РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

САБИТОВ КИРИЛЛ

СТ М3235, Itmo University, Russia, St. Petersburg
Email address: sabitovkirill@ya.ru
URL: t.me/ne1mnn

Abstract. В эру, когда технологический прогресс критически зависит от способности обрабатывать и анализировать огромные объемы данных, распределенные системы выступают в качестве краеугольного камня как

- в **пользовательской** среде - обучение моделей исскуственного интелекта в самых разных областях, серверные приложения с большим количеством сервисов, распределенные базы данных, blockchain
- так и в **научной** среде - обработке данных из экспериментов, таких как проводимые в Европейском центре ядерных исследований (CERN), где они используются для анализа результатов столкновений элементарных частиц, физически корректного моделирования в астрофизике, климатологии и других науках, где требуются сложные вычисления для моделирования и понимания явлений мира.

Таким образом, распределенные вычисления становятся не просто технологией, а основополагающим инструментом для дальнейшего развития и реализации технического прогресса, открывая новые горизонты в научных открытиях и пользовательском опыте. В связи с их значимостью и необходимостью для современного мира, **целью данной работы** является осветить основные моменты разработки данных технологий на примере уже известных подходов с точки зрения курса ОС.

Содержание

1. Концепции архитектуры распределенных систем

Перед более подробным рассмотрением подходов к реализации распределенных систем рассмотрим ключевые концепции их архитектур, а так же рассмотрим какие проблемы они решают:

1.1. **Совместное использование ресурсов.** Концепция совместного использования ресурсов (Resource sharing) заключается в обеспечении совместного доступа к ресурсам, которые могут быть распределены по разным узлам системы. Разделение ресурсов позволяет улучшить эффективность

использования ресурсов, обеспечивает масштабируемость и повышает доступность системы для пользователей.

Разделение ресурсов в распределенных системах предполагает, что ресурсы (как аппаратные — хранилища данных, принтеры, сканеры, и т.д., так и программные) распределены по сети и доступны для использования разными узлами или клиентами. Это может включать в себя общий доступ к файлам, принтерам, информационным базам данных и другим ресурсам.

Преимущества такого подхода включают в себя:

- Экономию ресурсов: общий доступ к дорогостоящим ресурсам позволяет избежать необходимости дублирования этих ресурсов в каждом узле.
- Масштабируемость: распределенные системы легко масштабируются, так как добавление новых ресурсов или узлов не требует значительных изменений в архитектуре.
- Надежность и доступность: распределенные системы могут быть более устойчивыми к отказам. Если один узел выходит из строя, другие узлы могут продолжать работу.
- Упрощение совместной работы: позволяет географически разбросанным группам людей работать вместе, используя групповое программное обеспечение для совместного редактирования, телеконференций и т.д.

1.2. **Distribution transparency.**

Прозрачность распределения заключается в скрытии факта физического распределения процессов и ресурсов по множеству компьютеров, которые могут быть разделены большими расстояниями. Она делает распределение процессов и ресурсов невидимым для конечных пользователей и приложений. Достигается это с помощью так называемого промежуточного программного обеспечения (middleware) (более подробно об этом будет в `следующей части`).

Различают разные аспекты прозрачности распределения, каждый из которых решает свои уникальные задачи:

- Прозрачность доступа (Access Transparency): Обеспечивает скрытие различий в доступе к локальным и удаленным ресурсам. Это позволяет пользователю обращаться к любым ресурсам, не заботясь о их физическом расположении.
- Прозрачность местоположения (Location Transparency): Скрывает физическое местоположение ресурсов и процессов в системе. Пользователи и приложения могут взаимодействовать с ресурсами, не зная их реального расположения.
- Прозрачность миграции (Migration Transparency): Позволяет перемещать ресурсы и компоненты внутри системы без влияния на работу пользователей. Это полезно для балансировки нагрузки и оптимизации производительности системы.
- Прозрачность репликации (Replication Transparency): Скрывает наличие копий (реплик) ресурсов или компонентов от пользователей

и приложений. Это обеспечивает повышение доступности и отказоустойчивости системы.
- Прозрачность параллелизма (Concurrency Transparency): Позволяет нескольким процессам работать с ресурсами параллельно, скрывая от них детали синхронизации доступа.
- Прозрачность отказов (Failure Transparency): Способность системы скрывать отказы отдельных компонентов, обеспечивая непрерывную работу приложений.

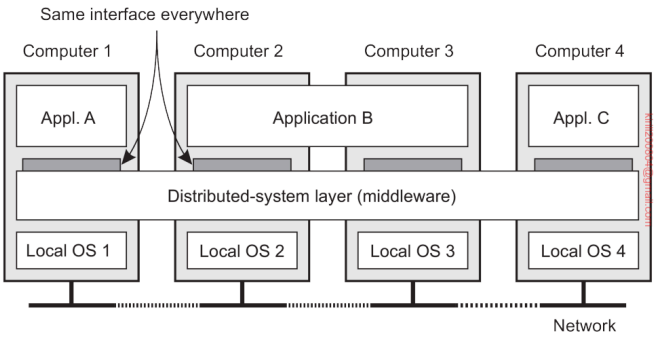
Так же важно упомянуть, что Distribution transparency применимо не всегда, например может быть существеннен факт различия в часовых поясах узла системы и конечного пользователя, что не возможно определить, если считать все узлы единым целым.

1.3. **Промежуточный уровень (Middleware).**

Middleware в контексте распределенных систем - это программное обеспечение, которое обеспечивает связь и управление данными между различными частями распределенной системы. Его главная роль - облегчить сложность взаимодействия между компонентами системы, предоставляя абстракцию и управление ресурсами.

Исторически, развитие middleware связано с потребностью в эффективном управлении распределенными системами и решением проблем, связанных с гетерогенностью оборудования и программного обеспечения. Middleware позволяет разным приложениям и системам взаимодействовать друг с другом, скрывая техническую сложность и различия в сетевых протоколах, платформах и языках программирования.

Это решает проблемы, связанные с интеграцией различных систем, обеспечением безопасности, управлением транзакциями и предоставлением универсального доступа к ресурсам и службам. В современном контексте, middleware играет ключевую роль в облегчении создания и управления сложными распределенными системами, такими как облачные вычисления, большие данные и интернет вещей. Проще говоря middleware обеспечивает Distribution transparency.



Использования промежуточного уровня (middleware) для достижения прозрачности доступа

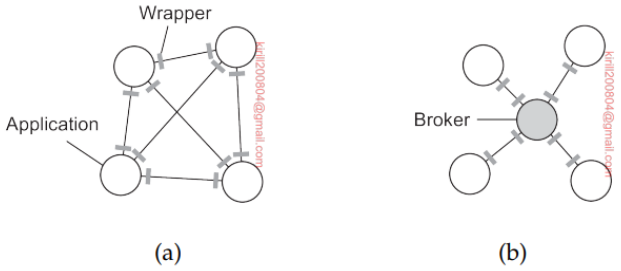
Теперь рассмотрим организацию промежуточного уровня, для этого чаще всего используют 2 паттерна — *wrapper*, *interceptor*, выполняющих различные задачи.

1.3.1. *Wrappers.*

Обертки (wrappers) – паттерн построения middleware распределенных систем, для интеграции существующих компонентов с разными интерфейсами. Они обеспечивают адаптацию интерфейсов, позволяя клиентским приложениям взаимодействовать с различными компонентами системы. Это решает проблему несовместимости интерфейсов, трансформируя функции клиентского приложения в те, которые доступны в компоненте.

В контексте распределенных систем обертки выходят за рамки простых трансформаторов интерфейсов. Например, адаптер объекта может позволить приложениям вызывать удаленные объекты, реализованные с помощью библиотечных функций, работающих с таблицами реляционной базы данных.

Для уменьшения количества оберток, используется middleware и такие компоненты, как брокеры. Брокер - это централизованный компонент, который обрабатывает доступ к различным приложениям. Например, брокер сообщений, описанный в разделе 4.3.3, принимает запросы от приложений и обеспечивает необходимое взаимодействие и трансформацию данных. Это позволяет сократить количество необходимых оберток с $O(N^2)$ до $O(N)$, упрощая масштабирование и интеграцию в распределенных системах.



(a). Обертка для доступа приложений к функционалу других узлов.
(b). Использование брокера.

1.3.2. *Interceptors.*

Перехватчики (interceptors) являются важным паттерном, позволяющим адаптировать middleware к конкретным потребностям приложений. Прерывания изменяют обычный поток управления и позволяют выполнять специфический для приложения код, что делает middleware позволяют выполнять спецефичные для приложения задачи, делая его более открытым.

Что бы более детально разобраться в том, как перехватыватели выполняют поставленную задачу рассмотрим процесс обработки вызова методов удаленного узла:

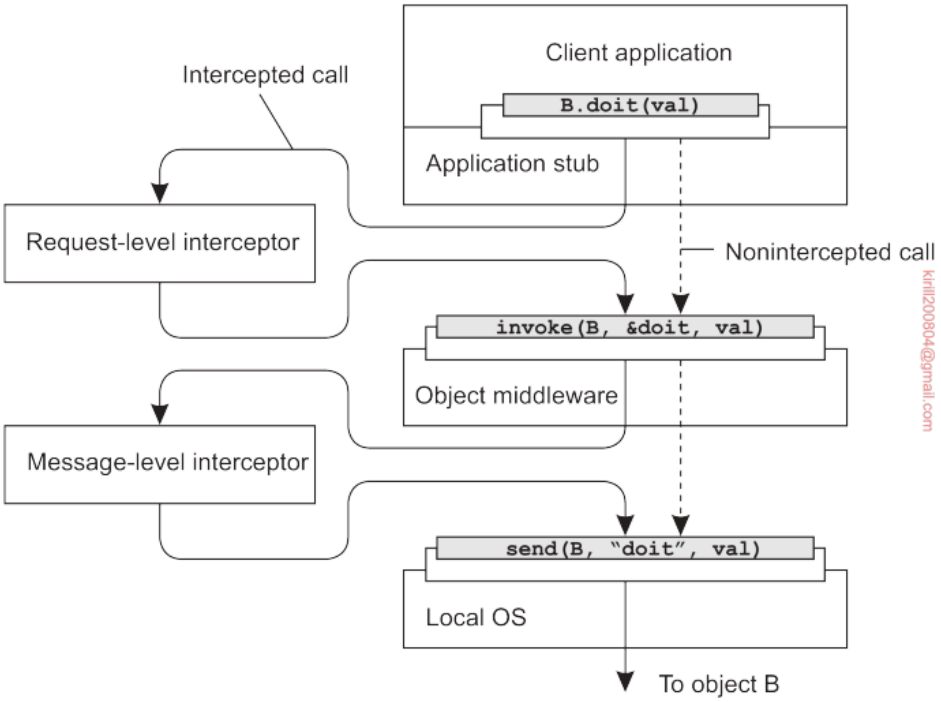


Схема применения перехватчиков, для реализации обработки вызова методов удаленного узла.

1. Клиентское приложение инициирует вызов метода (например, B.doit(-val)), предназначенного для объекта B.
2. Request-level interceptor перехватывает этот вызов и может, например, дублировать его, если объект B реплицирован.
3. Затем вызов трансформируется в универсальный объектный вызов (например, invoke(B, &doit, val)), который может быть обработан middleware.
4. Message-level interceptor может вмешаться на этапе формирования сетевого сообщения, например, для фрагментации больших данных перед отправкой через сетевой интерфейс локальной ОС.

Request-level interceptor обеспечивает уровень абстракции, где можно управлять запросами на более высоком уровне, например, для балансировки нагрузки или репликации.

Message-level interceptor работает на более низком уровне, близком к передаче данных, и может оптимизировать или модифицировать пакеты данных перед их отправкой по сети.

1.4. **Открытость.**

Открытость (openness) в распределённых системах означает создание системы с компонентами, которые легко интегрируются в другие системы или могут быть использованы ими. Эти системы часто состоят из компонентов, взятых из других систем. Открытость подразумевает соответствие компонентов стандартным правилам, описывающим их синтаксис и семантику предоставляемых услуг, обычно с использованием языка описания интерфейса (IDL). Однако сложность заключается в точном определении семантики интерфейсов, которая часто формулируется неформально.

Для достижения открытости часто используется:

- Применение перехватчиков, что позволяет различным реализациям систем или компонентов взаимодействовать, опираясь только на стандартизированные сервисы. Опыт показывает, что это позволяет приложениям быть исполнененым на другой системе с теми же интерфейсами (переносимость).
- Проектирование таким образом, что бы сисема могла легко настраиваться из разных компонентов и позволять добавлять новые или заменять существующие без воздействия на оставшиеся. Примером расширяемости могут служить плагины для веб-браузеров или систем управления содержимым вебсайтов.

  Для этого важно разделять политику (policy) от механизма, что и позволяет менять части без воздействия на всю систему. Такой подход противоположен монолитным системам, которые обычно закрыты и менее гибки.

В идеале, открытая система должна предоставлять механизмы, позволяя пользователям или разработчикам определять политику их использования. Это может быть реализовано через определение набора параметров или возможность интеграции пользовательских политик. Пример параметров для определения политики:

1. Хранение: Определяет, где кешировать данные, например, в памяти или на диске.
2. Исключение: Какие данные удалять из кеша, когда он заполняется.
3. Общий доступ: Будет ли кеш общим между пользователями или приватным для каждого.
4. Обновление: Когда система должна проверять актуальность данных в кеше.

1.5. **Расширяемость.**

Расширяемость (scalability) в распределённых системах описывает способность системы эффективно масштабироваться в трёх измерениях: размер, географическое распределение и административные границы. Размерная расширяемость позволяет системе поддерживать увеличение числа пользователей и ресурсов без потери производительности. Географическая расширяемость позволяет пользователям и ресурсам эффективно взаимодействовать на большие расстояния. Административная

расширяемость относится к способности системы оставаться управляемой, даже если она охватывает множество независимых административных организаций.

Техники масштабирования включают скрытие задержек коммуникации, распределение работы и репликацию. Скрытие задержек помогает в географической расширяемости, позволяя выполнять другие задачи, пока ожидается ответ от удалённого сервиса. Распределение работы может включать разделение компонентов или ресурсов на более мелкие части и их распределение по системе. Репликация увеличивает доступность и может помочь сбалансировать нагрузку, улучшая производительность и сокращая задержки коммуникации.

2. Архитектурные стили для распределенных систем

Архитектурный стиль распределенной системы определяется компонентами, способами их соединения, обмена данными и конфигурации в целостную систему. Ключевым элементом архитектуры является компонент - модульный блок с определенными интерфейсами, который может быть заменен без остановки всей системы. Это особенно важно, поскольку во многих случаях невозможно полностью остановить систему для обслуживания или обновления.

Другим важным элементом архитектуры является коннектор - механизм, который обеспечивает связь и взаимодействие между компонентами, например, через удаленные вызовы процедур, передачу сообщений или потоковые данные.

Существует несколько архитектурных стилей для распределенных систем, среди которых наиболее значимыми являются:

- Слоистые архитектуры, где система делится на логические слои с определенными функциями.
- Сервис-ориентированные архитектуры, фокусирующиеся на предоставлении сервисов как основных строительных блоков.
- Архитектуры на основе публикации-подписки, где компоненты взаимодействуют через обмен сообщениями.

В реальных распределенных системах часто применяется комбинация нескольких архитектурных стилей, особенно распространена практика использования слоистого подхода в сочетании с другими стилями. Это обеспечивает гибкость и масштабируемость системы, позволяя ей адаптироваться к различным требованиям и условиям эксплуатации.

Далее мы более подробно рассмотрим каждый из этих стилей, их особенности и применение в создании эффективных распределенных систем.

2.1. **Слоистая архитектура.**

Слоистая архитектура является одним из ключевых стилей в проектировании распределенных систем. Этот стиль характеризуется

организацией компонентов системы в виде иерархических слоев, где каждый слой предназначен для выполнения определенного набора функций и взаимодействует только со слоями, расположенными непосредственно выше или ниже.

2.1.1. *Основные принципы слоистой архитектуры:.*

1. **Вертикальная организация компонентов**: Компоненты распределены по слоям, причем каждый слой обслуживает определенный уровень абстракции или функциональности. Компонент в слое L_j может вызывать (делать downcall) компонент в более низком слое L_i (где $i < j$), ожидая от него ответа.
2. **Ограниченное взаимодействие между слоями**: Как правило, компоненты общаются только с соседними слоями. То есть компонент в определенном слое может взаимодействовать с компонентами в непосредственно прилегающих слоях - ниже или выше.
3. **Исключения в коммуникации**: В некоторых случаях допускаются так называемые upcalls - вызовы компонентами нижнего слоя к компонентам более высокого слоя. Это бывает полезно, например, когда операционная система должна уведомить приложение о каком-либо событии.

Слоистая архитектура облегчает понимание, тестирование и поддержку системы за счет четкой структуризации и разделения ответственности между слоями. Она также обеспечивает гибкость в изменении или замене отдельных компонентов системы без влияния на другие части.

2.1.2. *Пример приложения.*

Для лучшего понимания того, что из себя представляет приложение со слоистой архитектуре рассмотрим простейшее приложение на языке C под Linux. Каждый слой отвечает за определенный аспект работы приложения и взаимодействует только с соседними слоями. В качестве примера, давайте рассмотрим простое клиент-серверное приложение, где сервер обрабатывает запросы клиента.

2.1.2.1. *Структура Директории.*

```
server/
    ├── network_layer.c
    ├── network_layer.h
    ├── business_logic.c
    ├── business_logic.h
    └── main.c

client/
    ├── network_layer.c
    ├── network_layer.h
    ├── presentation_layer.c
    ├── presentation_layer.h
    └── main.c
```

2.1.2.2. *Реализация.*

2.1.2.2.1. ***Слой Сетевого Взаимодействия (Сетевой Слой):*** *Устанавливает и управляет сетевыми соединениями.*

- Отвечает за создание сокета (`socket()`).
- Привязывает сокет к адресу и порту (`bind()`).
- Слушает входящие соединения (`listen()` и `accept()`).

network_layer.h

```h
#ifndef NETWORK_LAYER_H
#define NETWORK_LAYER_H

#include <netinet/in.h>

int initialize_server_socket(int port);
int accept_connection(int server_fd, struct sockaddr_in *address);

#endif
```

network_layer.c

```c
#include "network_layer.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <unistd.h>

int initialize_server_socket(int port) {
    int server_fd;
    struct sockaddr_in address;

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))
< 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    return server_fd;
}

int accept_connection(int server_fd, struct sockaddr_in *address) {
```

```
    int addrlen = sizeof(*address);
     int new_socket = accept(server_fd, (struct sockaddr *)address,
(socklen_t*)&addrlen);
    if (new_socket < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    return new_socket;
}
```

2.1.2.2.2. ***Слой Бизнес-Логики (Логический Слой)****: Обрабатывает данные, полученные от клиента, и формирует ответ.*

Функция processData() является частью этого слоя. Обрабатывает данные, полученные от клиента. Генерирует ответ на основе бизнес-правил или логики.

business_logic.h

```
#ifndef BUSINESS_LOGIC_H
#define BUSINESS_LOGIC_H

void process_data(const char *input, char *output);

#endif
```

business_logic.c

```
#include "business_logic.h"
#include <ctype.h>

void process_data(const char *input, char *output) {
    int i = 0;
    while (input[i] != '\0') {
          output[i] = toupper(input[i]); // Преобразование в верхний
регистр
        i++;
    }
    output[i] = '\0';
}
```

2.1.2.2.3. ***Слой Данных (Data Layer):*** *Хотя в данном примере он неявно представлен, он может быть использован для управления данными, такими как доступ к базе данных или файловой системе..*

2.1.2.2.4. ***main.c*** *(Точка входа).*

```
#include "network_layer.h"
#include "business_logic.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>

```

```c
#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    char buffer[BUFFER_SIZE] = {0};
    char response[BUFFER_SIZE] = {0};

    int server_fd = initialize_server_socket(PORT);
    struct sockaddr_in address;
    int new_socket = accept_connection(server_fd, &address);

    read(new_socket, buffer, BUFFER_SIZE);
    printf("Received: %s\n", buffer);

    process_data(buffer, response);
    send(new_socket, response, strlen(response), 0);
    printf("Response sent: %s\n", response);

    close(new_socket);
    close(server_fd);

    return 0;
}
```

Этот код демонстрирует разделение на слои в приложении. В реальном проекте слои могут быть более сложными и включать дополнительную функциональность, а так же скорее всего Слоистая архитектура будет использована в месте с другими архитектурными стилями.

2.2. **Сервис-ориентированная архитектура (SOA).**

2.2.1. *Основные Принципы.*

Сервис-ориентированная архитектура (SOA) представляет собой архитектурный стиль, в котором приложения состоят из отдельных, независимых сервисов, каждый из которых инкапсулирует определенную функциональность. Эти сервисы могут быть реализованы в виде объектов, микросервисов или других независимых единиц, выполняемых как отдельные процессы или потоки.

2.2.2. *Ключевые Характеристики.*

1. **Инкапсуляция Сервисов:** Каждый сервис представляет собой самодостаточную сущность, которая может взаимодействовать с другими сервисами, но также может функционировать автономно.
2. **Объектно-ориентированный Подход:** В основе SOA лежит объектно-ориентированный дизайн, где каждый объект (или компонент) инкапсулирует состояние и методы, предоставляя четко определенный интерфейс для взаимодействия с другими объектами.
3. **Распределенные Объекты и Прокси:** В распределенных системах объект может быть физически расположен на одной машине, в то время как его интерфейс доступен на другой. При этом используются прокси и скелетоны (стабы) для маршалинга вызовов методов и обмена данными между клиентом и сервером.
4. **Микросервисы:** SOA может включать концепцию микросервисов, где каждый микросервис представляет собой отдельный процесс, выполняющий конкретную функцию. Микросервисы подчеркивают модульность и независимость.

2.2.3. *Преимущества и Недостатки.*

Сервис-ориентированная архитектура имеет достаточно спецефичные аспекты, в следствии этого предлагается ознакомиться с ее преимуществами и недостатками.

2.2.3.1. *Преимущества:.*

1. Модульность и Гибкость: Разделение функциональности на отдельные сервисы облегчает разработку, тестирование, развертывание и масштабирование.
2. Возможность Интеграции: SOA позволяет легко интегрировать и комбинировать сервисы разных производителей и технологий.
3. Улучшенное Управление Зависимостями: Снижается прямая зависимость между различными компонентами системы.

2.2.3.2. *Недостатки:.*

1. Сложность Управления: Управление множеством сервисов и их взаимодействиями может быть сложным.
2. Производительность: Взаимодействие между сервисами через сеть может влиять на производительность.

3. Сложность Интеграции: Хотя SOA облегчает интеграцию, она также может привести к сложным сценариям интеграции и управления данными.

2.3. **Архитектура на основе публикации-подписки.**

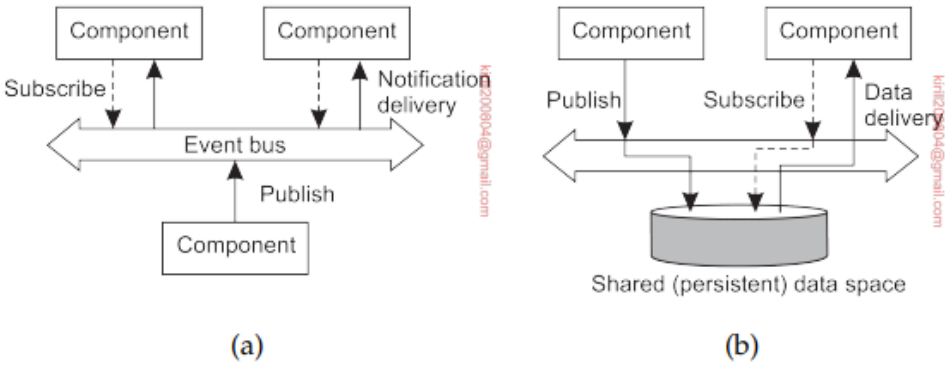
Архитектура на основе публикации-подписки представляет собой подход к построению распределенных систем, где взаимодействие между процессами или компонентами системы минимизирует прямые зависимости. Это достигается за счет использования модели, в которой процессы автономно функционируют, публикуя события (сообщения) и подписываясь на интересующие их события.

2.3.1. *Характеристики.*

1. **Разделение обработки и координации**: Процессы в системе работают автономно, а координация (коммуникация и сотрудничество) достигается через механизмы публикации и подписки.
2. **Временное и ссылочное разделение**: Процессы могут быть временно развязаны (не нуждаются в одновременной работе) и/или ссылочно развязаны (не знают друг о друге явно).
3. **Модели координации**:
    - **Прямая координация**: Процессы временно и ссылочно связаны, например, прямое общение через телефон.
    - **Координация через почтовый ящик**: Процессы временно развязаны, но ссылочно связаны. Общение происходит через обмен сообщениями, которые хранятся в почтовом ящике.
    - **Событийно-ориентированная координация**: Процессы временно связаны, но ссылочно развязаны. Процессы публикуют уведомления о событиях, на которые подписываются другие процессы.
4. **Общие пространства данных (Shared Data Space)**: Процессы общаются через обмен данными, структурированными в виде кортежей (tuples). Процессы могут извлекать или добавлять кортежи, соответствующие определенным критериям поиска.

2.3.2. *Примеры и Применение.*

- **Событийно-ориентированная система**: Процессы публикуют уведомления о событиях и подписываются на интересующие их типы уведомлений. Это может использоваться в системах обработки потоков данных, системах управления событиями и т.д.
- **Системы с общими пространствами данных**: Процессы обмениваются данными через общее пространство, где данные представлены в виде кортежей. Это может использоваться в системах, где необходимо эффективно обмениваться и обрабатывать большие объемы данных, например, в распределенных базах данных или системах управления потоками информации.



(a). Событийно-ориентированная система.
(b). Системы с общими пространствами данных.

3. Высокопроизводительные распределенные системы

Высокопроизводительные вычисления начались с введения мультипроцессорных систем, где несколько ЦПУ организованы таким образом, что все они имеют доступ к одной и той же физической памяти. Эта модель была удобной для улучшения производительности программ и относительно простой в программировании. Однако она обладает ограниченной масштабируемостью, особенно в контексте систем с множеством ядер.

Для преодоления ограничений систем с общей памятью высокопроизводительные вычисления перешли к использованию систем с распределенной памятью. Это требовало перехода от моделей с общей памятью к моделям передачи сообщений для коммуникации и синхронизации между потоками. К сожалению, модели передачи сообщений оказались более сложными и подвержены ошибкам по сравнению с моделями программирования с общей памятью.

Исследования в области распределенной общей памяти (DSM) были направлены на создание систем, позволяющих процессору обращаться к памяти на другом компьютере, как если бы она была локальной. Это достигалось за счет **техник, доступных операционной системе**, например, путем отображения всех страниц основной памяти различных процессоров в одно виртуальное адресное пространство. Однако попытки воссоздать системы с общей памятью с использованием мультикомпьютеров в конечном итоге были оставлены из-за несоответствия ожидаемой производительности, и программисты предпочли более сложные, но предсказуемо работающие модели передачи сообщений.

В современном мире высокопроизводительные распределенные системы могут быть разделены на две основные подгруппы:

1. Кластерные
2. Распределенные (Grid)

3.1. **Кластерные архитектуры.**

3.1.1. *Развитие и Популярность.*

Кластерные вычислительные системы стали популярными с улучшением соотношения цены и производительности персональных компьютеров и рабочих станций. Построение суперкомпьютеров с использованием доступных в массовом производстве технологий путем объединения простых компьютеров в высокоскоростную сеть стало финансово и технически привлекательным. Кластерные вычисления обычно используются для параллельного программирования, при котором одна ресурсоемкая программа выполняется параллельно на нескольких машинах.

Таким образом кластерные вычислительные системы представляют собой ключевую часть высокопроизводительных вычислительных сред, обеспечивая мощные ресурсы для обработки сложных вычислительных задач. Благодаря использованию стандартных компонентов и оптимизированных сетевых технологий, они обеспечивают гибкость, масштабируемость и эффективность, необходимые для современных научных и инженерных исследований. Важно отметить, что кластерные системы продолжают развиваться, адаптируясь к новым технологическим трендам и потребностям пользователей.

3.1.2. *Организация.*

В современных суперкомпьютерах, организованных в виде кластеров, используются конфигурации с более чем 100 000 ЦПУ, каждый из которых имеет 8 или 16 ядер. Существуют несколько сетей, наиболее важной из которых является сеть, образованная выделенными высокоскоростными взаимосвязями между различными узлами. Для управления и контроля организации и работы системы используется отдельная управляющая сеть, а также узлы.

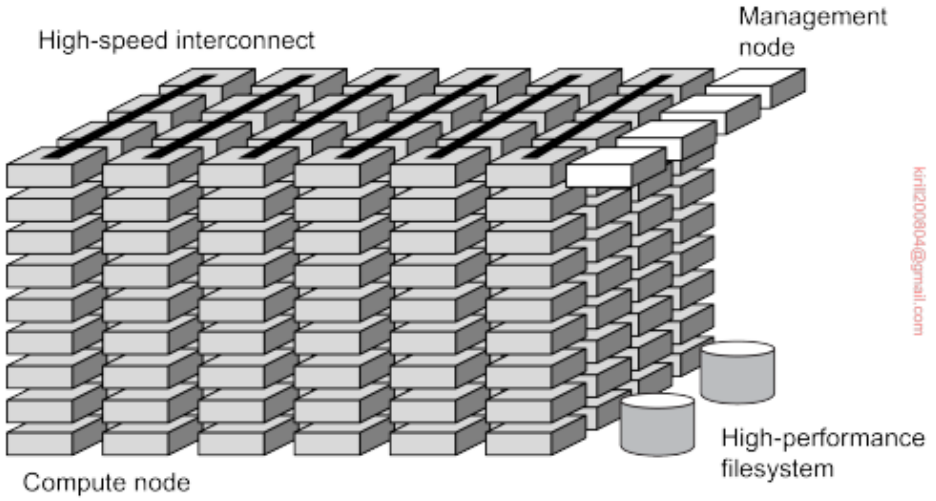
3.1.3. *Управление и Эксплуатация.*

Управляющий узел обычно отвечает за сбор заданий от пользователей и распределение связанных задач между различными вычислительными узлами. В практике при работе с очень крупными кластерами используются несколько управляющих узлов. Таким образом, управляющий узел выполняет программное обеспечение, необходимое для выполнения программ и управления кластером, в то время как вычислительные узлы оснащены стандартной операционной системой, расширенной типичными функциями для коммуникации, хранения, обеспечения отказоустойчивости и так далее.

3.1.4. *Роль Операционной Системы.*

Существует четкая тенденция к минимизации операционной системы до легковесных ядер, чтобы обеспечить минимально возможные накладные расходы. Однако недостатком таких операционных систем является то, что они становятся высокоспециализированными и точно настроенными на базовое оборудование. Эта специализация влияет на совместимость или открытость системы. В качестве компенсации наблюдается постепенный переход к так называемым мультиядерным подходам, при которых

полноценная операционная система работает рядом с легковесными ядрами (чаще всего в контейнерах), обеспечивая сочетание лучших качеств обеих систем. Эта комбинация становится необходимой, особенно в свете требований к одновременному выполнению множества независимых задач на высокопроизводительных вычислительных узлах. На сегодняшний день 95% всех высокопроизводительных компьютеров работают на системах, основанных на Linux; мультиядерные подходы разрабатываются для многоядерных ЦПУ, где большинство ядер процессора исполняют легковесные ядра Linux, и лишь малая часть - стандартную систему Linux. Это позволяет поддерживать новые разработки, такие как контейнеры. Влияние этих нововведений на вычислительную производительность невозможно преуменьшить.



Пример организации кластерной архитектуры.

3.2. **Grid - архитектуры.**

Grid-вычисления представляют собой уникальный подход к распределенным системам, позволяя объединять ресурсы из различных административных доменов для совместной работы и исследований. Благодаря своей гибкости и масштабируемости, они обеспечивают мощную инфраструктуру для сложных вычислительных задач, расширяя границы научного исследования и сотрудничества.

Они характеризуются высокой гетерогенностью, в отличие от традиционных кластерных систем, которые часто представляют собой однородную среду. В grid-системах нет предположений о сходстве оборудования, операционных систем, сетей, административных доменов, политик безопасности и т.д. Эти системы часто состоят из децентрализованных, федеративно объединенных компьютерных систем, каждая из которых может быть под управлением различных административных доменов и использовать различное оборудование и технологии.

3.2.1. *Организация.*

Одной из ключевых задач в grid-вычислениях является объединение ресурсов от разных организаций для совместной работы группы людей из различных учреждений, формируя так называемую виртуальную организацию. Процессы, принадлежащие к одной виртуальной организации, имеют доступные права к предоставленным ресурсам, которые могут включать вычислительные серверы (включая суперкомпьютеры, возможно, реализованные как кластерные компьютеры), хранилища данных, базы данных, а также специальные сетевые устройства, такие как телескопы, сенсоры и т.д.

3.2.2. *Слоистая Архитектура Grid-Вычислений.*

Архитектура grid-вычислений состоит из четырех уровней:

1. **Нижний Уровень (Fabric Layer)**: Обеспечивает интерфейсы к локальным ресурсам на конкретном сайте, предназначенные для совместного использования ресурсов в рамках виртуальной организации.
2. **Уровень Соединяемости (Connectivity Layer)**: Содержит коммуникационные протоколы для поддержки транзакций в grid, охватывающих использование множества ресурсов, включая протоколы передачи данных и доступа к ресурсам издалека, а также протоколы безопасности для аутентификации пользователей и ресурсов.
3. **Ресурсный Уровень (Resource Layer)**: Отвечает за управление отдельным ресурсом, использует функции, предоставляемые уровнем соединяемости, и напрямую обращается к интерфейсам, предоставляемым нижним уровнем. Этот уровень отвечает за контроль доступа к ресурсам, полагаясь на аутентификацию, выполняемую на уровне соединяемости.
4. **Коллективный Уровень (Collective Layer)**: Занимается управлением доступом к множественным ресурсам и обычно включает в себя услуги по обнаружению ресурсов, распределению и планированию задач на нескольких ресурсах, репликации данных и т.д. В отличие от уровней соединяемости и ресурсов, коллективный уровень может состоять из множества протоколов, отражающих широкий спектр услуг, которые он может предложить виртуальной организации.

3.2.3. *Приложения и Программное Обеспечение.*

На самом верхнем уровне находится прикладной уровень, который состоит из приложений, работающих в рамках виртуальной организации и использующих среду grid-вычислений. Коллективный, соединительный и ресурсный уровни в совокупности образуют ядро того, что можно назвать промежуточным программным обеспечением grid (grid middleware). Эти уровни совместно

обеспечивают доступ к ресурсам и управление ими, которые могут быть распределены по нескольким сайтам.

3.2.4. *Перспектива Промежуточного ПО и SOA.*

Важное наблюдение с точки зрения промежуточного программного обеспечения заключается в том, что в grid-вычислениях понятие сайта (или административной единицы) является общепринятым. Это подчеркивается постепенным переходом к сервис-ориентированной архитектуре (SOA), в которой сайты предоставляют доступ к различным уровням через коллекцию веб-сервисов. Это привело к определению альтернативной архитектуры, известной как Open Grid Services Architecture (OGSA), основанной на первоначальных идеях Фостера и др. (2001), но прошедшей процесс стандартизации. Реализации OGSA обычно следуют стандартам веб-сервисов.

4. Ссылки

1. Maarten van Steen, Andrew S. Tanenbaum - “DISTRIBUTED SYSTEMS” (4th edition, 2023).
2. George Coulouris, and others - “DISTRIBUTED SYSTEMS. Concepts and Design” (5th edition, 2012).
3. https://en.wikipedia.org/wiki/Distributed_computing
4. Kurgalin, Sergei; Borzunov, Sergei - “A Practical Approach to High-Performance Computing” (2019). SpringerLink. DOI: 10.1007/978-3-030-27558-7.
5. https://habr.com/ru/companies/alconost/articles/522662/
6. https://ru.wikipedia.org/wiki/Общий_ресурс