

Paxosクラウド

～過半数(セル化技術)で故障管理を克服

～30年来の夢の実現

～ディスクよ、さようなら

- ・概要
- ・信頼性
- ・Paxosモジュール
- ・Sessionモジュール
- ・ファイルシステム(PFS)

2010年11月
2011年4月 (Version 1)
トライテック
渡辺典孝

I部 概要

- 故障は回避できない
 - 信頼性を上げるには膨大なコスト
 - ⇒故障を前提にソフトで解決(30年来の夢)

逆転の発想

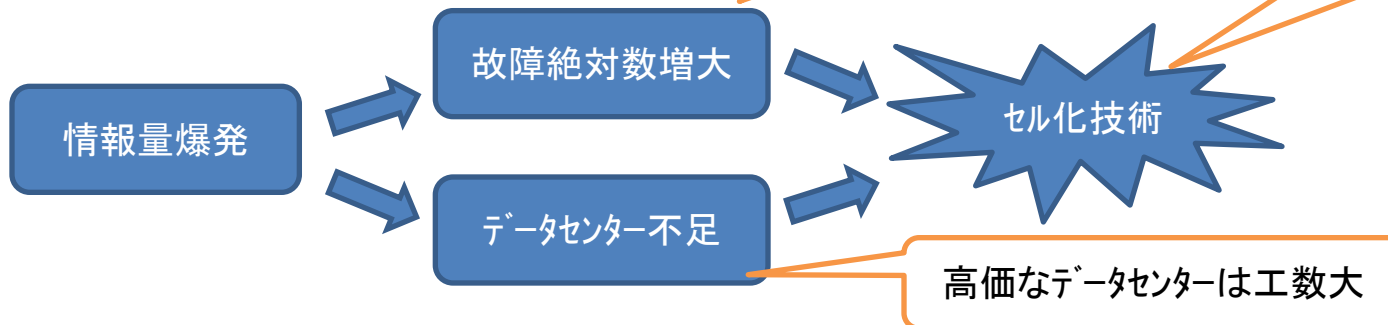
- 協働自律システム(セル)
 - 協働で信頼性を上げる(べき乗)
 - 市販製品でコスト低下

- キャッシュが前提
 - ディスクは必要ない

永続化安心の放棄
ただし、無限のデータは無限の通信時間

無視できない

丸ごとべき乗で改善
低コスト



クラウド(コンピュータ)の三分野

- 情報系—情報爆発

- ペタバイト 量が質に転化
- 参照が殆ど
- 分割統治が可能⇒分散できる

cloudの特徴

scalability

availability(←reliability)

virtualization

- 業務系

- テラバイト以下
- 仮想化で燃えているが？

- 科学技術計算

- エキサバイト？
- 分割統治ができない⇒CPU(パイプライン、キャッシュレジスタ)を速くするしかない
 - 逆行列は逐次的に解く
 - ロングテール(影響の連鎖)
- 並列は邪道？(グリッド)
 - モンテカルロ法？

分散量がハンパではない
⇒信頼できる構成ファイルの管理
⇒信頼できるネットワーク同期
⇒信頼できる死活監視

ハウジングは閑古鳥、
⇒データセンターへ？

生き残る



故障は回避できない

- **災害、通信、ソフトウェア、ソフトバグ等の故障は回避不能**
 - 1サーバーの故障で多数のクライアントが影響を受ける
 - キャッシュ情報は回復できない
 - サイト建設に多額の費用
 - 耐震建屋、2重化設備、UPS、自家発電、冷却、特機マシン
 - 宇宙・**原発**のような過酷な環境
- **協働自律システム(セル)による信頼性の劇的改善**
 - セル故障率は $O(r^{**}[N/2])$ となる
 - キャッシュ故障率は $O(r^{**}N)$ となる
 - コモディティ製品、通常建設
 - 計画的保守交換が可能
 - 広域分散で災害に対応できる

故障率と費用

~協働自律化の費用は線型

故障率と費用には図のような曲線が想定される。

例えば、

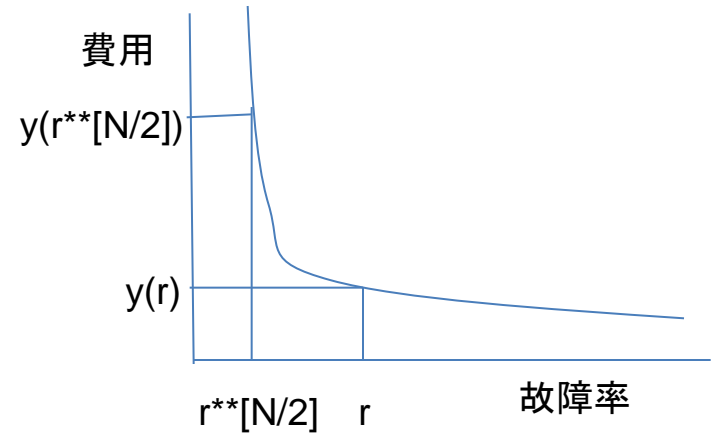
$$y \propto 1/x$$

を仮定する。

故障率 $r^{**}[N/2]$ を実現する場合、

セルでは、 $N*y(r) \propto N*1/r$ の費用

1台では、 $y(r^{**}[N/2]) \propto 1/r^{**}[N/2] = 1/r^{**}([N/2]-1)*y(r)$ の費用
となり、 $N:1/r^{**}([N/2]-1)$ となる。



セルの故障率(近似値)

故障率	3台	5	7
5%	0.75%	0.125%	0.021875%
3%	0.27%	0.027%	0.002835%
1%	0.03%	0.001%	0.000035%

費用比(1台/セル)

故障率	3	5	7
5%	6倍	80	1142
3%	11.1	222.2	5291
1%	33.3	2000	142857

故障率3%の5台構成セルで0.027%を実現。
費用は、1台20万円として100万円だが、
1台で実現すると222倍の2億2千万円となる！

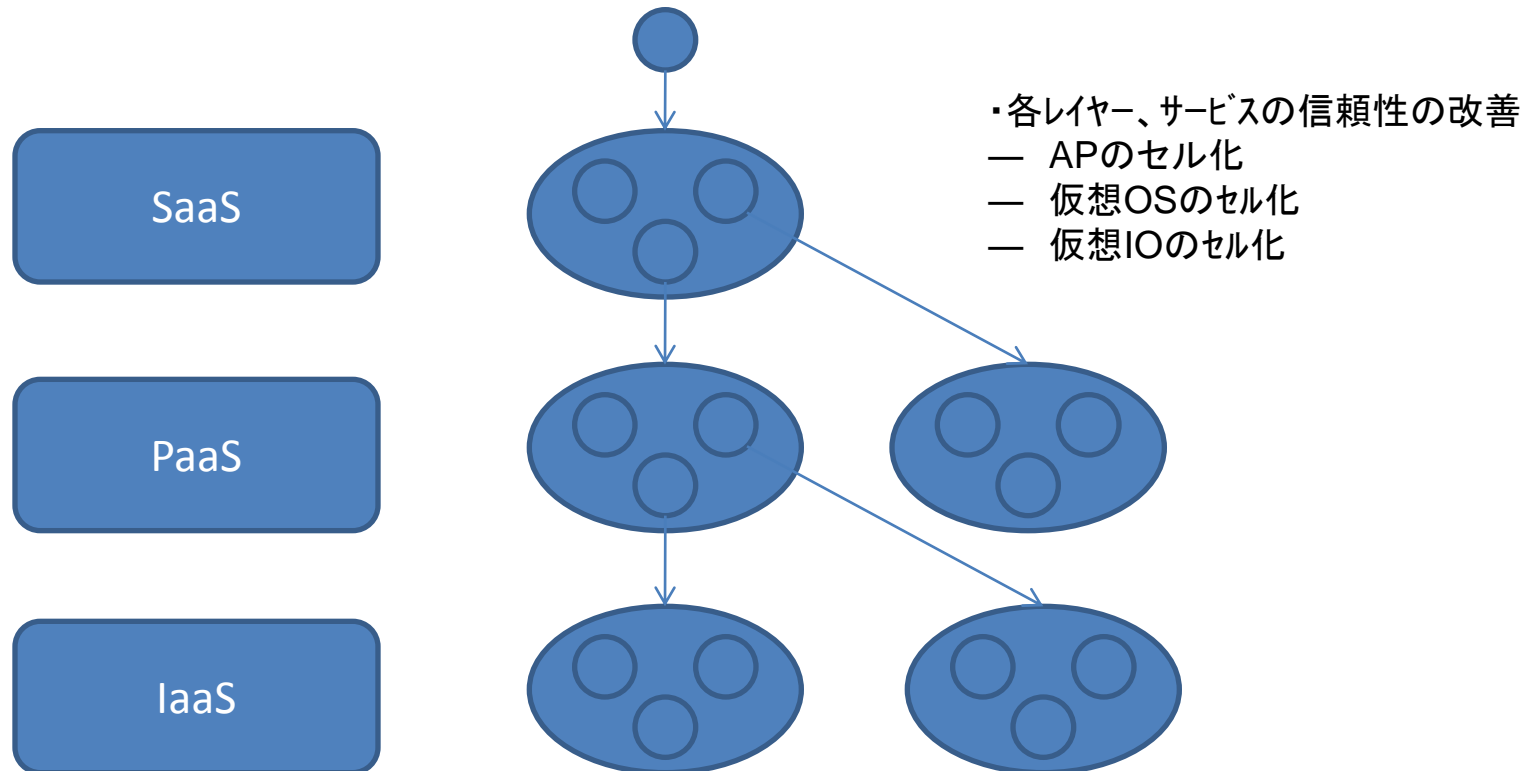
ディスクとメモリの比較(GB)

~ディスクよ、さようなら

GB当り	スペース	価格	電気代	速度
ディスク	大			小
メモリ	小			大

- ・メモリは修復機能は必要ない。エラー通知機能(1ビットエラーの検知)は必要。

セル化の適用



トライテックの着想点

～ Paxosによる超高可用性システムの実現

Paxos実装化の課題を解決

- 大容量データのPaxos合意への取り込み(特願2010-023612)
- 自動キャッチアップ機能とログのキャッシュ化(特願2010-127356)
- マスターの自律的選択機能(特願2010-127357)
- セッション管理とイベント管理及び参照の分散化(特願2010-538664)
- マルチPaxos実現方式(特願2010-123944)

☆上記機能について、特許出願。

企画(予定)

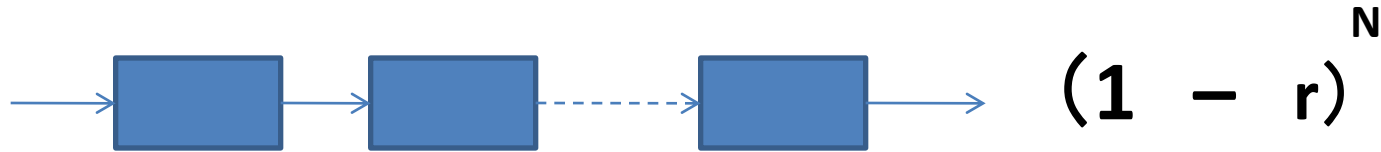
- Android
 - PFSストレージクラウドの実装(作業中)
- コンバータ(既存システムに挿入)
 - HTTPクラウド
 - SIPクラウド
 - xjIrisクラウド(セールスフォース相当)
- メモリDBクラウド
- 機器死活監視クラウド
- RAIDクラウド

Ⅱ部 信頼性

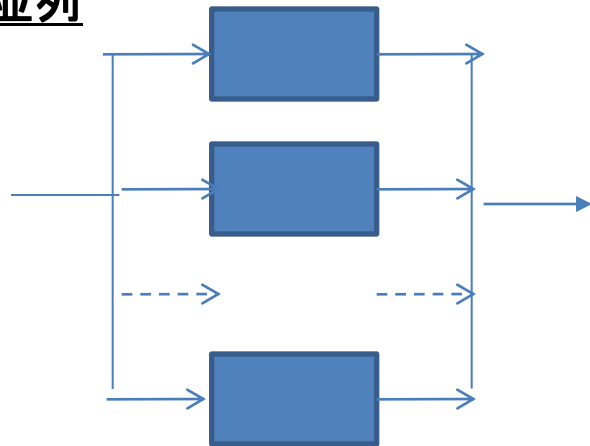
- 信頼性モデル
 - 直列
 - 並列
 - 過半数(セル)
- 堅牢性
 - プロセシングの堅牢性
 - データの堅牢性

信頼性モデル(故障率r)

直列



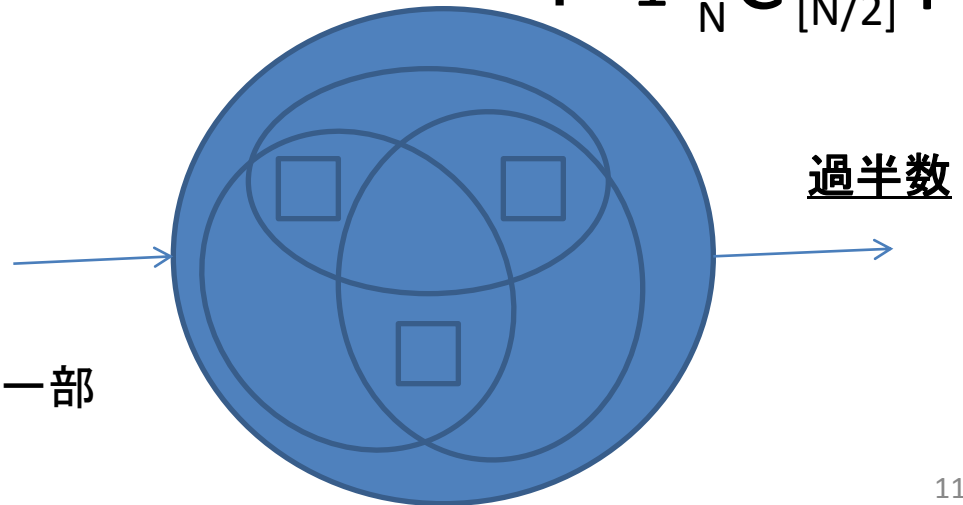
並列



$1 - r^N$

$$\sum_{i=[N/2]}^N \binom{N}{i} r^{N-i} (1-r)^i$$

$$\doteq 1 - \binom{N}{[N/2]} r^{[N/2]}$$



直列、並列は、組合せの二項展開の一部

過半数方式(3/5の例)

$$\sum_{i=3}^5 {}_5C_i r^{5-i} (1-r)^i$$

個別の信頼性	セルの信頼性
0.9	0.99144
0.95	0.998842
0.96	0.999398
0.97	0.999742
0.98	0.999922
0.99	0.99999
0.995	0.999999

90%で99%
99%で99.999%
99.5%で99.9999%
となる。
⇒ファイブ9は容易に達成できる?!

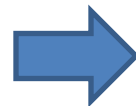
2つの堅牢性

- プロセッシングの堅牢性

$$\sum_{i=0}^N C_N^i r^{N-i} (1-r)^i \rightarrow \text{協働で任意の高信頼性の実現}$$
$$\doteq 1 - N C_{[N/2]}^{[N/2]} r^{[N/2]}$$

- データの堅牢性 (1台でも生きていればよい)

$$1 - r^N$$



状態、アクション列の永続化は必要ない
⇒ キャッシュ化
⇒ 高速処理
⇒ メンテナンスが可能
☆ チェックポイント方式はキャッシュ情報が消失

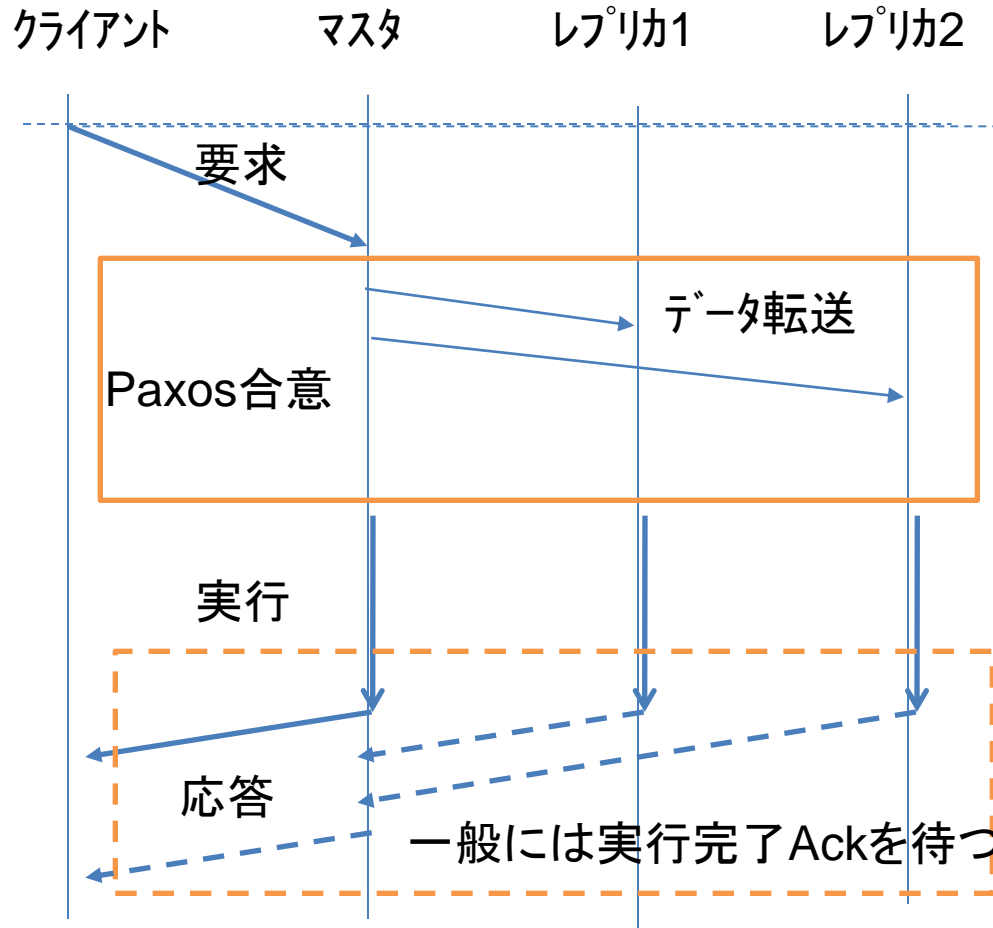
二重化構成

- ・ プロセッシングの故障率は $2*r$ で倍悪くなるが
- ・ データの故障率は $r^{**}2$ でべき乗で改善される
⇒ 復旧を超高速化する？

故障と自律回復方式

- 再起動と再参入(プロセッシングの堅牢性)
 - 再起動時には過半数の持続を確認し再参入する
 - この確認時にセルからSnapshotを取得する
 - 過半数を確認できなければ放棄する
- 1台でも生きていれば(データの堅牢性)
 - 直接サーバからSnapshot取得、catchupで再参入する
 - 順次再参入し、過半数を再構築できる
- バグと雑音の区別
 - レプリカが同一箇所パニックすればバグ
 - 個々のパニックは雑音

性能モデル

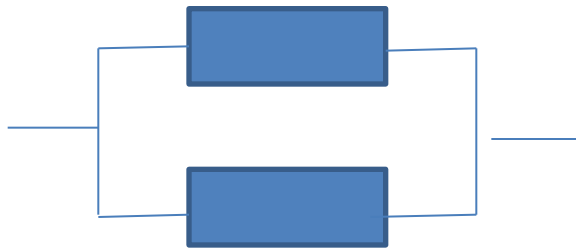


- 内部データ転送時間
 - Paxos合意時間。
 - レプリカのAckは必要ない。
 - 実行は並列処理
- ☆合意コストと転送コストの増加
要求・応答・実行コストが大であれば、無視できる。

基本はキャッシュでOK
レプリカ自体がバックアップ
実行は突き放し
⇒高速!!

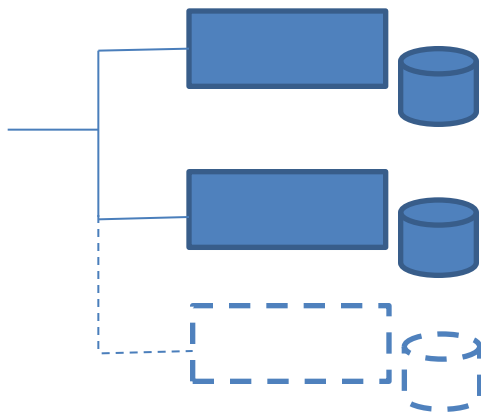
考察:二重化について

状態を持たないタイプ(通信)⇒並列型



$1-r^{**2}$ の故障率で進行する

状態を持つタイプ(RAID)⇒直列型

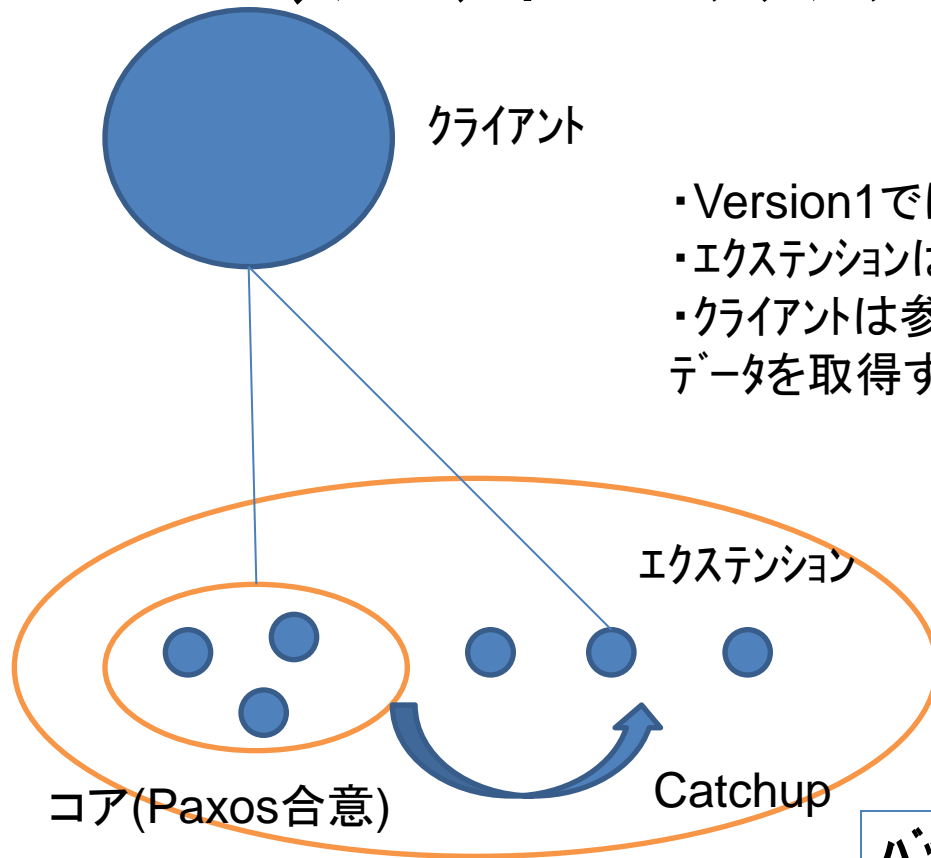


状態は $1-r^{**2}$ の故障率で保持されるが、
1台が故障を起こすと復旧されるまで進行が停止する
⇒3台の過半数方式とすれば進行する
⇒RAIDクラウド?

RAID二重化は定足数2過半数2のPAXOS
⇒PAXOSにすれば突き放しなので速い

Paxosエクステンション

~リアルタイムバックアップをいくつでも



クライアント

- ・Version1ではエクステンションをサポート
- ・エクステンションは常時catchupを行い、レプリケーションを作る
- ・クライアントは参照(非同期)時、負荷の低いサーバからデータを取得する

効能

- ・リアルタイムバックアップ
チェックポイント方式は遅い
- ・参照の負荷分散

バックアップ

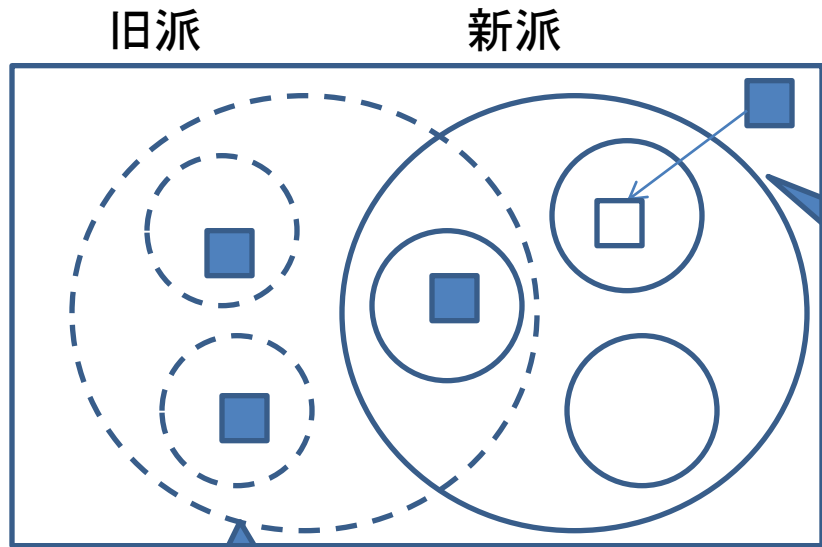
- ・コアが故障したらエクステンションをコアに参入
- ・コアが構成できない時には大修復

ハードは技術革新でリプレースが必要なのでどの道、大修復がある

Ⅲ部 Paxosモジュール

- 過半数
- Paxos合意
- 実現のための課題
- ラウンド値
- マスター選出
- 帯域外データ
- Multi-Paxos
- Catchup
- API

過半数(Paxos合意)



新たな提案値は最弱として投入されるが、既提案値があれば拒否される

新派が形成される
合意値があれば継承
なければ投入

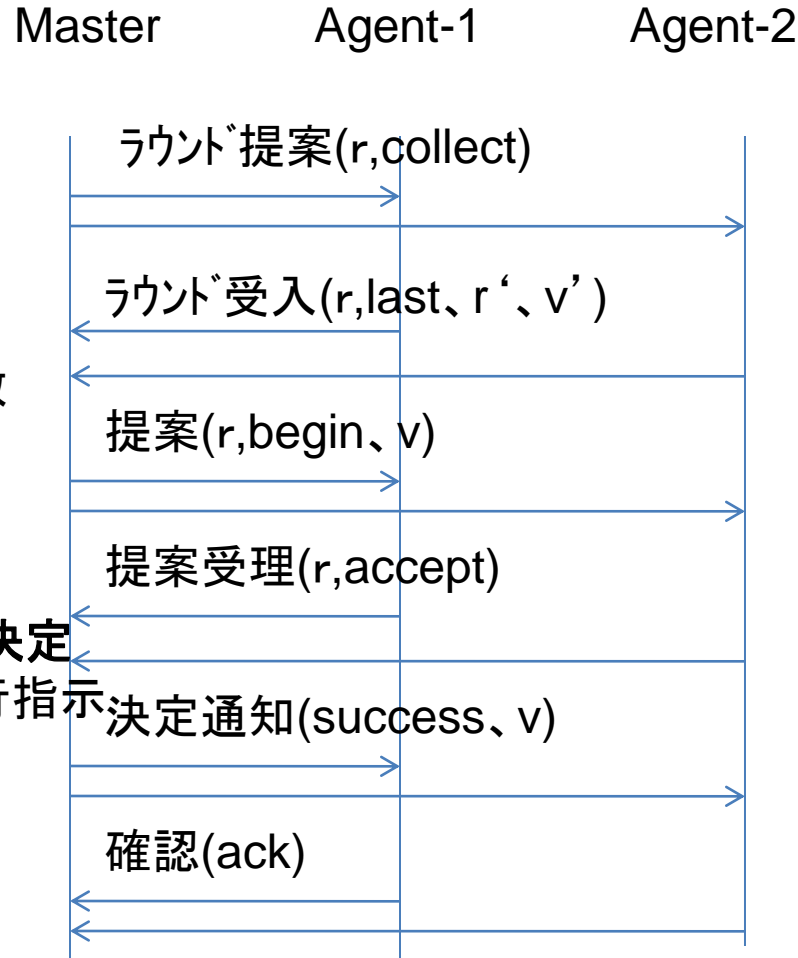
旧派による合意形成

- ①新派には少なくとも1つ旧派の合意がある
- ②投入提案値は最弱なので旧派の合意値が伝播する

☆過半数の賛成を得た決定に、たとえ後から参入しても、全員が従うことを保証する

各自が投票し、任意の過半数A,Bで合意が成立した時、少なくとも一つは共通であるので、A,Bは共通の合意を持つ。即ち、A,Bは自律的にそれぞれ合意を有するがA,Bの合意は同一である。

Paxos合意

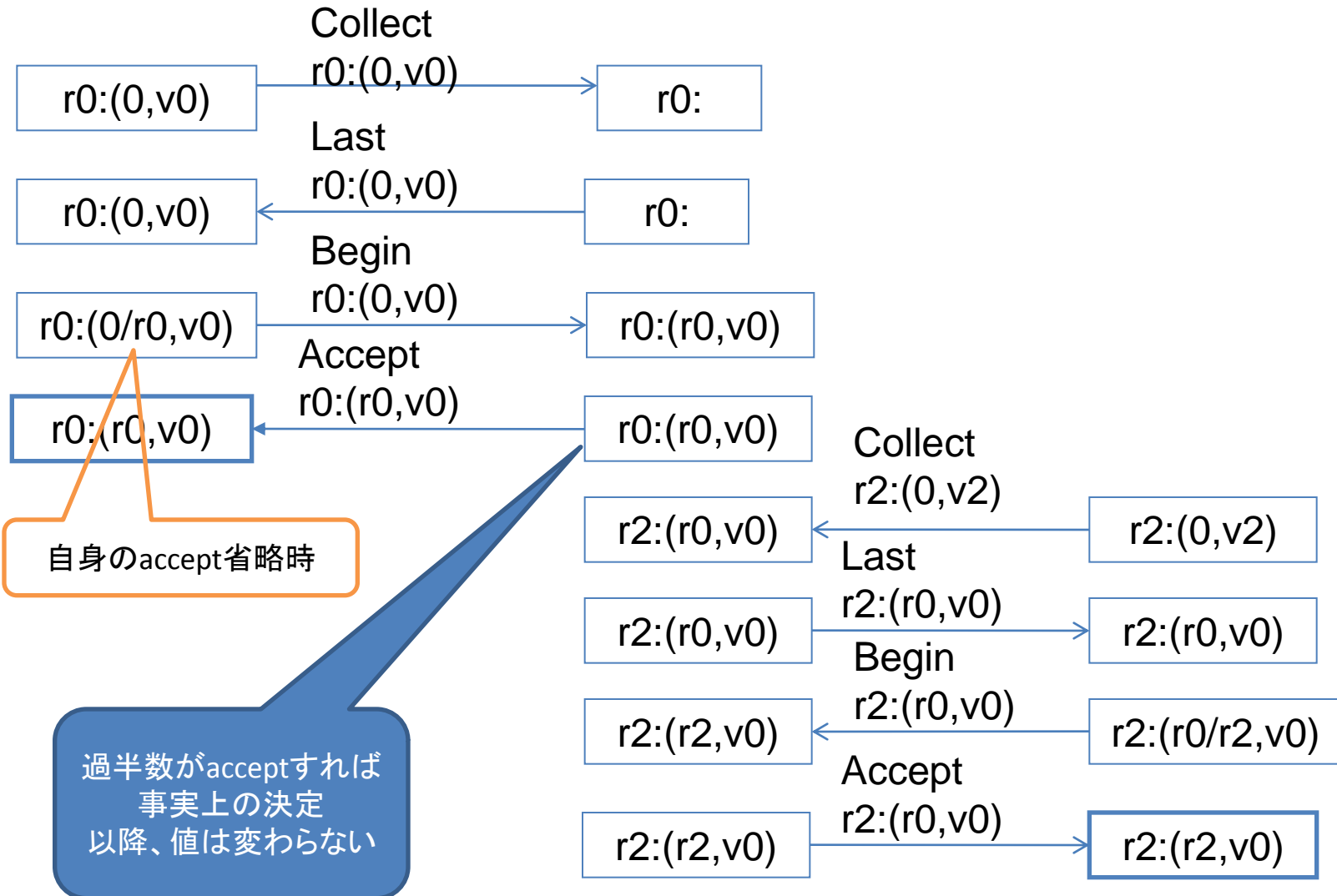


- ①ラウンド(r)は、一意な強さ
- ②提案値(v)は、投入時は最弱とする
- ③collectで提案を集める
- ④lastで最強を保持
- ⑤過半数のlast応答で③をbegin送信
- ⑥過半数のaccept応答で確定
- ⑦success,ackはvの実行指示

ポイント

- ・最初の提案値は最弱で、lastで最強が選択され、beginで現在強度が付与される。最強が次に引き継がれる。
- ・過半数であれば旧派と新派で少なくとも1個は共有され、提案値は引き継がれる。
→過半数のacceptの決定は乱されない
- ・マスターが複数のとき、合意は進展しない。

ケース-1

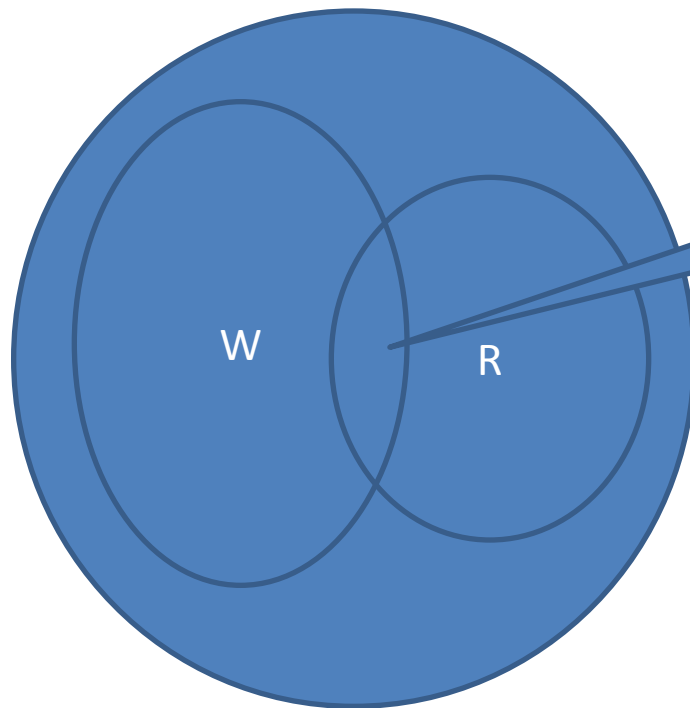


実現のための課題

- 一意なラウンド値
- 唯一のマスター選出
 - Paxosでマスターを決定する(Master lease)
 - Paxosは唯一のマスターを前提としているので矛盾！
 - 衝突したら乱数で回避
- 値(大容量データ)
 - RDMA、FTP等の別データ通信手段は？
- Paxos列
 - インスタンスは連続する
 - 順序管理
- Catchupの発生
 - 合意決定時の少数派は遅れる
- クライアントとセルの接続方式
- 参照の分散

考察: Majority方式

- ・更新(W)、参照(R)として定足数(N)に対して、 $W+R>N$ のアルゴリズム
- ・Wによる更新はRは必ず知ることができる
- ・Wを過半数とするとPaxosと同じか



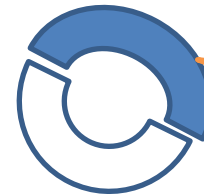
$W+R>N$ であれば共通項がある

- ・Wが過半数より小さければ、複数の更新データが存在する
- ・Rでは複数のデータを取得できるが取得者が判断する

ラウンド値

- 強さ、ユニークな順位(無限の自然数)
 - 複数のサーバは独立しており、共通の自然数を共有できない
 - コンピュータでは無限の自然数を実現できない
- 自然数を(通番、サーバID)とし、これをラウンドとする
 - 通番を第1順位、サーバIDを第2順位とする
 - 神(人)の手によるユニーク性(共有できる)
 - 最弱は通番0とする
- 無限は循環とし、通番の差 $((int)a - (int)b)$ で大小およびサーバIDで順位を判断する
 - 各サーバは0を除いて通番をインクリメント
 - ビット幅nであればn-1幅で有効

2進レジスタの特性を利用

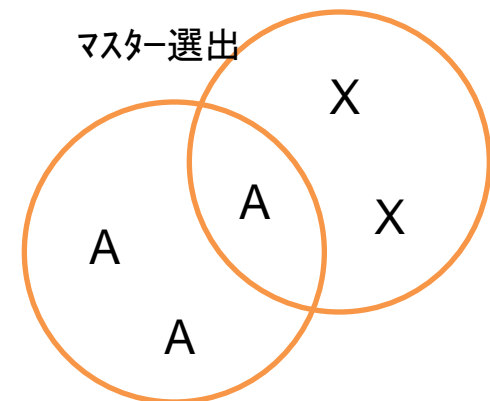
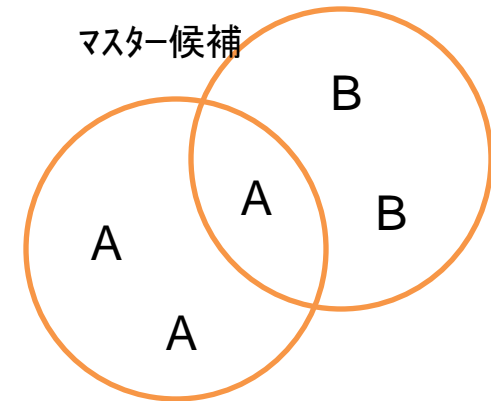


半分で有効

Masterの自律的選出

～最長老方式 若者ではない！

- 全サーバが同意できる方式が必要
- 各サーバは、死んだり生き返ったりする
- そこで、安定した絶対基準として時刻を選ぶ
 - 立ち上げ時あるいはmasterを降りた時を誕生時刻とする⇒頻繁なサーバ交替を抑止
 - サーバidとすると、上位サーバが参入する度にMaster遷移が発生する---好ましくない
- 前提
 - 生きているサーバ同士は常時通信している
- アルゴリズム
 - ① (過半数のサーバから)最長老をmaster候補として他サーバに通知する
 - ② 過半数のサーバのmaster候補が一致していればmasterと認識する⇒各サーバが自律的にmasterを認識する



アウトバウンドデータ(帯域外データ)の取り込み - 1

- 問題点

- Paxos調停はBy Value(値渡し)である

- 原子性、孤立性を確保

- UDP通信(最大64KB)では大容量データを送れない

- collect,lastはテーブル単位

- 全提案値は送れない

- begin,accept,successはインスタンス単位

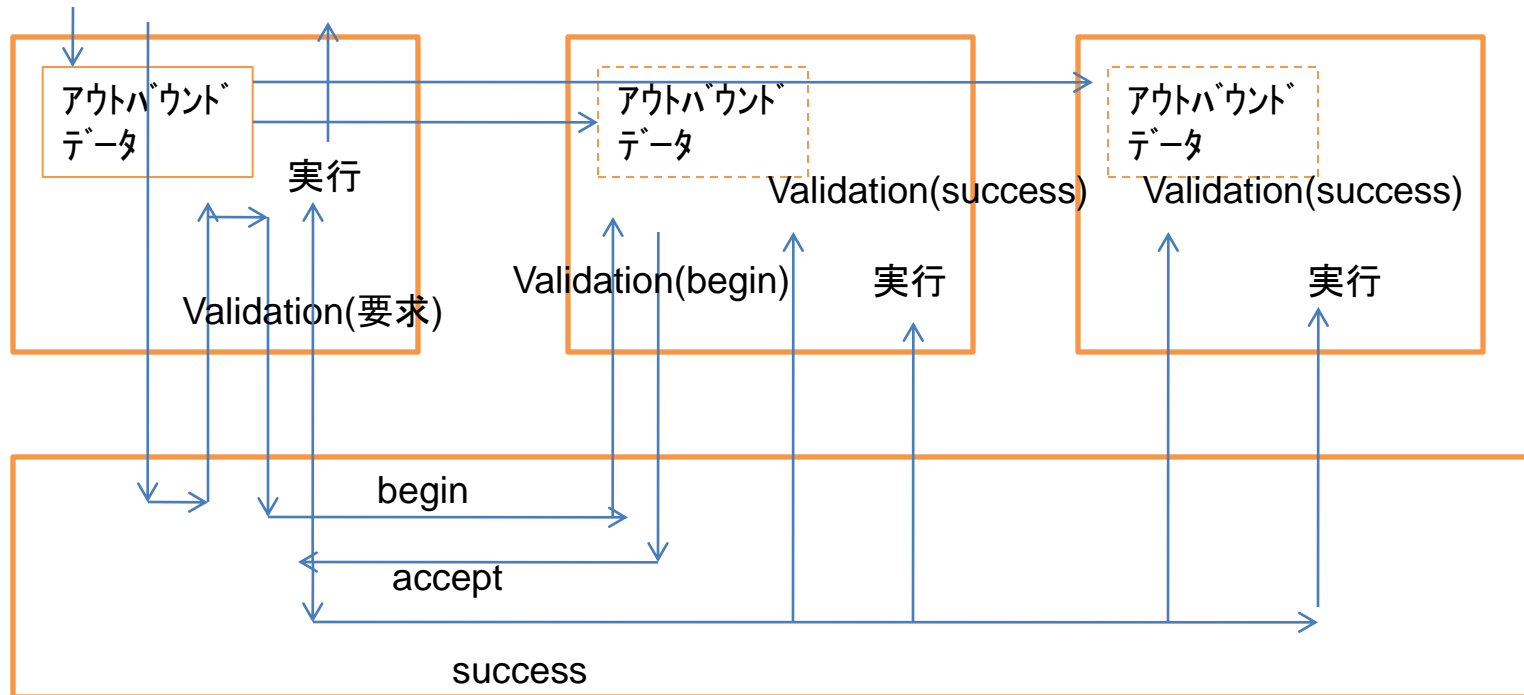
- インスタンス毎の提案値は送れるが最大64KBである

- そもそも大容量データは別メカニズムが一般的

⇒ Meta-Dataしか扱えない

アウトバウンドデータの取り込みー2

データ投入 要求 応答



- Masterにアウトバウンドデータを投入する
- Paxos調停の各段階でValidationを発行する
- Validationでアウトバウンドデータを取り込む
- 失敗すれば、調停が進まない
- クライアントは一定時間後、再試行

要求受付時にアウトバウンド化することも考えられる

アウトバウンドデータの取り込みー3

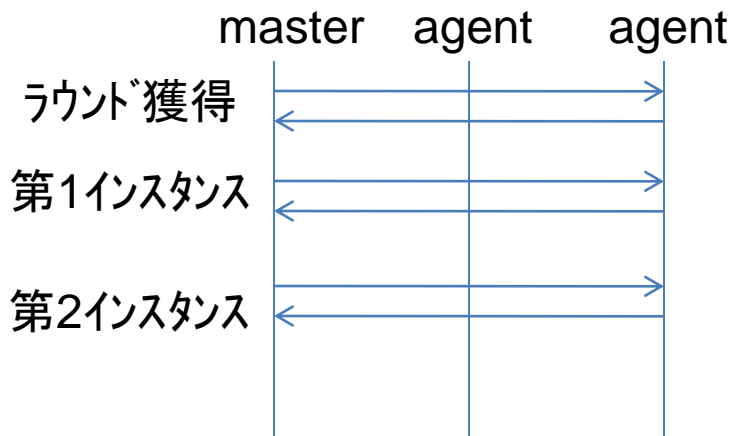
Validation

- 要求
 - 要求前にクライアントがデータ投入
 - Paxos投入前に確認
- begin
 - begin発行元からデータ取得
- Accept
 - masterとagentが同じ場合、決定前に確認
 - master交替でbeginが欠落し、決定がなされる可能性があるので、決定前にaccept元からデータ取得
- Success
 - beginが欠落し、successが到達する。success元からデータ取得

通常

Multi-Paxos

- ラウンド獲得後は、begin、acceptのみで可
 - 強いラウンドが出現すればPaxos調停は進展しない
 - 過半数の決定は新ラウンドに引き継がれる
- 複数のPaxos合意に適用できる



複数がMaster立候補すると？
進行しないので唯一にする

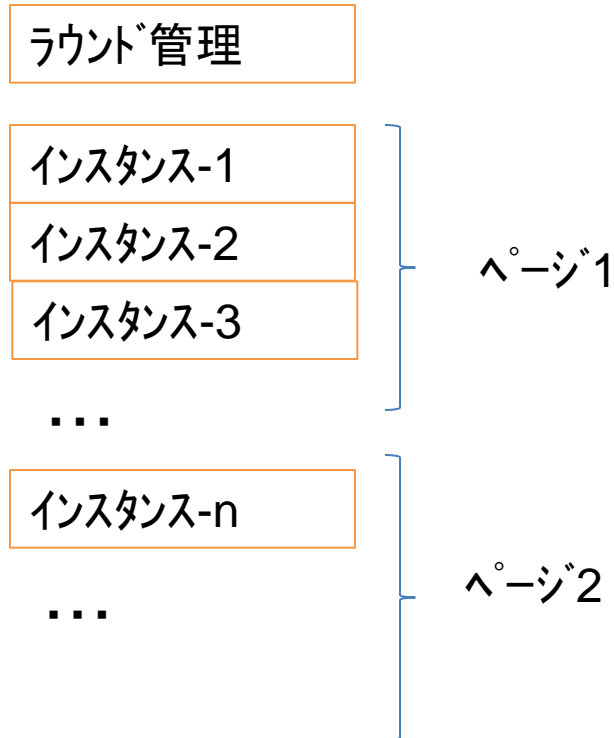
→ただのquorum(過半数)となる

つまり、マスターが決まればラウンド獲得は
必要ない？

答:ラウンド獲得では提案を集め、旧派の
最強を引き継ぐ

ラウンド獲得後は提案権を保持する

Multi-Paxosの実現方式



①masterは無限のインスタンスについてcollectで決定値、提案値をagentに通知する

②agentは、決定値を無条件に受け入れ、masterの知らない決定値、提案値をlastでmasterに通知する

③masterは決定値を無条件に受け入れる

④masterは**インスタンス毎**にbeginをagentに通知する

⑤agentはacceptをmasterに通知する

⑥masterはsuccessでagentに**実行指示**を通知する

⑦agentはackで実行完了を通知する

ページの導入

無限テーブルは実装できないのでページ分割を行う。ページ内の全てのインスタンスが決定しないとページ更新を行わない

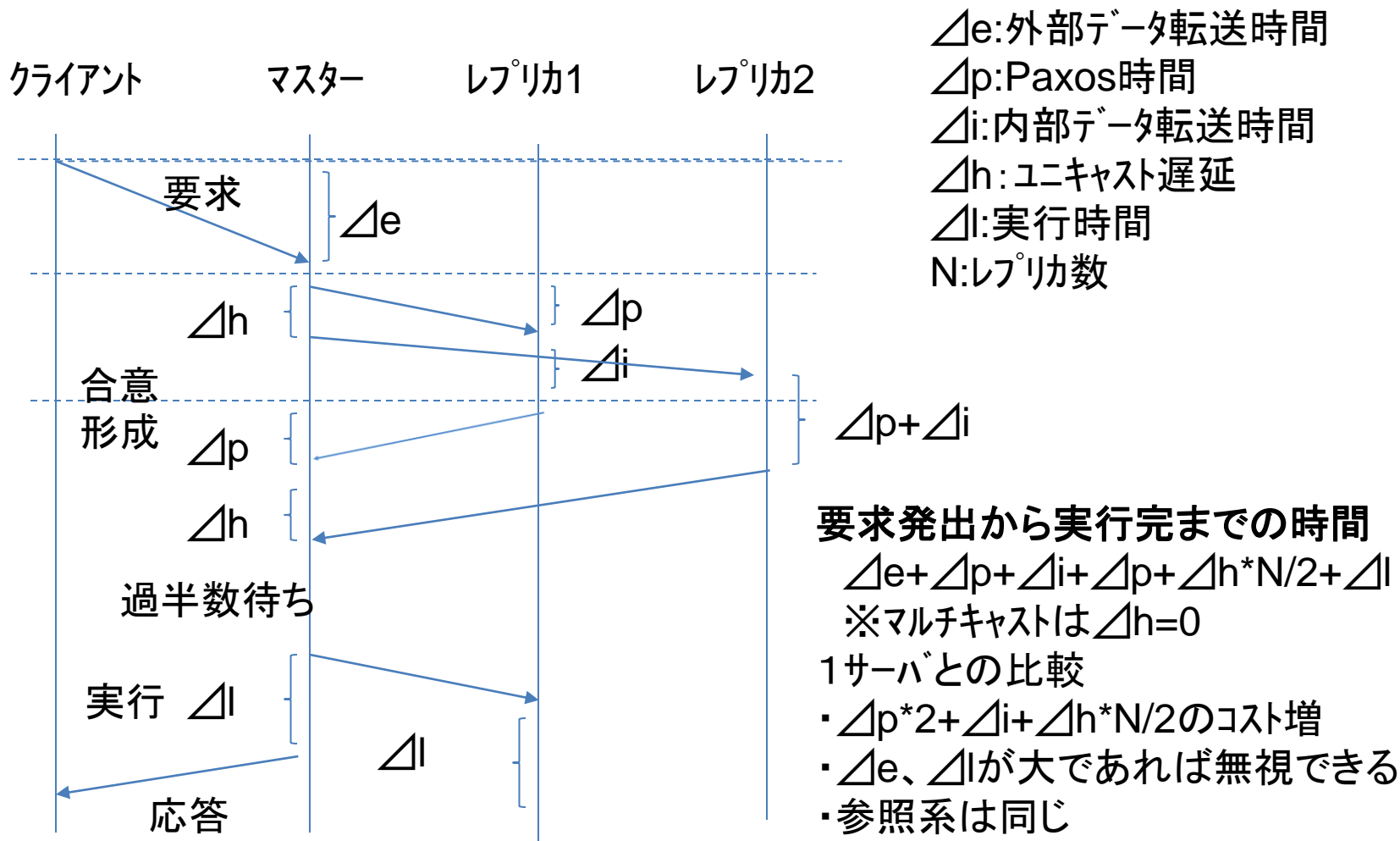
- ・クライアント多重化を実現できる
- ・実行順序の保証が課題となる

Catchup

～並行動作

- 少数派は遅れるので追いつく必要がある
- catchupの契機
 - 受信した連続する最大決定通番と保有する連続する最大決定通番を比較する
 - 遅れていればcatchup要求を発行する
- catchup要求
 - 発行は開始通番と終了通番とする
 - また、Paxos実行抑止時間を付する
 - この時間はPaxosに反応しない
- catchup応答
 - 開始通番から終了通番までのアクションを応答する
 - 終了通番受信時にはcatchup終了を通知し、抑止を解除する

性能(更新系)



PaxosライブラリのAPI一覧

- 定数
 - PAXOS_SERVER_MAX コアサーバ数
 - PAXOS_SERVER_MAXIMUM 全サーバ数
 - PAXOS_MAX 多重度
 - PAXOS_FIRE_MAX キャッチアップ最大数
 - PAXOS_COMMAND_SIZE 帯域内コマンド最大値
- Paxos構造体の初期化
 - PaxosGetSize サイズの取得
 - PaxosInit 構造体の初期化
- 上位モジュールのタグ付け
 - PaxosSetTag タグ設定
 - PaxosGetTag タグ参照
- Paxosの開始
 - PaxosStart 開始

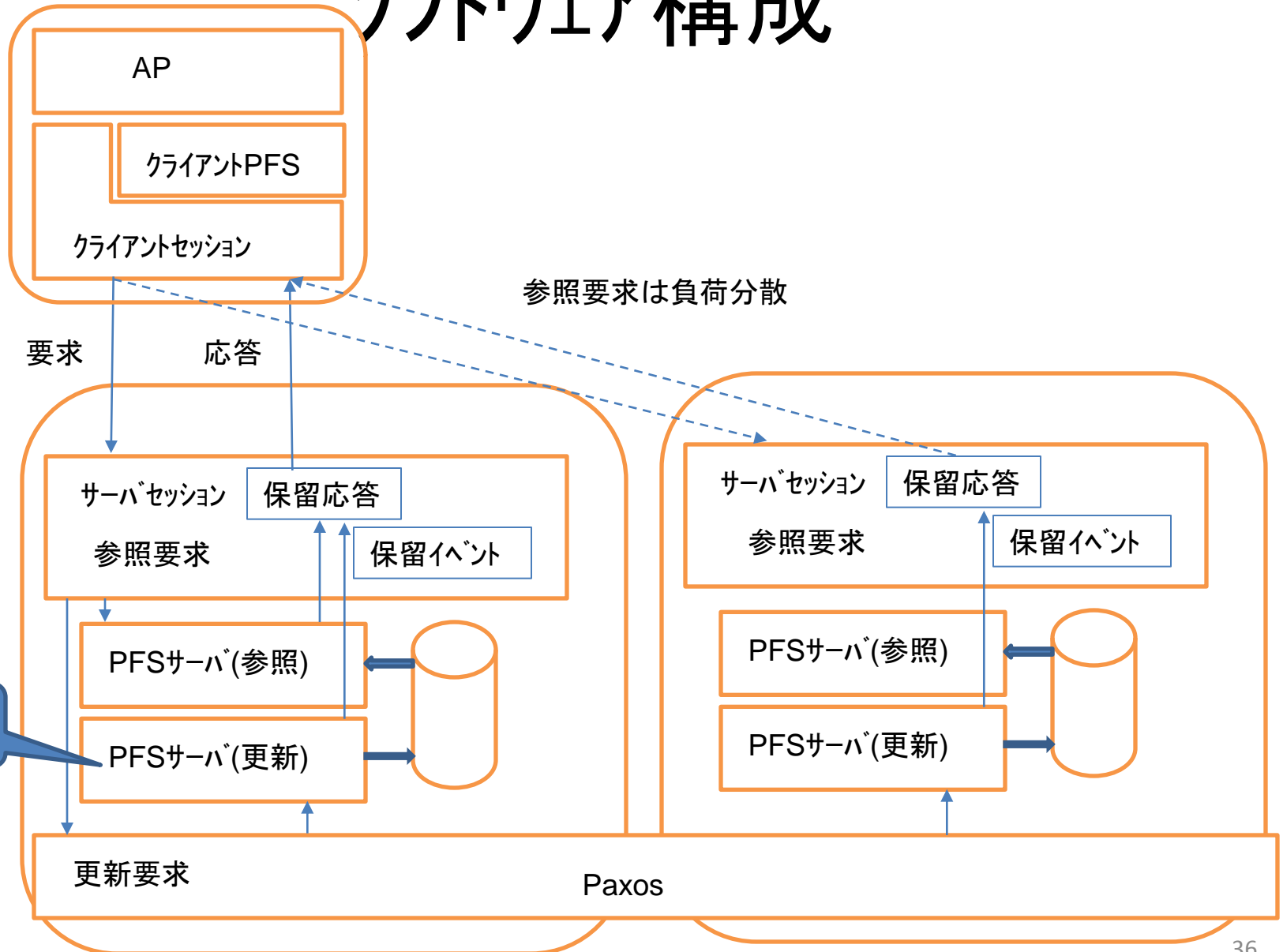
PaxosライブラリのAPI一覧

- 投入と実行
 - PaxosRequest 要求投入
 - PaxosRecvFromPaxos 合意受信
- その他
 - PaxosIsMaster マスターか
 - PaxosGetMaster マスター参照
 - PaxosMasterElectionTimeout 最大マスター選出時間
 - PaxosDecidedTimeout 最大決定時間
 - PaxosCacheTimeout キャッシュ保持時間

IV部 Sessionモジュール

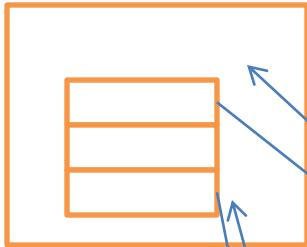
- ソフトウェア構成
- セッション管理
 - 接続、Master選択、参照サーバ選択
- 順序管理
- イベント管理
- 参照の負荷分散

ソフトウェア構成



セッション管理

Client



絶対識別子

セッションID:(INアドレス、プロセスID(,付与ID))

トランザクションID(INアドレス、プロセスID(,付与ID,)通番)

Cell

Masterは？

MasterはXXX

初期は仮マスターを立てる

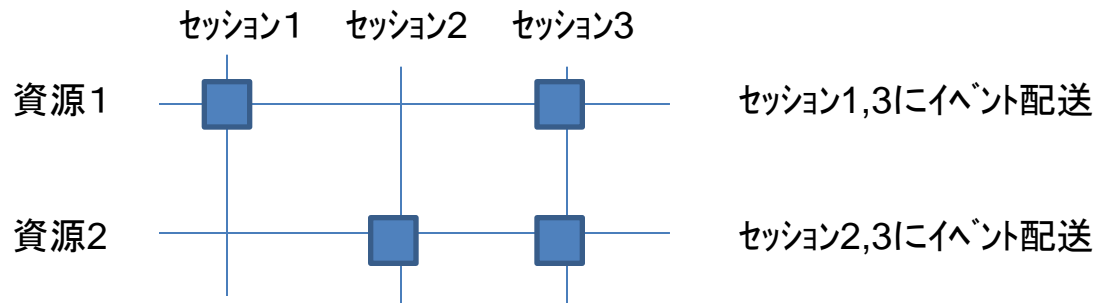
- ① 接続
 - a. connectで接続する
 - b. セッションIDで通信端点(セッション)を作る
- ② master選択(masterが不明の場合)
 - a. 接続済みのサーバに「Masterは？」を問い合わせる
- ③ master切断で選択モード
- ④ masterイベント
 - a. Cellから「Master選択イベント」が到着したらmaster選択待ちを解除する
 - b. 新旧混在認識時は選択モード
- ⑤ ガーベージ
 - a. Masterとなってから一定時間後、サーバはセッションのガーベージを行う
 - b. ClientはMaster変更を検知したら速やかにMaster接続を行う

順序管理

- セッション毎に順序管理をする
 - 要求通番 要求/応答の識別
 - 直近の更新通番 参照用
- クライアント
 - 「要求通番」と「直近の更新通番」で要求を発行する
 - 「要求通番」に対する応答を受け付け、重複応答は破棄する
 - 一定時間経っても応答がなければ再度要求する(重複要求)
- サーバ
 - 参照要求は「直近の更新」実行終了確認後、実行する
 - 要求受理時、実行後、それぞれ通番を登録する
 - 重複要求は、受理通番を参照して破棄する
 - 応答があれば、応答を返す(重複応答)

イベント管理

- イベント取得用のセッションを起こす
- イベントの設定/解除(直ちに制御が戻る)
 - イベント資源とセッションを関連付ける
- イベントの取得(イベントがなければ制御が戻らない)
 - 各種資源のイベントが返される
 - 取得要求がなければ保留される



参照の負荷分散

- 各サーバの負荷
 - 各サーバで負荷を計算する(例えば、接続数)
 - サーバ間で負荷情報を交換する
 - サーバはクライアントに最小負荷のサーバを通知する
- クライアント
 - 直近の更新通番を記憶する
 - 通知された最小負荷のサーバに参照要求
- サーバ
 - 参照要求はPaxos投入をしない
 - 更新通番実行を待つ

クライアントAPI

- セッション構造体の初期化/破棄
 - PaxosSessionGetSize サイズの取得
 - PaxosSessionInit セッション構造体の初期化
 - PaxosSessionDestroy セッション構造体の破棄
- セッションのオープン/クローズ
 - PaxosSessionOpen セッションのオープン
 - PaxosSessionClose セッションのクローズ
- 要求/応答
 - PaxosSessionReqAndRplByMsg 要求/応答
- イベント
 - イベント資源の設定/解除は上位APIによる
 - 配送は、PAXOS_SESSION_EVENTコマンドでの要求/応答による

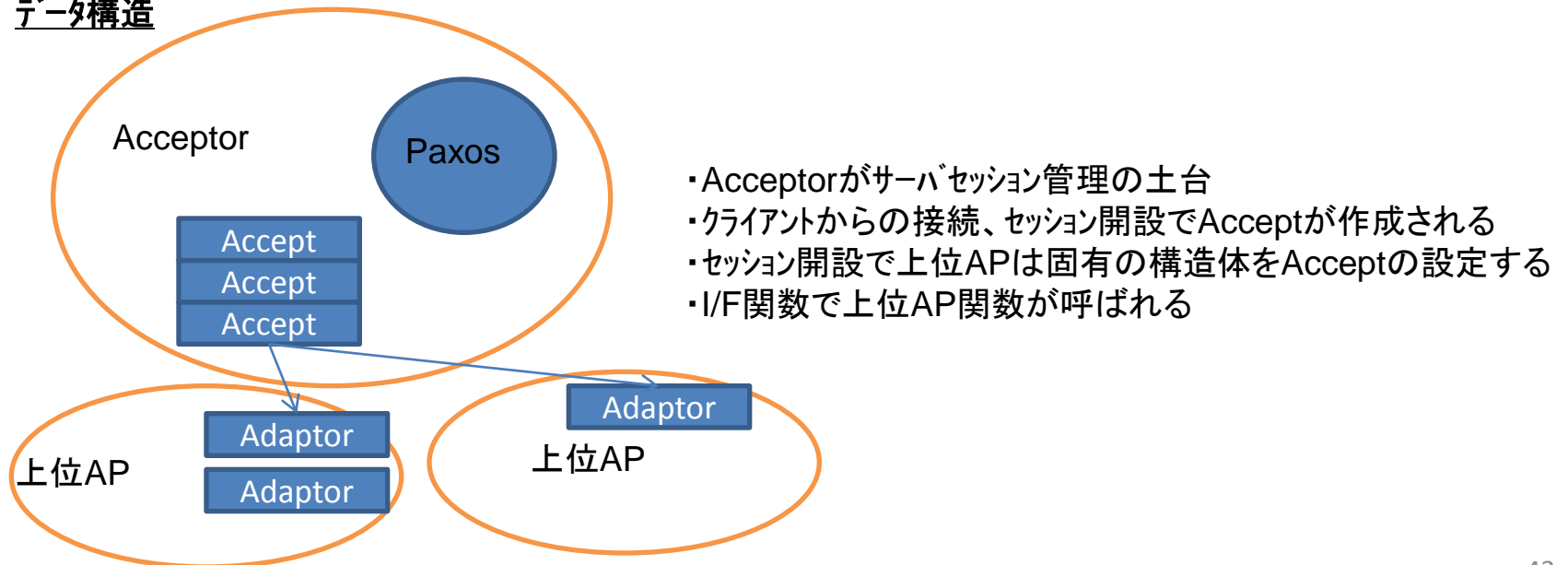
サーバAPI一覧(1)

- Acceptorの初期化
 - PaxosAcceptorSize Acceptor構造体のサイズ
 - PaxosAcceptorInit Acceptor構造体の初期化
 - PaxosAcceptorIF_t I/Fコールバック関数群
- Acceptorの開始
 - PaxosAcceptorStart Acceptorの開始
- 上位構造体のAcceptへの設定/参照
 - PaxosAcceptSetTag 設定
 - PaxosAcceptGetTag 参照
- 重複要求のチェック
 - PaxosAcceptIsAccepted 重複要求のチェック
- 応答の送信
 - PaxosAcceptReply 応答の送信(マスターの時)
 - PaxosAcceptSend 応答の直接送信

サーバAPI一覧(2)

- イベント
 - JOINT リンク設定
 - UNJOINT リンク解除
 - PaxosAcceptEventRaise イベント励起

データ構造



V 部 PFSアプリケーション

- 機能

- ネットワークLock機能
- ファイル機能
- エフェメラルファイル機能(機器監視)
- テレクトリ機能
- イベント機能
- ネットワークキュー機能

- API

Paxosファイルシステム

- Lock機能
 - lockRead/lockWrite
 - unlock
- ファイルアクセス機能
 - write/read/delete
 - outbound write
 - ephemeral
 - 死活監視用(クライアント消滅時に消失)
- ディレクトリ操作機能
 - mkdir/rmdir/readdir
- イベント管理
 - EventDir(ディレクトリ変更)/EventLock(ロック要求イベント)

クライアントAPI一覧(1)

- セッション構造体の初期化
 - PaxosSessionGetSize サイズの取得
 - PaxosSessionInit セッション構造体の初期化
- セッションのオープン/クローズ/スナップショット
 - PaxosSessionOpen セッションのオープン
 - PaxosSessionClose セッションのクローズ
 - PaxosSessionSnapshot スナップショット指示
- ファイル操作(Open/Close/Lseekはクライアントのみ)
 - PFSOpen ファイルの新規作成/オープン
 - PFSClose ファイルのクローズ
 - PFSDelete ファイルの削除
 - PFSLseek オフセット設定
 - PFSRead ファイルの読み込み
 - PFSWrite ファイルの書き出し(大容量は帯域外処理)

セッションの初期化(1)

- 通信/メモリ関連の外出し
 - void* Connect(struct Session*, int Id)
 - Id番目のサーバに接続し、ハンドルを返却する
 - int Close(void* pH)
 - ハンドルをクローズする
 - int GetMyAddr(void* pH, struct sockaddr*)
 - 自アドレスを取得する
 - int SendTo(void* pH, char* pBuf, size_t Len, int Flags)
 - データを送信する
 - int RecvFrom(void* pH, char* pBuf, size_t Len, int Flags)
 - データを受信する
 - void* Malloc(size_t Size)
 - メモリの取得
 - void Free(void* pAddr)
 - メモリの解放

セッションの初期化(2)

- PaxosSessionGetSize() セッション構造体のサイズ
- PaxosSessionInitの引数
 - struct Session* pSession セッション構造体ポインタ
 - void*(*fConnect)(struct Session*, int) 接続関数
 - int (*fShutdown)(void*,int) 切断関数
 - int (*fClose)(void*) クローズ関数
 - int (*fGetMyAddr)(void*,struct sockaddr*) 自アドレス取得
 - int (*fSendTo)(void*,char*,size_t,int) 送信関数
 - int (*fRecvFrom)(void*,char*,size_t,int) 受信関数
 - void* (*fMalloc)(size_t) メモリ取得関数
 - void (*fFree)(void*) メモリ解放関数

セッションのオープン/クローズ

- PaxosSessionOpen(struct Session*,int No, bool_t Sync)
 - セッションをオープンする
 - No ユーザ付与番号
 - Sync 更新と参照の同期
- PaxosSessionClose(struct Session*)
 - セッションをクローズする

ファイル操作

~サーバの状態はステートレス

- `File_t* PFSOpen(struct Session* pSession, char* Path, int Flags)`
 - ファイルは絶対パス名を指定する
 - Flags
 - 0 通常ファイル
 - 1(FILE_EPHEMERAL) 切断後に消滅する一時ファイル
 - 本関数はクライアントにファイル構造体を確保するだけ
 - サーバアクセス時に存在しなければ作成される
- `int PFSClose(File_t* pF)`
 - ファイル構造体を解放する
- `int PFSDelete(struct Session* pSession, char* Path)`
 - ファイルを削除する
- `int PFSLseek(File_t* pF, off_t Offset)`
 - ファイル構造体にオフセットを設定する
- `int PFSRead(File_t* pF, char* pBuf, size_t Len)`
 - オフセットから読み込み、オフセットを更新する
- `int PFSWrite(File_t* pF, char* pBuf, size_t Len)`
 - オフセットから書き出し、オフセットを更新する
 - 大容量データは帯域外処理を行う

ディレクトリ操作

- `int PFSMkdir(struct Session*, char* Path)`
 - ディレクトリを作成する
 - ディレクトリは絶対パス
- `int PFSRmdir(struct Session*, char* Path)`
 - ディレクトリが空の時、削除する
 - ディレクトリは絶対パス
- `int PFSReadDir(struct Session*, char* Path, FileDirent_t* pE, int32_t Ind, int32_t* pN)`
 - ディレクトリのInd番目のエントリから*pN個を読みだす
 - *pNには読み込んだ個数が返却される
 - ディレクトリは絶対パス

ネットワークロック操作 ～勧告ロック

- int PFSLockW(struct Session*, char* Name)
 - writeロックを取得する
 - 取得できるまで永遠待ち
- int PFSLockR(struct Session*, char* Name)
 - readロックを取得する
 - 取得できるまで永遠待ち
- int PFSUnlock(struct Session*, char* Name)
 - ロックを解放する

イベント操作

- `int PFSEventDirSet(struct Session*, char* Path)`
- `int PFSEventDirCancel(struct Session*, char* Path)`
 - イベント取得を解除する
- `int PFSEventLockSet(struct Session*, char* Name)`
 - ロックに要求が来たときのイベント取得を設定する
- `int PFSEventLockCancel(struct Session*, char* Name)`
 - イベント取得を解除する
- `int PFSEventQueueSet(struct Session*, char* Name)`
 - キューのイベント取得を設定する
- `int PFSEventQueueCancel(struct Session*, char* Name)`
 - イベント取得を解除する
- `int PFSEventGet(struct Session*, int32_t* pCnt, int32_t* pOmitted)`
 - イベントを取得する
 - イベントがなければイベント待ちとなる
 - 外出し関数
 - `int PFSEventDir(struct Session*, FileEventDir_t*)` テレクトリイベント
 - `int PFSEventLock(struct Session*, FileEventLock_t*)` ロックイベント

ネットワークキュー操作

- `int PFSPost(struct Session*, char* Queue, int Len, char* Data)`
 - QueueにDataをポストする
- `int PFSWait(struct Session*, char* Queue, int* Len, char* Data)`
 - Queueの先頭からDataを取得する
 - LenにはDataのバッファ長
 - 取得できるまで永遠待ち
- `int PFSPeekMsg(struct Session*, char* Queue, int* Len, char* Data)`
 - PFSWaitと同じだがメッセージは残る
- `int PFSDeleteMsg(struct Session*, char* Queue, PaxosClientId_t*)`
 - メッセージを削除する
- `iint PFSEventQueueSet(struct Session*, char* Queue)`
 - キューにイベント取得を設定する
- `int PFSEventQueueCancel(struct Session*, char* Queue)`
 - イベント取得を解除する

おわり