seminar 11

Вектор версий, CRDT, Operational Transformation

Одной из задач, возникающих в процессе работы распределенной системы, является разрешение возможных конфликтов.

Отношение happens-before и конкуретность.

Возьмем распределенную систему, работающую с помощью message-passing.

Определение. Отношение happens-before ($\rightarrow$) на множестве событий в распределенной системе это такое минимальное отношение, удовлетворяющее трем свойствам:

1. Если события a и b это два события, произошедшие в одном процессе, и событие a произошло до события b, то $a \rightarrow b$.
2. Если событие a это событие отправки сообщения от процесса A до процесса B, а событие b это прием этого сообщения, то $a \rightarrow b$.
3. Если $a \rightarrow b \wedge b \rightarrow c$, то $a \rightarrow c$.

При этом два события a и b называются *конкурентными* (англ. concurrent), если $a \nrightarrow b \wedge b \nrightarrow a$, обозначается как $a \| b$.

На рисунке пары событий (a, f), (b, f), (c, g), (c, h), (d, h) и (e, h) являются конкурентными.

Часы Лэмпорта

Самая цитируемая статья в computer science и фундаментальный результат, на который сейчас опирается добрая половина всех систем в разработке — работа Лэнсли Лэмпорта о времени и порядке в распределенных системах[1].

Часы Лэмпорта (англ. Lamport timestamp) являются одним из из примеров *логических часов* — механизма, позволяющего распределенной системе отслеживать временные и каузальные связи между событиями.

Представим систему, состоящую из процессов (узлов), каждый из которых проходит через набор событий. В ответ каждый узел может изменить свое состояние или инициировать отправку сообщения. Узел, желающий поддерживать часы Лэмпорта, заводит у себя глобальную переменную, назовем ее `time`. В момент наступления некоторого события, которое *в достаточной степени(*)* меняет состояние, узел увеличивает `time` и дальше обрабатывает событие:

```
time = time + 1

message = process_event(event, time)
if (message):
   send(message, time)
```

Узел, принявший сообщение, меняет свою метку времени в соответствии с новой информацией:

```
message, new_time = recv()
time = max(new_time, time) + 1

response = process_message(message, time)
if response:
    time += 1
    send(response, time)
```

Такая метка устанавливает порядок happens-before на пространстве событий, наступающих в распределенной системе, тем самым задавая eventually consistent модель взаимодействия (разные процессы могут по-разному думать о состоянии часов, но за конечное время все процессы "договорятся" о текущем значении).

Достоинством часов Лэмпорта является простота реализации. Недостатком является тот факт, что процесс не может понять, в какой метке логического времени находится другой процесс.

(*) определение уровня достаточности степени изменения состояния лежит целиком и полностью на архитекторе системы

Векторные часы

Векторные часы (англ. vector clock) — механизм, позволяющий процессам лучше понимать обновления состояний других процессов.

Пусть наша система состоит из N процессов. Каждый из них будет хранить вектор часов Лэмпорта $vc = (lc_1, lc_2, \ldots, lc_n)$, где vc — vector clock, lc_i — Lamport clock процесса i.

Зафиксируем контекст:

```
N = get_process_count()  # number of processes
i = get_process_id()   # local number of current process
vector_clock = (0 for _ in range(N))
```

При наступлении локального события, процесс увеличивает свою часть часов:

```
vector_clock[i] += 1
message = process_event(event, vector_clock)

if message:
    send(message, vector_clock)
```

При получении события сначала идет обновление локальной копии часов, затем происходит увеличение собственного времени, а затем обработка события:

```
message, recv_vector = recv()

for k in range(N):
    vector_clock = max(recv_vector[k], vector_clock[k])

vector_clock[i] += 1

response = process_message(message, vector_clock)
if response:
    vector_clock[i] += 1
    send(response, vector_clock)
```

На векторных часах определен частичный порядок: метка $\vec{s}$ считается меньше метки $\vec{t}$, если выполнено следующее:

$$s < t \iff [\forall i\ s_i \leq t_i] \wedge [\exists i\ s_i < t_i]$$

Тогда отношение happens-before принимает такой же вид: $s \to t \iff s < t$.

Вектор версий уже предлагает causally consistent модель взаимодействия — если событие a произошло до b для одного процесса, то a произошло до b *на всех процессах системы*.

Вектор версий

Вектор версий (англ. version vector) — механизм, позволяющий различным *репликам* одной системы (например, реплики базы данных) договариваться об изменениях, произошедших в системе.

Вектор версий использует векторные часы в качестве основного механизма, добавляя к нему еще один вариант взаимодействия: синхронизацию.

Когда две реплики решают синхронизироваться, они обе отсылают друг другу свою версию часов, после чего они обе обновляют свои локальные копии часов и состояния:

```
j = get_replica_to_sync()

send(vector_clock)   # other replica will first recieve, then send
other_clock = recv()

for i in range(N):
    vector_clock[i] = max(vector_clock[i], other_clock[i])
```

CRDT

Приведенные выше подходы позволяют в какой-то степени понимать, насколько реплики отстают друг от друга, дают понимание *порядка* событий в системе, но не дают *никакого* механизма разрешения конфликтов, если нам не нравится подход last writer wins. CRDT (англ. commutative replicated data type) решает эту проблему, позволяя очень просто реплицировать состояние.

Есть два типа CRDT — основанная на операциях (англ. operation-based) и основанная на состоянии (англ. state-based). Здесь мы разберем state-based CRDT.

State-based CRDT

Для создания sbCRDT нам нужно для нашего типа данных придумать две функции: *update* и *merge*. Пусть S_1 и S_2 — два разных состояния.

Функция merge должна быть

- *коммутативной*: $merge(S_1, S_2) = merge(S_2, S_1)$
- *ассоциативной*: $merge(S_1, merge(S_2, S_3)) = merge(merge(S_1, S_2), S_3)$
- *идемпотентной*: $merge(merge(S_1, S_2), S_2)) = merge(S_1, S_2)$

Функция update должна монотонно изменять состояние, то есть мы должны построить на значениях нашего типа данных частичный порядок. Обычно это делается с помощью version vector.

Для примера разберем создание G-counter (англ. grow-only conuter) с помощью CRDT. Семантика у этого счетчика следующая: любой процесс может увеличить значение счетчика, при этом в какой-то момент времени в будущем все процессы должны сойтись к одному состоянию счетчика.

Фиксируем контекст:

```
N = get_process_count()
i = get_process_id()

state = (0 for _ in range(N))
```

Функция update будет просто увеличивать значение счетчика на своей позиции:

```
def update(state):
    state[i] += 1
```

Функция merge будет брать поэлементный максимум значений из двух состояний:

```
def merge(left_state, right_state):
    result = list()
    for i in range(len(left_state)):
        result.append(max(left_state[i], right_state[i]))
    return result
```

Само значение счетчика получается из суммы значений счетчика на своих индексах:

```
def value(state):
    return sum(state)
```

Стоит отметить, что здесь не хватает версионирования записей с помощью vector clocks, но, в случае G-counter, такое версионирование будет излишним. В общем случае функция merge может использовать информацию о версии записи для своей правильной работы (здесь слово "правильный" следует понимать как "соответствующий семантике, заложенной разработчиком").

Такой тип данных очень просто реплицировать: нам достаточно семантики at most once на транспортном уровне для рассылки состояний по репликам, а они уже сами смержат разные состояния в одно состояние.

Operational transformation

Допустим нам надо решить задачу создания среды для совместного редактирования документов: пользователи видят не только свои изменения, но и изменения своих коллег. Здесь нам на помощь приходит техника OT.

Суть очень проста: пусть у нас есть некоторый тип данных T, над которым определен набор операций $OP_1, \ldots, OP_n$. Пусть конкурентно были совершены две операции: O_1 и O_2. Для того, чтобы применить операции к состоянию, реплика *преобразовывает* операции таким образом, что порядок применения этих операций не играет никакой роли.

Рассмотрим на примере текстового документа. Пусть S — последовательность символов. Определим операцию `put(pos, char)`, вставляющую символ `char` на позиции `pos`, и операцию `delete(pos)`, удаляющую символ на позиции `pos`.

Пусть два пользователя вставили по одному символу в разные места строки на позиции pos1 и pos2 соответственно и обозначим эти операции за ins1 и ins2. Реплика, ответственная за изменения состояния, может увидеть эти изменения либо в порядке (ins1, ins2), либо в порядке (ins2, ins1). Без потери общности зафиксируем порядок (ins1, ins2) применений изменений. После применения операции ins1 некоторые изменения строки потеряют свою корректность — это операции вставки и удаления с позицией большей, чем pos1. Для того, чтобы вернуть корректность операций, все последующие операции изменения строки *преобразуются*, чтобы учитывать уже произведенные изменения, поэтому операция `insert(pos2, char)` станет операцией `insert(pos2 > pos1 ? pos2 + 1 : pos2, char)`. То же самое произойдет со всеми последующими операциями.

Для того, чтобы не наращивать лог трансформаций, реплики используют вектор версий, где каждый элемент это порядковый номер операции от конкретного пользователя.

Источники

1. Lamport, Leslie. Time, clocks, and the ordering of events in a distributed system, Communications of the ACM, 1978 DOI PDF
2.