNSW-CSE-TR-9709, School of Computer Science and Engineering, University of New South Wales, December 1997.
15. K. Elphinstone, S. Russell, G. Heiser. Supporting Persistent Object Systems in a Single Address Space. Technical Report 9601, School of Computer Science and Engineering, The University of New South Wales, February 1996.
16. A. Fekete, M. F. Kaashoek, N. A. Lynch. Implementing Sequentially Consistent Shared Objects Using Broadcast and Point-to-Point Communication *Proc. International Conference on Distributed Computing Systems* pp.439-449, 1995.
17. B. Ford, J. Lepreau. Evolving Mach 3.0 to use migrating threads. Technical Report UUCS-93-022, University of Utah, August 1993.
18. B. Ford, J. Lepreau. Microkernels Should Support Passive Objects, *Proc. I-WOOOS'93*, December 1993.
19. R. Golding. Weak-Consistency Group Communication and Membership. PhD thesis, University of California, Santa Cruz, 1992.

20. Li Gong. A Secure Identity-Based Capability System. *Proc. IEEE Symposium on Security and Privacy*, pp.56-65, 1989.
21. G. Heiser, K. Elphinstone, S. Russell, J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report 9314. School of Computer Science and Engineering, The University of New South Wales, 1993.
22. G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9), July 1998.
23. P. Homburg. The Architecture of a Worldwide Distributed System. PhD thesis Vrije University, Advanced School of Computing and Imaging, Amsterdam, 2001.
24. A. K. Jones, R. J. Lipton, and L. Snyder. A Linear Time Algorithm for Deciding Security. *Proc. 17th Symposium on Foundations of Computer Science*, Houston, Texas, pp. 33-41, 1976.
25. The L4Ka team. L4 experimental kernel reference manual, version X.2. February 2002.
26. R. Ladin, B. Liskov, L. Shirira, S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10:4, pp.360-391, 1992.
27. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM* 21:7, pp.558-565, July 1978.
28. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.
29. K. Li Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale, September 1986.
30. J. Liedtke. On μ-Kernel Construction. *Proc. of the 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, pp.237-250, 1995.
31. J. Liedtke. L4 reference manual (486, Pentium, Pro). Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.
32. J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. Achieved IPC performance (still the foundation for extensibility). *Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pp.28–31, Chatham (Cape Cod), MA, May 1997.
33. Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification. version 0.9., October 1995.
34. P. Mockapetris, K. J. Dunlap. Development of the Domain Name System. *Proc. ACM SIGCOMM*, Stanford, CA, 1988.
35. Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. version 2.6.1, May 2002.
36. Object Management Group (OMG). CORBAservices: Common Object Services Specification, 1998.
37. Object Management Group (OMG). Life Cycle Service Specification. version 1.2. September 2002.
38. D. Plainfosse, M. Shapiro A Survey of Distributed Garbage Collection Techniques *Proc. Int. Workshop on Memory Management Kinross*, Scotland (UK), September 1995.
39. D. Potts, S. Winwood, G. Heiser. L4 Reference Manual: Alpha 21x64. Technical Report UNSW-CSE-TR-0104, University of New South Wales, Sydney, March 2001.
40. R. van Renesse, K. P. Birman, B. Glade, K. Guo, M. Hayden, T. M. Hickey, D. Malki, A. Vaysburd, W. Vogels. Horus: A Flexible Group Communication Subsystem. Technical Report TR 95-1500, Cornell University, Ithaca, NY, 1995.
41. A. Ricciardi, A. Schiper, K. Birman. Understanding Partitions and the "No Partition" Assumption. *Proc. 4th IEEE Workshop on Future Trends of Distributed Systems*, Lisboa, September 1993.
42. D. M. Ritchie, K. Thompson. The UNIX time sharing system. *Comm. ACM 17:7*, pp 365-375, July 1974.

43. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, 1988.
44. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, Volume 29, No 2, pp.38-47, February 1996.
45. A. Schiper, A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. *Proc. 23rd International Symposium on Fault-Tolerant Computing Systems*, Toulouse, France, pp.534–543, June 1993.
46. F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4). pp.290–319, December 1990.
47. M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pp.198-204, May 1986.
48. M. Shapiro, P. Gautron, L. Mosseri. Persistence and Migration for C++ Objects. *Proc. Third European Conference on Object-Oriented Programming*, pp.191-204, 1989.
49. A. C. Skousen, SOMBRERO: A Very Large Single Address Space Distributed Operating System. MS Thesis, Computer Science and Engineering Department, Arizona State University, December 1994.
50. M. Van Steen, P. Homburg, A. S. Tanenbaum Globe: A Wide-Area Distributed System. *IEEE Concurrency,* pp.70–78 January-March 1999.
51. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.
52. B. Stroustrup. The C++ Programming Language (3rd Edition). Addison-Wesley, 1997.
53. Sun Microsystems, Inc. EJB specification 2.0.
54. C. Szyperski Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 1998.
55. A. S. Tanenbaum, M. F. Kaashoek, R. van Renesse, H. Bal. The Amoeba Distributed Operating System - A Status Report. *Computer Communications,* vol. 14, pp.324-335, July/August 1991.
56. A. S. Tanenbaum, S. J. Mullender, R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. *Proc. Sixth International Conference on Distributed Computing Systems,* IEEE, pp. 558-563, 1986.
57. F. Torres M. Ahamad, M. Raynal. Timed Consistency for Shared Distributed Objects. *Proc 18th ACM Int Symposium on Principles of Distributed Computing (PODC 99)*, Atlanta, pp 163-172, 1999.
58. P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.