ЛЕКЦІЯ 3
ПРОБЛЕМИ ПОБУДОВИ РОЗПОДІЛЕНИХ СИСТЕМ
ВИМОГИ ДО РОЗПОДІЛЕНИХ СИСТЕМ

Дві події називають одночасними, якщо вони відбуваються протягом одного й того ж часового інтервалу.

Задачі паралельні, якщо виконуються паралельно – очевидно. Не обов'язково дві задачі повинні виконуватися точно в один й той самий час.

Наприклад, якщо взяти певну секунду, коли перша задача виконується в першу десяту частину секунди, а друга в третю – то такі задачі та процеси ми все ще називаємо паралельними, так як з точки зору людини секунда це достатньо малий проміжок часу. Але на практиці часто беруть ще більші відрізки часу.

Ціль технологій паралелізму – надати програмам можливість виконувати великий обсяг задач за відносно невеликий час, використати ресурси програми максимально ефективно.

Є два основні підходи для досягнення паралельності:

1) Методи паралельного програмування дозволяють розділити роботу програми між двома чи більше процесорами в рамках одного фізичного, або віртуального комп'ютера.
2) Методи розподільного програмування дозволяють розділити програму між процесами декількох комп'ютерів. Якщо ми застосовуємо методи паралельного програмування для програми, то в цій програмі ми можемо виділити процеси та потоки, про які ми поговоримо пізніше.

Найпростіша модель паралельного програмування – PRAM

Найпростіша модель паралельного програмування це модель PRAM (Parallel Random Access Machine – паралельна машина з довільним доступом) Це спрощена теоретична модель з n процесами, що використовують загальну глобальну пам'ять. Всі процеси мають доступ до читання та запису до

глобальної пам’яті. В цій середі можливий одночасний доступ. Припустимо, що всі процеси можуть одночасно виконувати арифметичні та логічні операції, окрім того, кожний з процесів може звертатися до пам’яті в одну неперервну одиницю часу. Модель PRAM має як паралельними так і виключними алгоритмами зчитування даних. Виключні алгоритми використовуються тоді, коли обов’язковою умовою є те, що дві програми (два процеси) не можуть одночасно записувати дані з однієї одиниці пам’яті, а паралельні для того, щоб два процеси могли одночасно зчитувати інформацію з однієї й тієї самої одиниці пам’яті.

Декомпозиція – процес розбиття задачі та її рішення на частини. Іноді частини задачі групуються в логічні блоки (пошук, сортування, обчислення, ввід та вивід даних), або ще можна групувати по логічним ресурсам (файл, база даних, принтер). Декомпозиція програмного рішення часто зводиться до декомпозиції робіт, декомпозиція робіт в свою чергу визначає що повинні робити різні частини ПЗ.

Одна з центральних проблем паралельного програмування – натуральна декомпозиція робіт для програмного рішення. Не існує простого та однозначного підходу до декомпозиції, тому що кожна задача і кожна програма потребує індивідуального підходу. Моделювання задачі за допомогою алгоритмів, принципів та паттернів може суттєво спростити задачу декомпозиції. Моделюючи задачу ми бачимо природній паралелізм (але можемо його і не побачити, тому що не всі задачі потребують паралельних обчислень, якщо паралелізм не потрібен – штучно нав’язувати його не треба).

Базові рівні програмного паралелізму:

- Паралелізм на рівні інструкцій. Такий паралелізм виникає тоді, коли декілька частин однієї інструкції можуть виконуватися одночасно. Такий паралелізм як правило підтримується директивами компіляторів та не контролюється програмістом.

- Паралелізм на рівні підпрограм. Структуру програми можна представити у форматі ряду функцій, ці функції розподіляються на потоки та можуть виконуватися на окремих процесорах.
- Паралелізм на рівні об’єктів. Кожен об1єкт можна призначити окремому потоку чи процесу.
- Паралелізм на рівні програм.

Проблеми координації. Якщо програма може бути розділена на підпрограми, що виконуються паралельно та в свою чергу можуть використовувати одні й ті самі дані одночасно, то неодмінно виникають проблеми координації.

Наведемо приклад проблеми координації.

Припустимо, у нас є програма підтримки електронного банку, яка дозволяє знімати гроші з рахунку та класти їх на депозит.

Припустимо, що ця програма розділена на 3 задачі A, B, C, що можуть виконуватися паралельно.

Задача A отримає запит від задачі B на виконання операції знаття грошей з рахунку. Задача A також отримає запити від задачі C покласти гроші на депозит. Задача A приймає всі запити та опрацьовує їх по по принципу fifo (first in, first out). Тепер уявимо, ща на рахунку є $1000, при цьому задача C вимагає покласти на депозит $100, а задача C хоче зняти з рахунку $1100.

Що відбудеться, якщо задачі B та C будуть намагатися оновити рахунок одночасно?

Очевидно, що значення на рахунку в певний момент часу може мати тільки одне значення.

Ми зіштовхуємося із проблемою координації.

Якщо запит задачі B буде виконаний на долю секунди раніше, ніж запит задачі C, то на рахунку буде від’ємне значення.

Для координації задач необхідно налагодити зв’язок між ними та забезпечити синхронізацію.

При некоректному зв’язку чи синхронізації як правило виникає 4 типи проблем.

Проблема 1. Гонка даних. Якщо декілька задач одночасно будуть намагатися змінити деяку спільну область даних, а кінцеве значення залежить від того, яка задача буде виконана першою, то таку ситуацію називаються race condition. Якщо декілька задач намагаються одночасно оновити деякий ресурс даних, то це називають data race. Не дивлячись на те, що нам би хотілося, щоб ми могли проводити скільки завгодно операцій по зняттю на додаванню грошей на рахунок – нам треба координувати задачі, коли ці дві операції проводяться над одним і тим самим рахунком одночасно. В такому випадку треба заздалегідь продумати правила координації. Наприклад в нашій програмі треба щоб операція ‘покласти гроші на рахунок’ була проведена завжди до операції зняття грошей. Або встановити правило. Що доступ до рахунку у момент часу може мати лише одна транзакція. Такі правила синхронізації дозволяють координувати процеси ефективніше.

Проблема 2. Нескінченна відсрочка. У плануванні, коли деякі процеси повинні очікувати того, що виконається якась задача чи виникнуть сприятливі умови можуть виникнути доволі непрості задачі для реалізації. По-перше очікувана подія чи умова повинні відбуватися регулярно. По-друге між задачами треба налагодити зв’язок. Якщо декілька задач очікують виконання іншої задачі для того, щоб виконатися, а ця задача не виконалась, виконується занадто довго, або виконалася частково, то наші задачі можуть так ніколи і не виконатися. Задачі завжди будуть знаходитися у стані очікування. Така ситуація називається нескінченною відсрочкою. Повертаючись до прикладу з електронним банком. Припустимо, ми, для того щоб позбавитися від проблеми data race ми встановили правило, за якого все процеси зняття грошей з рахунку очікують виконання всіх процесів поповнення рахунку. Тоді, задачі знаття

грошей можуть бути нескінченно відсроченими. Ми виходили з положення, що рахунок буде обов'язково поповнено і операція поповнення пройде без помилок. Але якщо наприклад не буде жодного запиту на поповнення, то знімати гроші також не вийде. І навпаки, якщо будуть постійно поступати запити на поповнення рахунку, ситуація буде та сама. Методи, що дозволяють запобігати нескінченні відсрочки ми розглянемо пізніше.

Проблема 3. Взаємне блокування. Для демонстрації цієї проблеми з затримками уявимо, що в нашій програмі підтримки банку два процеси працюють не з 1, а з 2 рахунками. Задача A отримає запити від задачі B на зняття грошей з рахунку, а від задачі C на перевід грошей на депозит. Задачі A,B,C можуть виконуватися паралельно, але задачі B та C можуть оновлювати одночасно тільки 1 рахунок. Задача A надає доступ задачам B та C до кожного рахунку по принципу fifo. Також припустимо, що задача B має монопольний доступ до 1 рахунку, а C до 2. При цьому обом задачам для виконання потрібен доступ до обох рахункуів. Задача B утримує рахунок 1, очікуючи доки задача C звільнить рахунок 2, а задача C очікує поки звільниться рахунок 1. Таким чином ці дві задачі попадають у тупікову ситуацію, що називають взаємним блокування.

Проблеми організації зв'язку. Багато розповсюджених паралельних середовищ часто складається з гетерогенних комп'ютерних мереж. Ми вже знаємо що таке гетерогенні комп'ютерні мережі, найбільша проблема у роботі з ними це те, що системи різних комп'ютерів можуть відрізнятися параметрами передачі даних. Це робить обробку помилок та виключень набагато складнішою. Наприклад іноді потрібно організувати сумісну роботу програм, що написані на різних мовах програмування. Навіть якщо розподільна або паралельна середа не є гетерогенною, проблема взаємодії між процесами та потоками залишаються нагальною. Кожен процес має свій власний адресний простів, тому для сумісного використання змінних, параметрів та значень, що повертаються функціями необхідно застосовувати технологію міжпроцесорної взаємодії.

В процесі розробки та планування програм, що використовують паралельні та розподільні технології виникають ці, та ще багато інших проблем, про їх рішення ми поговоримо в наступних лекціях.

Вимоги до розподільних та паралельних систем

Поговоримо про вимоги до розподільних та паралельних систем.

Першою вимогою є Прозорість (Transparency),

Transparency, тобто прозорість. Розподільна система не показує свою розподільну природу. Для користувача чи програміста множина комп'ютерів повинна й уявлятися єдиною комп'ютерною системою. Стандартний еталонної для моделі розподільної обробки у відкритих системах Reference Model for Open Distributed Processing визначають декілька типів прозорості.

Найбільш важливі з них:

- Прозорість доступу. Не залежно від способів доступу до ресурсів та їх внутрішнього представлення, звертання до локальних та віддалених ресурсів повинно бути однаковим. На базовому рівні скривається різниться архітектур та обчислювальних платформ, головне, щоб користувачеві ресурси різнорідних машин уявлялися єдиним образом. Найкращий приклад – API для роботи з файлами, що зберегаються на множині комп'ютерів різних архітектур, що представляє однакові виклики для локальних та віддалених файлів.
- Прозорість розташування. Дозволяє звертатися до ресурсів без знання про їх розміщення.
- Прозорість переміщення. Переміщення ресурсу, або процесу в інше місце, наприклад переміщення даних користувача, залишається непомітним для користувача для користувача розподільної системи.
- Прозорість реплікації. Якщо для підвищення ефективності, або рівню доступності створюється декілька копій ресурсу, цей факт залишається скритим від користувача. Всі копії ресурсу повинні,

очевидно, мати однакове ім’я та синхронізуватися не залежно від їх положення.

- Прозорість одночасного доступу. Дозволяє декільком користувачам користуватися одним і тим самим ресурсом одночасно, не знаючи цього.
- Прозорість відмов. Очевидно, що система повинна намагатися приховувати відмови та помилки у самій системі до їх виправлення та продовжувати свою роботу.

Степінь прозорості.

Важливо сказати, що в залежності від задач побудови розподільної системи степінь прозорості кожного з пунктів вище може варіюватися. Повністю скрити розподіл процесів та ресурсів дуже складно, особливо останній пункт – прозорість відмов та помилок повністю не можуть скрити навіть GitHub та Twitter, де ми іноді бачимо повідомлення про збої на сервері та відмови деяких функцій.

Друга – Відкритість (Opennes).

Відкрита система – це така, що реалізує відкриті специфікації та інтерфейси, служби та підтримуємі формати даних достатні для того, щоб забезпечити:

- Можливість переносу програмного забезпечення на широкий діапазон систем з мінімальними змінами.
- Взаємодію з іншими прикладними додатками на локальних та віддалених платформах.
- Взаємодію з користувачем, що робить перехід між платформами легше.

Відкрита специфікація – це така специфікація, що підтримується відкритим узгоджуваним процесом, спрямованим на постійну адаптацію до нових технологій та відповідає стандартам.

Згідно цьому визначенню відкрита специфікація не залежить від конкретної технології, тобто не залежить від конкретних технічних та програмних засобів.

В контексті розподільних систем вищенаведена визначення означає, що відкритість не може бути досягнена, якщо специфікація та опис ключових інтерфейсів програмних компонентів системи не доступні для програмістів.

Одним словом, ключові інтерфейси повинні бути описані та опубліковані.

Такий опис дозволяє процесам, яким потрібна деяка служба, звернутися до інших процесів, що реалізують цю службу, через відповідну службу. До того ж, такий підхід дозволяє створювати декілька реалізацій однієї і тієї ж служби, які з точки зору зовнішніх процесів будуть працювати однаково. Відповідно, декілька реалізацій однієї програмної системи можуть взаємодіяти та працювати сумісно, утворюючи єдину розподільну систему.

Таким чином, досягається інтероперабельність або, іншими словами, здатність до взаємодії (у попередніх лекціях ми розбирали проблеми розподільного програмування та говорили про те, що різні програмні рішення складно суміщати). Більш того, прикладний додаток, що розроблений для системи А, може без змін виконуватися у розподільний системі В, що реалізує ті ж самі інтерфейси, що А. Тобто досягається переносимість.

Третя вимога до розподільних систем – Масштабованість (Scalability).

Масштабованість – це здатність обчислювальної системи ефективно справлятися зі збільшенням користувачів та підтримуємих ресурсів без зменшення ефективності та збільшення навантаження на адміністраційну систему. При цьому систему називають масштабуємою, якщо вона здатна збільзувати ефективність своєї роботи при додаванні нових апаратних засобів.

Масштабованість – здатність системи рости з ростом навантаження на неї.

Для розподільних систем як правило виділяють декілька параметрів, що характеризують їх масштаб: кількість користувачів та кількість компонентів системи, степінь територіальної віддаленості мережевих комп'ютерів один від одного та кількість адміністаційних організацій, що обслуговують частини розподільної системи. Тому масштабованість розподільних систем також можна також визначають по наступним напрямам:

- Масштабованість навантаження. Здатність системи збільшувати свою ефективність при збільшенні навантаження шляхом заміни існуючих апаратних компонентів на більш потужні (вертикальна масштабованість) та додавання нових апаратних компонентів (горизонтальна масштабованість).
- Географічна масштабованість. Здатність системи зберігати свої основні характеристики, такі як ефективність, простота та комфортність використання, при територіальному рознесенні системи від локальної до більш віддаленої.
- Адміністраційна масштабованість. Характеризує простоту керування системою при збільшення кількості адміністраційно незалежних організацій, що обслуговують частини розподільної системи.

Складності масштабування.

Побудова розподільних систем включає в себе рішення великої кількості різноманітних задач та часто стикається з обмеженнями реалізованих у обчислювальних системах централізованих служб, даних та алгоритмів. А саме, багато служб централізовані у тому плані, що вони реалізовані у вигляді єдиного процесу та можуть виконуватися на одному сервері.

Проблема такого підходу в тому, що при збільшенні кількості користувачів або додатків, що використовують цю службу, сервер, на якому вона виконується стане слабким місцем системи, тому що його потужність обмежена. Навіть якщо потужність цього серверу можна збільшувати до

нескінченності (вертикальне масштабування), то обмежувальним фактором стане провідна здатність ліній зв’язку, що з’єднує сервер з іншими компонентами розподільної системи.

Аналогічно, централізація даних потребує централізованої обробки, що призводить до тих самих обмежень. В якості прикладу переваг децентралізованих систем можна навести систему доменних імен DNS. Розподільна база DNS підтримується ієрархією DNS серверів, що взаємодіють по певному протоколу.

Якщо всі дані DNS були б централізовані, розміщувалися б на єдиному сервері та при кожному запиті на інтерпретацію на доменного імені передавалися би на цей сервер, користуватися такою системою в масштабах всього світу було б неможливо.

Розподільні алгоритми мають наступні **характеристики**:

- Відсутність знання про глобальний стан. Як вже було сказано, централізовані алгоритми мають повну інформацію про стан системи та роблять висновки про наступні дії основуючись на стані системи у даний момент. В свою чергу, кожний процес, що реалізує частину розподільного алгоритму, має безпосередній доступ тільки до свого стану, але не до глобального стану всієї системи. Відповідно, процеси приймають рішення тільки на основі інформації про їх локальний стан. Інформація, що процеси отримали через повідомлення, на момент отримання повідомлення можу бути вже застарілою. Це нагадує ситуацію в астрофізиці. Дивлячись на зірки ми бачимо те, якими вони були мільйони років тому через обмеженість швидкості світла та розширення Всесвіту.
- Відсутність єдиного часу. Події, що виконуються у централізованому алгоритмі є повністю впорядковані. При виконанні розподільного алгоритму, процеси не мають спільного часу, тому їх не можна вважати повність впорядкованими.

- Відсутність детермінованості. Централізований алгоритм це строго детермінована послідовність дій. Виконання розподільного алгоритму має недетермінований характер через незалежне використання процесів з різною невизначеною швидкістю, а також через затримки зв’язку.
- Стійкість до відмов. Збій у будь-якому процесі або каналу зв’язку не повинні викликати помилок роботи розподільної системи. Для забезпечення географічної масштабованості потрібні свої методи та підходи.

Одна з основних причин поганої географічної масштабованості у багатьох розподільних системах є те, що в їх основі лежить принцип синхронного зв’язку. У цьому вигляді зв’язку клієнт, що викликає якусь службу блокується до отримання відповіді від неї. Це непогано працює, коли процеси відбуваються швидко і не залежно для користувача. Але при будь-яких затримках це може стати проблемою.

Інша проблема це те, що зв’язок в глобальних мережах по своїй природі нестабільний та майже завжди процеси взаємодіють point-to-point.

Технології масштабування. В більшості випадків складності масштабування проявляються у проблемах з ефективністю функціонування розподільних систем, викликаних обмеженістю потужності їх окремих компонентів: серверів та мережевих з’єднань.

Існує декілька основних технологій, що дозволяють зменшувати навантаження на кожен компонент розподільної системи. До них як правило відносять розповсюдження (distribution), реплікацію (replication) та кешування (caching). Distribution включає розбиття множини підтримуємих ресурсів на частини та рознесення цих частин по компонентам системи.

Replication та Caching також грають важливу роль, Replication не тільки підвищує доступність ресурсів у випадку часткової відмови, але й допомагає збалансувати навантаження між компонентами системи. Caching це форма

реплікації, коли копія ресурсу створюється безпосередньо близько до користувача, що використовує ресурс.