Distributed System 第二週作業

tags: DS paper notes

組別：第六組

組員: 梁耕銘(110062704), 周崑水(108005514),劉新正(109065851)

HackMD 連結
https://hackmd.io/sm5dGnlHSKmgEIsfEiEbvQ?view
(https://hackmd.io/sm5dGnlHSKmgEIsfEiEbvQ?view)

Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web

- cap theory
  - consistency
  - avalability
  - partition tolerance
- atomic and parition
  - lock
- atomic and availability
  - centralized
- available and partition tolerance
  - make any response (cache)
- partially synchronous vs asysnchronus
  - add clock
  - local timer

Q: what is stopping failure in paper

> fail to stop?
> — prof

Q: is there synchronous system

> it is impossible

Key words

Consistent, Avaialble and Partition-tolerant

These are 3 desired properties for designing a distributed web service

Distributed Web Service

Key Concepts

ACID when interacting with Web Service

> Interactions with web services are expected to behave in a transactional manner: operations commit or fail in their entirety (atomic), committed transactions are visible to all future transactions (consistent), uncommitted transactions are isolated from each other (isolated), and once a transaction is committed it is permanent (durable).

與 Web Service 互動時，ACID 的定義：

- automic: 只有全部完成交易或全部失效
- consistent: 所有排序中的交易都能看到已完成交易 (沒有暗盤)
- isolated: 尚未完成的交易彼此不相干
- durable: 交易一旦完成，那麼他就是持久的。

CAP Theorem

Brewer. 在 2000 年的 **PODC** (http://pods.org/podc2000) 提出，他猜想 web service 要同時達到 consistency,availability 以及 partition-tolerance 是不可能的（imposible)。

Consistency

- 參考 acid

availability

- 高可用性，也就是每個 web-service 請求 (request) 在網路可用的前提下，都可以收到成功的回應。

partition-tolerance

- 在高度分散的網路環境中， web service 需具備足夠的容錯能力。

fair executing

including those in which message are lost

Key Points

CAP 三者是 web service 需要的性質，針對 Brewer 提出的猜想，此篇 paper 作者則是先釐清 Brewer 的想法，將概念形式化後提出證明。並嘗試形式化的表達一些真實世界的解決方案。

Formal Model of CAP

Consistent service is as an atomic data object

- **任何一個緊接在寫入操作完成後開始的讀取操作，必得返回寫入操作完成後的結果或值。**

(https://docs.google.com/document/d/1CJaZkZb95t19BNMebhe442PCWTEWEGOkCNSB3Cdxw1c/edit#bookmark=id.i01rv0pss37j)

Available Data Objects

- Brewer 的原意，是幾乎 (almost available)，但此篇作者為簡化，將此定義為 100% 可用。
- 對一個分散式系統來說，一個持續有效的定義是：每一個接收到的請求，只要負責此請求操作的運算節點沒有失效，那麼都必須回應結果。也就是說，此服務所用到的任何一個演算法都會終止。
- 此定義看似並沒有考慮回應時間，似乎是個較弱的定義，然而若加上容錯的要求，那麼這個定義就變強，因為必須在即便有嚴重的網路失效的狀況下，仍然要終止每一個請求。

Partition Tolerance

一致性與高可用性都必須經過能容忍網路會出現異常分割的考驗。一旦網路因異常而分割成無法聯繫的獨立區塊，代表其間的要傳遞的訊息皆已丟失，在此況狀下，一致性要求的：即便在任意訊息丟失的狀況下對每一個請求的回應都必須是原子式的回應。高可用性的要求，依然是在任意訊息丟失的狀況下對每一個請求都必須做出回應。絕不能因為部分的網路失效，造成系統錯誤的回應。(若是全面的網路癱瘓則不在此限)

非同步網路 (Asynchronous Network)

不可能的結果

- 理論 1 在一個非同步網路模型中所實作出的具有讀寫功能的資料物件，在所有公平的任務執行之下，亦包括會有任

意訊息丟失，不可能仍保有可用性與原子式的一致性兩種性質。

> Theorem 1 It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:
> • Availability
> • Atomic consistency
> in all fair executions (including those in which messages are lost).

- 非同步網路模型：相對於同步網路模型具有同時性，非同步網路模型則無。
- 證明：略

證明的手法簡單來說，先假設 CAP 可以三者共存，若有兩個隔離網路 {G1,G2},各有一個節點，其中一個位於 G1 的節點接受寫入某個特定資料物件並以 v1 覆寫原有 v0 的指令，另一個節點接受讀取該資料物件寫入後的新值的指令，在隔離網路的通訊失效的狀況下，G1 節點的寫入資訊無法更新給 G2 的節點 (訊息丟失)，此時系統若選擇回應 (維持 Avability)，那麼從 G2 回應的值必是更新前的 v0。結果會不一致。若系統選擇一致性的結果，那麼就不能回應，因此與假設衝突。
引理 1.1

> Corollary 1.1 It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:
> • Availability, in all fair executions
> • Atomic consistency, in fair executions in which no messages are lost.

引理 1.1 與理論 1 的差異在於 Atomic consistency，在引理 1.1 中，即便沒有訊息的丟失，仍無法保證原子式的一致性。
此引理之所以可以成立，乃基於在非同步網路模型中，無法決定訊息是丟失還是隨機的延遲。因此進一步提出就算沒有訊息丟失，仍然無法保證原子式的一致性。

3.2 非同步網路模型的解決方案

在非同步網路模型中，CAP 兩倆成立是有可能的

3.2.1 Atomic, Partition Tolerant

若只考慮原子性質以及分割網路的容錯性質，而捨棄或不需要考慮可用性是否做得到呢？最簡單的作法，就是忽略所有的請求(不考慮可用性的狀況下，請求被接受了，但沒有任何回應。已讀不回)。
這麼說好了，只要有任何一丁點由於分割網路造成的失效，那麼就停止回應，以確保一致性。蠻多分散型的資料庫即採用此類型方案，因為對資料庫來說，維持一致性遠比可用性來得重要。

3.2.2 Atomic, Abailable

若不存在網路分割的狀況，則提供原子式、可用的資料是明顯可行的。事實上如 3.2.1 所描述的集中式演算即符合此要求。在內網(Intranet)以及區域網路(LAN)運行的系統為此類型演算的例子。

3.2.3 Availalbe, Partition Tolerant

不考慮一致性的要求，僅保證高可用以及分割網路的容錯，是可能的。因為最賴皮的做法，只要總是回應起始值(v0)就可以做到。若再積極一點，做到弱一致 (Weak Consistency) 即可有如 Web Cache 這一類型的應用。

部份同步網路 (Partially asynchronous network)

部分同步網路模型

真實世界很少有純粹的非同步模型，較多的是部分同步，相對於非同步模型不使用時鐘，部分同步網路模型每一個在地節點皆使用時鐘，只是這些時鐘彼此並沒有對時，他們的角色比較像是計時器，也就是可以用於量測經過多久時間。然而，在部分同步網路模型中，理論一所持主張仍然成立：

- 理論二 在一個部分同步網路模型中所實作出的具有讀寫功能的資料物件，在所有公平的任務執行之下，亦包括會有任意訊息丟失，不可能仍保有可用性與原子式的一致性兩種性質。

> Theorem 2 It is impossible in the partially synchronous network model to implement a read/write data object that guarantees the following properties:
> • Availability
> • Atomic consistency
> in all executions (even those in which messages are lost).

- 對比於引理 1.1，在部分同步網路模型中不會成立，原因是在引理 1.1 可以成立的條件乃取決於運算節點並不知道訊息遺失。然而在部分同步網路區域節點有計時器可以計算 time out, 若訊息的延遲超過設定的 time out，則一律當成訊息丟處理。
- 如此，有了可以判斷訊息是否已交付完成的機制，那麼就可以在確定訊息皆已交付完成的前提下，保證原子式的一致性。為弱一致性的保證。 (weak consistency) 因此有以下定義：

> Definition 3 A timed execution, α, of a read-write object is Delayed-t Consistent if:
>
> 1. P is a partial order that orders all write operations, and orders all read operations with respect to the write operations.
> 2. The value returned by every read operation is exactly the one written by the previous write operation in P (or the initial value, if there is no such previous write in P).
> 3. The order in
>   P is consistent with the order of read and write requests submitted at each node.
> 4. (Atomicity) If all messages in the execution are delivered, and an oper ation θ completes before an operation φ begins, then φ does not precede θ in the partial order P,
> 5. (Weakly Consistent) Assume there exists an interval of time longer than t in which no messages are lost. Further, assume an operation, θ, completes before the interval begins, and another operation, φ, begins after the interval ends. Then φ does not precede θ in the partial order P.

定義三的保證在訊息丟失時需允許可能出現的過時資料 (stale data)。與 Fekete 等學者提出的最終一致的想法相關。然而，我們還是想要知道如何得出一個明確的最終一致時間。

結論

在非同步有分割型態的網路， Web Service 要保證 CAP 三者是不可能的。
在真實的環境中，擇二則是可能的。

Perspectives on the CAP Theorem

Goal

- broader context
  - safety
  - liveness
  - unreliable
- pratical implications
  - consensus
  - consistency
  - synchrony
- trade-off
  - best availability
  - consistency
  - trading c for a
  - segmenting c and a

consensus

- what is consensus
  - multiple initial value, single value
- might disagreement
- requirements
  - agreement
  - validity
  - termination
  - integrity
- system may fail by carshing. but no partitions are possible

synchrony

- clock

- ideal
- eventual synchrony
- leader election service

consistency

- k-set

practical implications

- best effort avalability
  - keep consistency
  - chubby
- best effort consistency
  - keep avalability
  - web caching
- trading
  - setting the threshold
- segenebtubg c and a

CAP Twelve Years Later: How the “Rules” Have Changed

paper objections

- main argument
  - the “2 of 3” formulation was always misleading

cap confusion

- cannot forfeit P?
  - partition are rare occur
- invariants

cap -latency connection

- timeout
- partition decision

managing partitions

- detect
- limit some operations
- partition recovery
- maintain invariant

compensating for mistakes

- externalized mistake
  - ATM
  - airplane
- requires some log/history

Time, Clocks, and the Ordering of Events in a Distributed System

> — 梁耕銘

distributed system

> consists of a collection of **distinct processes** which are spatially separated, and whch **communicate** with one another by **exchanging message**

The Partial Ordering

> “happened before”

- relation “\(\rightarrow\)”
  1. if *a* and *b* are events in the same process, and *a* comes before *b*, then \(a \rightarrow b\)
  2. if *a* is the sending of a message by one process and *b* is tje receipt of the same message by another process, then *a* \(\rightarrow\) *b*
  3. if \(a \rightarrow b\) and \(b \rightarrow c\) then \(a \rightarrow c\)
  - two distinct events *a* and*b* are said to be concurrent if \(a \nrightarrow b\) and \(b \nrightarrow a\)
- space-time diagram

- \(p_1 \rightarrow r_4\)

- we say \(p_3\) and \(q_3\) are concurrent

Logical Clocks

- Clock Condition

  > if \(a \rightarrow b\) then \(C \langle a \rangle < C \langle b \rangle\)
  > we cannot expect the converse condition to hold as well

  - C1. If *a* and *b* are events in process \(P_i\), and a comes before *b*, then \(C_i \langle a\rangle < C_i \langle b \rangle\)
  - C2. If *a* is the sending of a message by process \(P_i\) and *b* is the receipt of that message by process \(P_j\), then \(C_i \langle a \rangle < C_j \langle b \rangle\)
- rules
  - IR1. Each process \(P_i\) increments \(C_i\) between any two successive events
  - IR2.
    1. If event *a* is the sending of a message *m* by process \(P_i\), then the message *m* contains a timestamp \(T_m = C_i \langle a \rangle\)
    2. Upon receiving a message m, process \(P_j\) sets \(C_j\) greater than or wqual to its present value and greater than \(T_m\)

Ordering the Events Totally

> we can use a system of clocks satisfying the Clock Condition

- relation \(\Rightarrow\)
  - if \(a\) is an event in process \(P_i\) and \(b\) is an event in process \(P_j\), then \(a \Rightarrow b\) if and only if either (i) \(C_i \langle a \rangle < C_j \langle b \rangle\) or (ii) \(C_i \langle a \rangle = C_j \langle b \rangle\) and \(P_i \prec P_j\)
  - \(\Rightarrow\) is a way of completing the “happened befor” aprtial ordering to a total

ordering
    - \(\Rightarrow\) is depends upon the system of clocks \(C_i\), and is not unique

> Consider a system composed of a fixed collection of processes which share a single resource

- request queue
    - a distributed algorithm
    - each process independently follows these rules, and there is no central synchronizing process or central storage
    - *State Machine*
    - a process must know all the commands issued by other processes
    - the resulting algorithm requires the active participation of **all the processes**

Anomalous Behavior

> issues a request A, and then telephones a friend to issue a request B

Physical Clocks

> let \(C_i(t)\) denote the reading of the clock \(C_i\) at physical time t
> clocks run continuously rather tha in discrete “ticks”
> assume that \(C_i(t)\) is a continuous, differentiable function of t except for isolated jump discontinuities where the clock is reset
> \(dC_i(t)/dt\) represents the rate at which the clock is running at time t

- PC1. \(\kappa \ll 1\), for all i: \(|dC_i(t)/dt - 1| < \kappa\)
- PC2. for all i, j: \(C_i(t) - C_j(t)| < \epsilon\)

> since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart

- devise an algorithm

> assume that when a clock is reset, it is always set foward and never back

- theorem

Cluster-Based Scalable Network Services

3 fundamental challenges

1. scalability
    - the load offered to the service increases, an incremental and linear increase in hardware can maintain the same per user level of service
2. availability
    - the service as a whole must available 24x7, despite transient partial hardware or software failures
3. cost effectiveness
    - the service must be economical to administer and expand, even though it potentially compreises many workstation nodes

Advantages of Clusters

- Scalability
- High Availability
- Commodity Building Blocks
- cost/performance

Challengers of Cluster Computing

- Administration
- Component vs. system replication
- Partial failures
- Shared state

BASE Semantics

- stale data can be temporarily tolerated

- soft state
- approximate answers
- may be more valuable than exact answers delivered slowly

Outline

Abstract

1. Identify 3 fundemental requirements for scalable network service:
    - incremental scalability and overflow growth provisioning
    - 24x7 availability through fault masking
    - cost effectiveness
2. Proposed a general, layered architecture for building cluster-based scalable network services that encapsulates the abve requirements for reuse, and a service-programming model based on TACC of internet content.
3. Two real implementation of services based on this architecture are discussed, the one is the TranSend, a Web distillation proxy deployed to the UC Berkeley dialup IP population and the other is the HotBot, a commercial implementation of the Inktomi search engine.

Key Concept

TACC

A Service Programming model based on composable workers that perform transformation, aggregation, caching and customization

BASE

A weaker-than-ACID data semantics that results from trading consistency for availability and relying on soft state for robustness in failure management

Scalibility, Availability and Cost effectiveness

3 fundemental challenges to the deployment of the network service
- By scalibility, 服務的負載升高時，可以線性的進行硬體擴容並且維持相同的服務水準。
- By availability, 即便遭遇短暫軟硬體失效的狀況下，仍可維持 7x24 的可用性
- By cost effectiveness, 即使在工作節點會不斷增加的狀況下，仍能維持經濟的管理以及擴充成本。

2.3 TACC : A PROGRAMMING MODEL FOR INTERNET SERVICE

TACC stands for transformation, Aggregation, Caching and customization

- Transformation
  an operation on a single data object that changes its content, such as filtering, transcoding, re-rendering, encryption and compression.
- Aggregation
  collect and collate data
- Caching
  資料在網路移動的成本比重新計算或者暫存成本都來得高。使用暫存可以有效降低網路負載。
- Customization
  個人化、客製化

TRANSEND

a scalable transformation and caching proxy for the 25,000 Berkley dialup IP users (connecting through a bank of 600 modems)

HOTBOT

It is the commercialized search engine from Inktomi, which performs millions of queries per day against a database of over 50 million web pages.

Motivation

觀察到工作站叢集(clusters of workstations)具有一些加以發揮就可以達成擴展性、可用性以及成本效益的良好性質。比如使用商用PC 就可以讓服務的提升與成本上升的陡峭曲線趨緩，叢集的 redundany 可以屏蔽掉移轉失效(transient

failures) 以及 **Embarrassingly parallel** (https://en.wikipedia.org/wiki/Embarrassingly_parallel) 類型的網路服務工作負載 (`network service workloads`)可以很好的對應到工作站網路。然而，要發展一個叢集軟體以及可以管理運行中的叢集仍是一複雜的事。

Solved Problem & Contribution

設計、分析以及導入了一個 layered framework 表明了上述所提的複雜之處，以及讓新的服務可以採用此架構即可解決 scalabilty, availability,等等問題，讓開發者可以專注在服務的內容上。

Solutions

Layered framework

LOWER LAYER:

handles scalability, avail- ability, load balancing, support for bursty offered load, and system monitoring and visualization

MIDDLE LAYER:

provides extensible support for caching, transformation among MIME types, aggregation of information from multiple sources, and personalization of the service for each of a large number of users

TOP LAYER:

allows composition of transformation and aggregation into a specific service, such as accelerated Web browsing or a search engine

Conclusion

1. Proposed a layered architecture for cluster-based scalable network services
2. Identify the class of cluster-based scalable network services can substantially increase the value of the internet access to end users while reamining cost-efficient to deploy and adminstrater

Research Value & Application

Challenge of cluster computing

paper 作者 提出相對於 SMP 架構的四點挑戰

Administraion

作者在前篇論文提出透過視覺化工具可以有效支援管理工作

Component vs. system replicaion

單一台商用 pc 的運算能力或許無法支撐整個服務的運算負載，但透過適當的功能限縮，以及可以可交換的工作類型，例如要 cached data 或對資料進行壓縮等工作。
重製系統以及工作的移轉相對是容易的。

PARTIAL FAILURE

必須要能解決系統部分失效的問題

SHARE STATE

如何避免跨叢集 share state 的需要或是降到最少
作者特別針對 partial failure 以及 share state 的問題提出以下解法：

BASE Semantic

相對於交易類型的 ACID，作者提出 BASE 分別代表 B: Basically Available, Soft sate, Eventual consistency

- Basically Availability
- Soft sate for fault tolerance and availability
    - SNS Component 借鏡了在廣域 tcp/ip 網路獲得巨大成功的方法之一，其乃依賴定期從同儕節點回報並且暫存的軟體狀態 (soft state) (?)
    - 另一方法則使用 time out 作為容錯機制。

Institution of the approach

Main Result, evaluated and mean

Improvement, limitation and weakness (Author’s future work)

- adaptation via distillation (?) 作者在過去提出這個方法認為可以用在動態的調整以符合使用者網路連結的行為，未來想要結合 www proxy prototype with the event notification mechanism 以提供一個 adaptive solution 給無線網路進來的 web access
- 尚未研究所提出的架構在網際網路的服務器架構是否依然可行。
- TACC 模型仍未成熟，期待可以開發成 SDK

Weighted Voting for Replicated Data

DS paper notes v1.0 (109065851)-20220303

tags: CS54260 Distributed System

Abstract

Every transaction collects a read quorum of r votes
to read a file, and a write quorum of w votes to write a file,
such that r+ w is greater than the total number of votes assigned to the file. This ensures that there is a non-null intersection between every read quorum and every write quorum. Version numbers make it possible to determine which copies are current. The reliability and performance characteristics of a replicated file can be controlled by appropriately choosing r, w, and the file's voting configuration. The algorithm guarantees serial consistency, admits temporary copies in a natural way by the introduction of copies with no votes, and has been implemented in the context of an application system called
Violet.
Quorum

- [x] Vr + Vw > V
- [x] Vw > V/2

Characteristics

- Every copy of a replicated file is assigned some number of votes.

- Every transaction collects a read quorum of r votes to read a file, and a write quorum of w votes to write a file, such that r+ w is greater than the total number of votes assigned to the file.
- This ensures that there is a non-null intersection between every read quorum and every write quorum. There is always a subset of the representatives of a file whose votes total to w that are current.
- Thus, any read quorum that is gathered is guaranteed to have a current copy.
- Version numbers make it possible to determine which copies are current.

Properties

- It continues to operate correctly with inaccessible copies .
- It consists of a small amount of extra machinery that runs on top of a transactional file system . Although “voting” occurs as will become evident later in the paper, no complicated message based coordination mechanisms are needed.
- It provides serial consistency . In other words, it appears to each transaction that it alone is running. The most current version of data is always provided to a user.
- By manipulating r, w, and the voting structure of a replicated file , a system administrator can alter the file’s performance and reliability characteristics.
- All of the extra copies of a file that are created, including temporary copies on users’ local disks, can be incorporated into our framework .
-

Objects

Environment & Assumption

- [x] Stable file system
- [x] File or Container object with unique names.

- [x] File system primitive~ operate on remote and local files alike.
- [x] File Suite : This is afile that is realized by a collection of copies, which we call representatives because of the democratic way in whichupdate decisions are reached. A file suite can also be made very reliable by having many equally weighted representatives 2 Classes
  - [ ] decentralized structure : equally weighting representatives
  - [ ] completely centralized scheme: assigning of all of the votes to one representative
- [x] Representatives : File copies, replication

Problem Solving

- [x] guarantee serial consistency for queries
- [x] weighted voters maintain flexibility
- [x] not insist that a majority of an object's copies be updated
- [x] algorithm that is easier to reason consistency and replication, also easy to implement.
- [x] allows for the inclusion of temporary copies.

Voting configuration created then ajust r, w to get reliability and performance,

- [x] read write ratio
- [x] the cost of reading and writing
- [x] desired reliability and perfomance

Algorithm

```
FileSuite: MONITOR [suiteName: File.ID] = BEGIN
    VersionNumber: TYPE = {unknown, 1, 2, 3, 4, ... }
    Set: TYPE = ARRAY OF BOOLEAN;
    SuiteEntry: TYPE = RECORD [
    name: File.ID,
    version: VersionNumber,
    votes: INTEGER];
    suite: ARRAY OF SuiteEntry;
    currentVersionNumbcr: VersionNumber;
    firstResponded: BOOLEAN; -- true whenfirst representative has
    responded
    r: INTEGER; .. number of votes required for a read quorum
    w: INTEGER; -- number of votes required for a write quorum
```

```
Read: PROCEDURE {file: File.ID, firstByte, count: INTEGER, buffer:
POINTER] =
    BEGIN
    -- select best representative
    quorum: Set ~- CollectReadQuorumfl;
    best: INTEGER ~-
    SelectFastestCurrentRepresentative[quorum];
    -- send request and wait for response
    File.Read[suite[best].name, firstByte, count, buffer];
    END;
 Write: PROCEDURE [file: File.ID, firstByte, count: INTEGER, buffer:
    POINTER] =
    BEGIN
        -- select write q~orura
        quorum: Set ,- CollectWriteQuorum[i;
        i, count: INTEGER ~- 0;
        process: ARRAY OF PROCESS;
        -- send requests to all members of quorum, and wait for responses
        FOR i IN [1..LENGTH[SUite]]
            DO
                IF quorum[i] THEN
                BEGIN
                count ,- count + 1;
                process[count] 4- FORK
                RepresentativeWrite[i, firstByte, count, buffer];
                END;
            ENDLOOP;
        FOR i IN [1..count]
            DO
            JOIN process[i];
            ENDLOOP;
    END"



RepresentativeWrite: PROCEDURE [i, firstl~yte, count: INTEGER, Buffer: POINTER
    BEGIN
        -- we are actin  on be alf of our parent;  oin its transaction
        Transaction.JoinParentsTransaction[];
        Update ersionNumber[i];
        -- write data on representative and inform parent process
        File.Write[suite[i].name, firstByte, count, buffer];
    END;

CollectReadQuorum: ENTRY PROCEDURE RETURNS [quorum: Set] =
    BEGIN
        i,  , votes: INTEGER:
        inde : ARRAY OF INTEGER;
        -- until t e first representative responds we don t  ave a seed for t
        UNTIL firstResponded DO WAIT CrowdLar er ENDLOOP;
        -- an endless loop t at only returns w en a quorum  as been establis e
        DO
        -- if we  ave a read quorum  ere. t en t e votin  rules are current
        inde  ~- SortRepresentativesBySpeedl];
        quorum 4- ALL[FALSE]; votes ,- 0;
        -- see  f we can find a read quorum
        FOR i IN [I..LENGTH[SUite]]
            DO
              ~- inde [i];
            IF suite[ ].versionNumber ~un nown THEN
            BEGIN
                quorum[ ] ~- TRUE;
                votes ~- votes + suite[ ].votes;
                IF votes   = r THEN RETURN[qUorum];
            END;
        ENDLOOP;
        -- we can t find a quorum
        WAIT CrowdLar er;
        ENDLOOP;
    END;
```

Refinement

1. Weak Representatives
    - representatives with no votes,called weak representatives. it bears no responsibility for the long term safekeeping of data

2. Lock Compatibility
    - locks are set by the stable file system to guarantee serial consistency, which reduces the amount of concurrency in the system.

3. Lower Degrees of Consistency
    - setting r to be 0. “give me the latest version you can find, but Idon’t care if it isn’t fresh”

4. Size of Replicated Objects

- The size of an object that is replicated should be chosen to match the needs of an intended application, but it depends on a file's version number remaining unchanged throughout a transaction, the smallest unit that can be individually locked is a file

5. Broken Read Locks
   - if tx abort, then result in, first, fewer transactions are aborted. Second, it is not necessary for a version number to remain unchanged during a transaction. The smallest lockable unit is no longer a file, but instead is the smallest lockable unit supported by the stable file system.

6. Reassigning Votes
   - Imagine that a transaction incorrectly assumes that an obsolete generation of voting rules, G, is current. Then there is a set voting rules, G + 1, that is one generation more recent. But when G+ 1 was established its rules were written into a write quorum (under G). Therefore, the transaction would have examined a representative that contained or once contained G+I. If this representative did not contain G + 1, then it contained a later generation of the voting rules, it is contradiction

7. Replicating Containers
   - a system operator does not know what a failure of container implies. Replicated containers provide a solution to this problem. a base container introduced

8. Releasing Read Locks
   - An enhancement to the algorithm would be to release the read locks on representatives that are not used as part of a quorum before committing. This would reduce the amount of communication at commit time significantly.

9. Updating Representatives in Background

- In conjunction with replicated containers, it is possible to operate servers that update obsolete representatives by examining a replicated container and initiating appropriate transfers.

Violet System

FileSuite

Structure

Conclusion

The separation of consistency considerations from replication has resulted in a conceptually simple approach which guarantees serial consistency in a straightforward way and is relatively easy to implement. When a decision has to be made by cooperating nodes with different probabilities of being correct, weighting the nodes' responses will improve the probability that a correct decision is reached.