

分散コンピューティングシステム上のオンラインゲーム開発環境 pfm の設計

弥富 豪宏

近年のオンラインゲーム産業においては、競争の激化から、開発・運営費およびサービス提供のためのインフラへの初期投資を軽減することが大きな課題となっている。課題の解決のためには、オンラインゲームが分散コンピューティングシステムであることによるプログラミングの困難さを緩和し、他のオンラインゲームと簡単にインフラを共用できるようにすることが必要であると考えられる。本稿ではこれらの問題の解決のためにプログラマブルな分散キーバリューストアとしてデザインされ、PaaSとして動作するゲーム開発環境である pfm について述べる。

Pfm : new architecture for online game development environment on distribute computing system

Takehiro Iyatomi

In recent online game industry, it is very important to decrease the cost for development, service maintenance and infrastructure because this region become more and more competitive. To solve these problems, we need to decrease the difficulty of programing on distribute computing system and enable to share infrastructure with other online games. In this paper, I explain new online game development environment called 'pfm' which is designed as programmable distribute key value store and run as PaaS, and show it can solve above problems.

1. はじめに

近年のオンラインゲーム市場においては、どのゲームを見ても基本的には同じようなゲームプレイ体験を提供するようなものになってしまっており、独創性の高いアイデアを元にしたような作品のリリースが少ない。なぜならオンラインゲームの開発は数年に渡る開発期間と、10億円規模での予算を必要としており、ビジネスとして収益をあげるために多数のユーザーを集めることが要求されているため、独創的なアイデアが魅力的であっても、失敗のリスクを考えると多数のユーザーが既に親しんでいるようなゲーム内容にしたほうが良いというビジネス上の圧力が強く働くからである。

どのゲームをプレイしても本質的な仕様がかわらないため、ゲームのソーシャル的な要素（知人がいるかどうか）といった部分だけが重要になり、新しいゲームを積極的にプレイすることが少なくなる。これによりさらに新規のオンラインゲームの開発リスクが増加し、ますます独創的なアイデアを適用する余地が小さくなるという負のスパイラルが発生している。

これはオンラインゲームという分野のゲーム性の発展を阻害しており、極めて大きな問題である。したがってオンラインゲームの開発および、その後の運営のためのコストを大幅に軽減し、ビジネスとして収益のあがるユーザー数のボーダーラインを下げるのが大きな課題となっていると考える。

現在のオンラインゲームの開発・運営においてコストがかかっているのは、分散コンピューティングシステム上での難解なプログラミングと、サービスを提供するインフラへの初期投資である。ウェブアプリケーションの分野では Ruby on Rails のように、開発の難易度を大きく下げたり、クラウドコンピューティングのようなインフラのコストを低する技術が生まれ、上記のようなコストを大きく減少させることが出来ていると言える。

一方でオンラインゲームの分野においては、後述するようにパフォーマンスへの要求がよりシビアであるため、上記のような技術をそのまま使うことは難しいが、独自にこの問題を解決しようとする動きはあり、幾つかのフレームワークがリリースされている。しかし、以下のような理由から、十分に問題を解決出来ているとはいえない状況である。

- **開発のためのプログラミング作業の難度が高い**

既存のフレームワークは中級以上のプログラミング言語の知識を必要としたり、軽量のスクリプト言語で開発を行える場合でも、簡単にシステムの負荷超過の問題を起こせてしまったりするため、未だゲームの内容をプログラムに落とし込むために専門知識を持ったプログラマーが必要である。そういったプログラマーの人件費は大抵の場合高額であり、したがって開発費を十分に抑制することが出来ない。

- **インフラのコストを削減するという観点が欠けている**

オンラインゲームのサービス開始時には採算ラインの見込みよりも相当多数のユーザーのためのインフラを用意することが常識となっている。サービスの開始時に集めたユーザー数がその後のサービスの利用人数を決定することが多く、最初にたくさんのユーザーを集めることが成功のための絶対条件となっているためであるが、これは新規にオンラインゲームを提供する場合には大きな負担である。既存の手法は1つのオンラインゲームの開発を容易にするという点にフォーカスしており、この問題にたいしては何らかの改善を与えていない。

著者はこれらの問題に十分な解決を与えるため、分散コンピューティングの一方式である PaaS に注目した。本稿では、プログラマブルな分散キーバリューストアとしてデザインされ、PaaS として動作するゲーム開発環境 **pfm** の提案と設計の技術的解説を行う。

pfm は、本質的にはマスターとワーカーノードを利用する形式の分散キーバリューストア（以下**分散 KVS**と呼ぶ）であるが、通常の分散 KVS が主にスケーラブルなウェブサービスのバックエンドとして動作するのに対し、**pfm** においては、分散 KVS のワーカーノードがそのままクライアントの接続を受け入れるフロントエンドとしても動作する。つまり、**pfm** 自体がフルスタックのアプリケーションフレームワークとなる。

また通常、分散 KVS が予め定義されたいくつかのコマンドでクライアントと通信するのに対し、**pfm** では、クライアントとワーカーノード、あるいはワーカーノードとワーカーノードの間で、バイナリシリアライズ形式を用いたイベント駆動型の **batched RPC**[a]でメッセージを送受信する。**pfm** にはスクリプトエンジンが埋め込まれており、**pfm** を用いる開発者（以下単に**開発者**と呼ぶ）は軽量のスクリプト言語を用いて受信した **RPC** の実際の動作を記述する。この際に、開発者は分散透過な形でその記述を行うことができる。上でプログラマブルな分散キーバリューストアと呼んでいるのはこのような特徴を指している。

さらに **pfm** のワーカーノード1つにつきサーバープロセスは1つしか存在しないが、このプロセス上で複数のオンラインゲームのサーバープログラムを動作させることができる。同じプロセス上の異なるオンラインゲームのプログラムにおいて、名前空間のアイソレーションが提供されているため、1ノードのみでもセキュリティ上問題なく複数のオンラインゲームをホスティングすることが可能である。

このアーキテクチャーによって以下のようなことが可能となる。

(1) 学習曲線が小さく、分散コンピューティングやマルチコアプログラミングについての特別なプログラムの知識がなくても開発が可能。

a) **RPC** の実装において、複数のリクエストをまとめて一度に送信し、非同期的に処理をすることで遅延を隠蔽する手法

(2) PaaS として動作させることができ、新しいオンラインゲームの開発に必要なインフラへの初期投資や運用時の追加投資を大きく削減できる。

これらが上記の問題を解決していることは明らかであろう。

本稿ではまず2章でオンラインゲームのネットワークサービスとしての性能要求について述べ、現在の一般的なウェブアプリケーションのフレームワークではその要求に対応することが難しいことを示す。3章ではその要求に対応し、さらに開発と運営のコストを大幅に削減する提案手法である **pfm** の基本となる着想について説明する。4章では **pfm** が PaaS として動作するために必要な機能について考察する。5章では **pfm** の構成要素と、その機能要件について論じる。6章では **pfm** を実装したフレームワーク **yue** の詳細について説明する。7章では6章で説明した **yue** を例にあげ、プレイヤーのログインの動作を順を追って説明する。8章では **pfm** の処理能力に関する性能評価のためのテストについて説明する。9章ではまとめと今後の課題について考察する。

2. オンラインゲームのネットワークサービスとしての性能要求

オンラインゲームの特殊な点は、それがゲームアプリケーションでもあるということである。ゲームアプリケーションは通常のアプリケーションと比較し、プレイヤーの入力に対して、極めて迅速にインタラクションを発生させること、またアプリケーションで管理されるデータ間の依存関係が深いことを特徴とする。[b]

こういった理由から、現在の一般的なウェブサービスを比較して以下のような特性があると言える。

(1) 更新の通知がクライアントに伝搬するまでの遅延が小さい

ゲーム中の状態の更新は、一般的に発生した後 100~200ms 程度で必要なクライアントに対して通知される。サービスがゲームとして成立するためにはプレイヤーの行動とそれによって発するインタラクションにラグを感じさせてはいけないためである。

[c]

(2) クライアントからのリクエスト頻度が高い

オンラインゲームの場合、3-10qps/client 程度が必要となる。ゲームに参加しているプレイヤー間で同期を取るため、クライアントでの操作の内容が迅速にサーバーに伝達される必要があるためである。

(3) クライアント・サーバー間の通信パケットサイズが小さい

b) 例えば、同じ部屋にいるプレイヤーの数によりプレイヤーのゲーム中での移動速度が変わるなど、ゲームではデータ同士に意図的にさまざまな参照関係をつけてルールを作っていくことが多いため。

c) 一般的に、人間が2つの事象が別のタイミングで発生していると感じる時間間隔の閾値が 100ms であると言われている。したがって、遅延が 100ms よりも大きくなる場合は基本的にプレイヤーからはラグと捉えられるため、その遅延を隠蔽するようなプログラム上の工夫が必要となる。200ms を超えるようであると、そういったプログラム上の工夫を行っても遅延を感じさせないことは難しくなる。

プレイヤーのゲームの操作に関連する通信パケットはほとんどが数十 byte である。大量のプレイヤーが周囲に存在する場合、最悪数百というオブジェクトの状態更新が一度に通知されたりするからである。

(4) 高いリクエストへの応答速度が要求される

インターネットの遅延が 50ms から 100ms 程度であることを考慮に入れると、各リクエストは最悪でも数十 ms で処理される必要がある。一般的には 1ms 未満である。

(5) ゲームが管理するデータの更新や参照の頻度が高い

ゲームに参加するプレイヤーのデータ等は数万レコード存在し、更新や参照が秒間数十回におよぶこともある。データの依存関係の高さにより 1 つのデータがさまざまなリクエストをトリガーとして更新・参照されるためである。

これらの特性から、通信プロトコルとして http を利用することが難しいことがわかる。更新を取得するために http 接続を確立するための通信で数百 ms を消費してしまうため、(1)で要求されている遅延時間での状態の更新の通知が間に合わないからである。仮に keepalive を利用したとしても、http の仕様から、一本のコネクションでは一度に 1 つのリクエストを送信するしか無いため、(2)の要求を満たすほどのスループットを得ることができない。1 クライアントあたり複数本の keepalive コネクションを張り続けるのはリソースの占有量を考えると現実的ではない。

さらに(3)の要求から、そもそもテキストプロトコルを使うべきではないことがわかる。通信パケットのサイズを小さくするため、各バイトは出来る限り効率的に利用されるべきであるし、さらにここで述べているほどの頻度で通信が発生する場合、テキストプロトコルのパースに関わる処理時間が無視できないほど大きくなるからである。[d]

以上から、通信プロトコルとして http を使うことができないといえる。http は現在のウェブアプリケーションのフレームワークでは唯一と言っていい選択肢であるため、オンラインゲームのプラットフォームとしてウェブアプリケーションの既存のフレームワークを使うことは難しい。

データの永続化についても、既存のフレームワークを利用した上で(4),(5)の要求を満たすことは困難である。データの更新が発生した瞬間に逐次 MySQL のようなディスクベースの RDBMS に書き込むといった、ウェブアプリケーションでよく行われるやり方では特に更新処理のパフォーマンスが不足してしまうためである。大量の DB サーバーを用意し sharding を行えば要求を満たすことも可能であると考えられるが、それはインフラへの投資を大幅に増加させ、今回の提案の目的を満たせなくなってしまうであろう。

以上から現在のウェブアプリケーションフレームワークによって、現状のオンライ

ンゲームの、特に開発コストの部分の問題を解消することは、以下の点で必要とされる性能要求を満たすという見地からみて困難であると考えられる。

- 通信の頻度および遅延時間
- 永続化されるデータの更新頻度

3. Pfm の基本的着想

3.1 性能要求の保証

本稿で提案する手法は 2 章で考察した通信およびデータ更新に対する性能要求を満たしつつ、1 章の(1)~(2)の利点を与えなければならない。本研究ではインフラのコストの削減(1 章(2))の必要から PaaS を利用しようとしている。そこで本節ではまず性能要求の保証のため、オンラインゲームのためとして十分な性能で PaaS として動作するソフトウェアとして適切な方式について考察した。

まず、十分なデータ更新性能を得るため、少なくとも更新の際には、メモリ上のデータのみを更新し、永続化のためのディスクへの書き込みを数秒~数十秒に一回のように lazy にするべきであると考えた。しかし、この手法をとった場合、障害発生時にその時間分の更新内容が失われてしまう。

この問題に対し、複数ノードのメモリ上にあるデータのコピーを保持しておきレプリケーションを行うことで、障害が発生しても更新内容が失われないようにするという手法を考えた。

この考察から、データを複数ノードに分散して保持し、耐障害性を与えるシステムとして、すでに多数の信頼ある実装をもつマスター・ワーカーノード方式の分散 KVS を、pfm の基本的なアーキテクチャーとして採用した。データ分散のアルゴリズムは consistent hash を利用し、永続化のためのディスクへの書き込みは lazy に行われることとする。これによりオンラインゲームに必要な頻度での更新を可能にしつつ、PaaS として動作するために必要な耐障害性とスケーラビリティを得ることができであろう。

次に、クライアントサーバー間の通信に対する性能要求について考察し、サーバーと persistent な接続を張った上で、バイナリシリアライズ形式を使い、http のような同期的な通信ではなく、イベント駆動型 IO により複数リクエストをまとめて送信し、非同期にレスポンスを受け取る方式を利用することとした。この方式は一般的なオンラインゲームで広く用いられており実績があるため十分なパフォーマンスが得られると考えられる。

この場合、pfm の何らかのサーバープロセスが persistent な接続を受け付けることになる。そのプロセスを pfm の分散 KVS 部分とは別に用意することも考えられたが、余分なフロントエンドをなくすことでリクエストへの応答速度を向上させ、必要なノ

d) 例えば memcached でも同様の問題からバイナリプロトコルが後に開発された。

ードの数を減らす目的から、分散 KVS のワーカノードが直接クライアントからの接続を受け付けることとした。

また 3.3 節で詳述するように、pfm では RPC を多用することになるが、通常よく行われているように、RPC のリクエストごとに OS のスレッドを割り当てる手法を採用することは難しい。通常のオンラインゲームでは persistent な接続を受け付ける 1 ノードあたり数百～千程度のコネクションを張ることが多く、同時に数百のリクエストを処理する必要があるからである。pfm のシステムから明示的にタイムスライスを割り当てて擬似的なスレッディングを提供するユーザーレベルスレッド方式を利用する必要があると考えた。

ここまでの考察で得られたアーキテクチャーによって、2 章で議論した、オンラインゲームとして動作するために必要な性能に対する要求はすべて満たされうると考えられる。この章の当初に述べたようにインフラのコスト削減（1 章(2)）は PaaS として動作することによって与えられる。したがって残るのは開発のコスト削減（1 章(1)）である。そのためには、オンラインゲームのプログラミングを専門的な知識が無くても行えるようにしなくてはならない。

3.2 開発のためのプログラミング難度の軽減

オンラインゲームにおいて、ゲーム内容を作成するレベルのプログラミングにおいても、実は専門的な知識が必要であることは多い。

例えば、通常のオンラインゲームでは、複数のプレイヤーが連携して行動することを助けるように、プレイヤーがチームを結成する処理が実装されている。そのような処理が提供する機能の中にはチームを構成するプレイヤーが長時間他のプレイヤーからの呼びかけに対して無反応な状態になった場合に、他のプレイヤーによるタイムアウト付きの多数決によってそのプレイヤーをチームから強制的に離脱させる、といったものがある。これは分散コンピューティングシステム上で定数ベースのコミットプロトコルを実装するのにほぼ等しく、さまざまな状況下で正しく動作するようにこういった処理を作成するためには、学習や経験から来る知識が必要である。

オンラインゲームの開発の難しさというものは、細かなゲームの仕様の中で上記のような複雑な分散コンピューティングのアルゴリズムの実装が要求されているため、個々の開発者がそういったアルゴリズムの実装とゲームとしての処理を作成しなくてはならないという点にあると言える。

もちろん、個別の分散コンピューティングのアルゴリズムは再利用可能な形にすることで、再発明をする必要がないようにはできるであろう。しかし新しいアルゴリズムが必要になった場合には、最初の実装をする開発者が依然として必要である。実装を再利用可能にする必要があれば、なおさらその開発者に求められるスキルは高くなり、やはり高コストの人材が必要ということになるだろう。本研究では専門知識のない開発者でも、少なくとも上の例で挙げたような程度の処理は作成できるようにした

いと考えた。

そのためには、多数のノードに分散して存在するオブジェクト群に対する一連の処理を分散処理を全く意識せず記述できるようにする必要がある。

3.3 RPC による分散透過な開発環境

分散処理をプログラム側から隠蔽する手法として RPC がある。pfm では、分散 KVS の上にスクリプト言語のプログラミング環境を用意し、その裏で RPC を実行する機能を与えることで、通信やリクエストのタイムアウトといった部分を隠蔽し、ゲーム内容を作成する側からは分散透過な形ですべての処理を記述させようと考えた。通常の RPC のライブラリでは、通信のための初期化処理や、受付可能なメソッドの定義などを開発者が記述する必要があるが、本研究では RPC を行うことが可能な対象を制限し、以下のような方法で、RPC を完全に隠蔽する。

- (1) 一般性を失わず、オンラインゲームのプログラムモデルはオブジェクト指向によって与えられると考えて良い。そのように考えてオンラインゲームをデザインしたときに、ゲームの構成要素となるデータ構造を永続化のため分散 KVS 上のレコードとして保存したものを以下単にオブジェクトと呼ぶ。
- (2) ゲーム内容を作成する開発者が処理を記述するためのスクリプト言語（以下単にスクリプトと呼ぶ）を用いて、オブジェクトに自身のデータを操作するための関数（メソッドと呼ぶ）と、オブジェクトを作成するための関数（コンストラクタと呼ぶ）を定義することができる。このメソッドとコンストラクタのみが RPC として呼び出される。
- (3) オブジェクトは他のオブジェクトのキーをデータとしてもつことができる。これはゲーム内におけるデータのリンク関係を表す。例えば下記図 1 のように、あるメソッドの呼出が、コンストラクタを呼び出してオブジェクトを生成した場合に、その戻り値を保持するとそういったデータをもつことになるだろう。
- (4) メソッドの中では、メソッドを呼び出されたオブジェクトが持つデータにアクセス出来る。また、上記のようにオブジェクトのキーをデータとしてもつ場合は、そのデータはスクリプトの中ではオブジェクトとして扱われる。
- (5) メソッド中で参照される自身とは別のオブジェクトについては、そのデータに直接アクセスすることはできず、メソッドの呼び出しだけが許される。このときメソッド呼び出しが行われたオブジェクトが同じワーカノードに存在していない場合は、メソッド呼び出しが自動的に RPC に変換される。

(6) pfm は consistent hash でオブジェクトを分散させる分散 KVS でもあるため、RPC のリクエストを送信すべきノードはオブジェクトのキーから consistent hash を使って計算する。

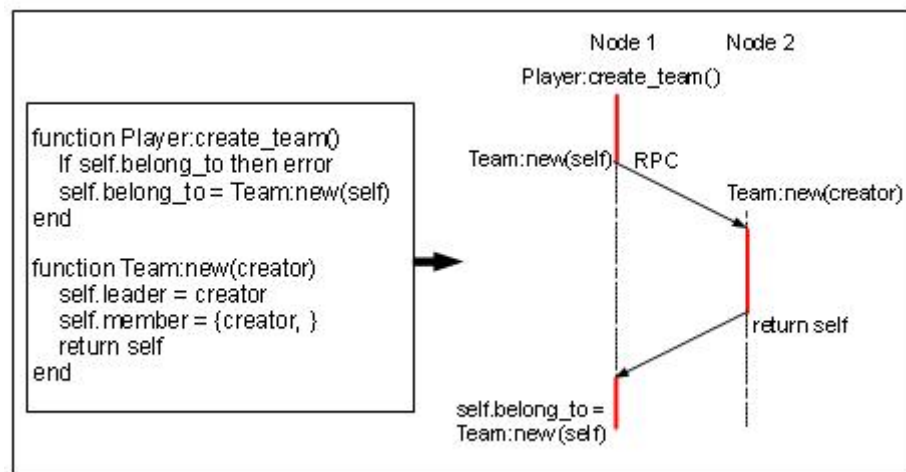


図 1 pfm におけるオブジェクト生成時の RPC とプログラムの関係のイメージ図

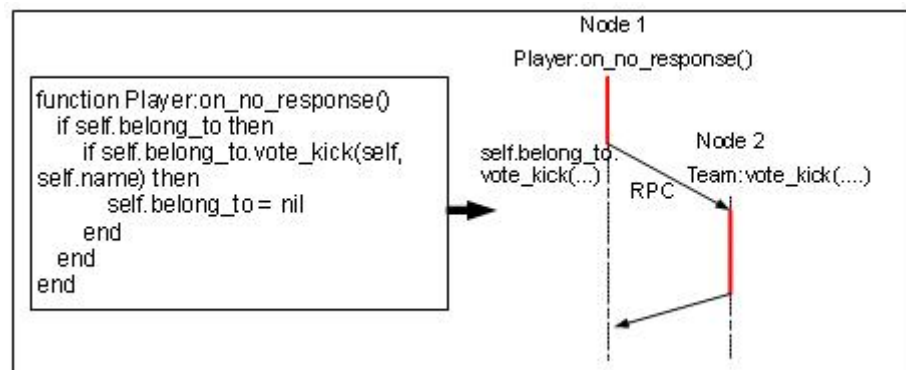


図 2 pfm におけるメソッド呼び出し時の RPC とプログラムの関係のイメージ図

例として、3.2 節で述べたチーム作成機能について考えてみる。簡略化されたコー

ドは図 1 の左の枠内のようなものである。まずチームを作成するには、プレイヤーのうち誰かが Player オブジェクトのメソッドである Player:create_team()を呼び出す。このメソッドは内部で Team オブジェクトのコンストラクタ Team:new(self)を呼び出している。この際に Team オブジェクトのキー値が決定し、そのキー値から Team オブジェクトが作成される Node2 が consistent hash によって計算され、そのノードに対してオブジェクト作成の RPC が行われることになる。RPC が完了すると、結果として作成されたオブジェクトが Node1 にレスポンスとして返され、belong_to という名前のデータとして保持される。

図 2 で示されているように、チームからプレイヤーを強制離脱させる処理も同様で、チーム作成時に初期化された self.belong_to はスクリプト中では Team オブジェクトとして扱われるため、self.belong_to.vote_kick(...)という形で Team オブジェクトに定義された多数決のためのメソッド vote_kick を呼び出すことができる。この場合、belong_to が実際に保持されているノードを belong_to のオブジェクトとしてのキー値から consistent hash を用いて計算し、そのノードに対して vote_kick を呼び出す RPC を発行する。

これらの例からも、開発者が全く分散処理を前提としないコードを記述 するだけで、内部的には自動的に分散処理が行われることがわかる。

4. pfm に必要な PaaS 機能についての考察

4.1 ノードの管理

pfm では管理の手間を省くためワーカーノードがマスターノードを UDP マルチキャストで検出して接続する方式をとる。しかし、オンラインゲームごとに使用できるノードについては、管理者がマスターノードにリクエストを送って手動で指定するようにした。オンラインゲームでは運営上の理由から、わざと「混んでいる (=人気がある)」ように見せるために、ノードの数をコントロールする必要があるからである。

あるオンラインゲームの処理となる RPC を定義するプログラムは、ノードがそのゲームのためのノードとして追加されたときにそのノードのスクリプト処理系にロードされることとした。これをゲームの初期化と呼ぶ。

またオンラインゲームにおいては、他のプレイヤーやゲームの内容を共有する必要から、自分の周囲に存在するプレイヤーや建物などのオブジェクトを知る必要がある。そこで、ゲームの初期化の前にならず 1 つの分散 KVS 上のオブジェクトが作成され、そのロードされたスクリプト処理系からグローバルな変数として名前を与えられるようにした。プレイヤーやオブジェクトはロードされたとき自身をこのグローバル変数に設定することでそういった情報の共有を行う。

4.2 RPC における名前空間のアイソレーション

pfm のサーバーは1 ノードあたり1 プロセスだけ動作する。これはプロセスのコンテキストスイッチを起きにくくすることで、persistent なコネクションを効率よく処理すること、また1 プロセスであればメモリ上に展開するべきスクリプトの処理系の数が少なくできることなどが理由である。しかし、PaaS として動作するためにはこの1 つのプロセス上で複数のオンラインゲームが動作できる必要がある。この特性のために同じスクリプトの処理系で動作していて、プログラム上同じ名前を指している異なる実体を参照するように出来る必要があると考えた。

6 章で紹介する pfm の実装では、この要求を、関数呼び出しごとにその関数のスコープで利用する名前空間を切り替えるというプログラム言語の機能 によって実現している。RPC を実行するときに、どのゲームのクライアントが発行した RPC なのかというのをリクエストに含めてもらうことで、そのゲームに対応した名前空間を指定した上で RPC を実行することで、アイソレーションを提供しようと考えた。

今回の実装では lua[1] というプログラム言語を利用するが、lua では環境テーブルという仕組みがこの機能を提供している。

4.3 認証機能及び KVS のキーの生成機能

3.1 節で説明したように、pfm は分散 KVS であるが、直接クライアントと接続しリクエストを受け付ける。このため、pfm のベースとなる分散 KVS は自分自身で保存されるデータのための一意なキーを生成できる必要がある。これについては各ノードがもつ mac address を利用してそういったキーを生成しようと考えた。

さらに分散 KVS 自体がクライアントからの接続を受け入れるため認証能力を持つ必要があるが、pfm 上で動作するゲームはそれぞれ異なった認証システムをもつ可能性があるため、認証処理については、pfm 上で動作するアプリケーション側で作成してもらうことを想定している。ただし、ゲームをプレイするプレイヤー自体もゲーム中のオブジェクトであるから、どこかのタイミングでゲーム中にロードされる必要があるだろう。これについては認証完了後に pfm がアカウントと対応するプレイヤーのオブジェクトをロードする。

5. pfm の基本構成要素と機能要件

pfm の基本構成要素は図 3 に示すように Nio, Connector, Serializer, DBM, UUID, Fiber, LL, Object, World, である。それぞれの役割と要求される機能要件を以下にまとめる。

(1) Nio

libev や後述する nbr のように Linux や Windows などの汎用 OS が提供する、ファイルディスクリプタのイベント通知機能を利用してマルチスレッドでイベント駆動 IO を提供するモジュールである。Nio が提供するコネクションを利用して pfm は非同期 RPC

のための通信を行う。また、1 章で述べたように、pfm は batched RPC をサポートするため、複数のリクエストを同時に送受信する機能をサポートしている必要がある。さらに UDP マルチキャストをサポートしている必要がある。

(2) DBM

Barkley Db や tokyocabinet のようなキーバリュ形式の DBM を提供するモジュール。データの永続化機能を提供する。

(3) Connector

Nio が提供する接続を管理するモジュール。各ノードへのコネクションの数を管理したり、障害発生時にコマンドのリクエスト先をバックアップノードに自動で切り替えたりといった処理を行う。

(4) Serializer

BISON や MessagePack のような RPC のプロトコルのためのバイナリシリアライズ形式を提供するモジュール。永続化の際もこのフォーマットに従ってデータがバックされる。

(5) UUID

4.3 節で説明したような、保存されるオブジェクトのキーとして利用出来る一意なデータを生成するためのモジュール。現在の実装では mac address を利用してローカルでユニークキーを生成する仕組みが使われている。

(6) Fiber

3.1 節で説明したように、pfm においては RPC リクエストの処理にユーザーレベルスレッドを利用する必要がある。Fiber はユーザーレベルスレッドを提供するためのモジュールである。それに加え、3.3 節で述べたように、pfm では RPC の実行中に実行されているメソッドが別の RPC を実行することがありうる。その場合、新しい RPC を実行したメソッドは自身が実行した RPC の完了まで実行を停止される必要がある。つまり任意のタイミングで実行権を明示的に移譲できる機能も必要である。

(7) LL

開発者に対して、スクリプトによる処理を記述できるようにするためのモジュール。Lua, ruby, squirrel のような、軽量の動的言語が適していると考えられる。

(8) Object

LL においてオブジェクトとして扱われる実体を管理するためのモジュール。LL の処理で発生したデータの更新を DBM に対して保存することも行う。

(9) World

pfm で動作するオンラインゲームを抽象化したモジュール。対応するオンラインゲームの実行環境として利用可能なノードのアドレスのリストを保持し、そのリストから consistent hash の計算を行う。

6. pfm の実装例

pfm のアーキテクチャーにおいて, Serializer として Msgpack[2], DBM として tokyocabinet[3], LL として lua, Nio として libnbr.a を利用して実装を行ったフレームワーク yue の実装について説明する. yue の実装は, linux 上でされており, 分散 KVS のマスターサーバープログラム yuem, ワーカーサーバープログラム yues, クライアントライブラリ libyuec.a, テスト用ロボット yuec からなる.

6.1 libnbr.a

libnbr.a はオンラインゲームに特有の通信パターンに特化して作成された自作の通信ライブラリであり, 以下のような特徴がある.

- epoll を利用した, マルチスレッドによるイベント駆動アーキテクチャにより多数の persistent なコネクションを張っているクライアントからの高頻度のリクエストを効率よく処理する.
- 基本アーキテクチャーが送受信をまとめて行う方式を取っているため, 単純にパケットの送受信を行うだけで自然に batched RPC と同等の通信が行われる. これにより小さなサイズの多数のパケットを効率よく送受信できる.
- あるクライアントからのリクエストの結果として, 他多数のクライアントに通知を行うことが頻繁に起こるオンラインゲームの特性に対応して, 異なるスレッドが処理しているコネクションに対しても, 高速に送信データを書き込むことが出来る.

これらの工夫により 1000 本の TCP コネクションを張った状態で平均 32byte のパケットを秒間 100000 回以上送受信することが可能である.

6.2 yue の各モジュール

● yuem

pfm のマスターノードで動作するサーバープログラムである. ワーカーノードで動作する yues からの接続を受け入れ, 各ノードの死活監視および, 各オンラインゲームが利用するノードの管理を行う.

● yues

pfm のワーカーノードで動作するサーバープログラムである. クライアントの認証処理や, ゲーム中実行される RPC の処理を行う.

● libyuec.a

C++ から yue にアクセスするためのライブラリ.

● yuec

libyuec.a を利用して作成したパフォーマンス測定のためのロボットプログラム.

7. プレイヤーのログインまでの手順

7.1 システムの起動

まず yuem が起動する. これにより yuem が起動した物理ノードのクラスタにおいて, pfm のシステムが起動する準備が整う. 一台以上の yues が起動すると, それぞれの yues サーバーは UDP マルチキャストによって yuem が動作しているノードを探し接続をする.

7.2 yue で動作するオンラインゲームの初期化

4.1 節で述べたように, 一台以上の yuem と yues が動作しているクラスタでオンラインゲームが動作するように初期化する場合には yuem に対してそのオンラインゲームのためのノードの追加コマンドを発行する. この際, 追加されたノードは初期化されたオンラインゲームのためのスクリプトを読み込むことが出来る必要がある. スクリプトを読み込むためのパスはノードの追加コマンドと一緒に指定される. 初期化が完了すると, クライアントからの接続を受け入れる準備が完了する.

7.3 ログイン

以下のような手順でクライアントがログインする.

- (1) クライアントはワーカーノードの 1 つに接続しログイン RPC を実行する.
- (2) ログイン RPC はまず yuem に送信され, 現在クライアントが接続しているノードがそのクライアントに対応したオンラインゲームのためのノードとして登録されているかをチェックする. 登録されているのであれば, (3) が実行される. そうでないのであれば, (4) が実行される.
- (3) yuem は yues に成功とそのアカウントに対応したプレイヤーデータの KVS 上でのキーを返す. yues はログイン RPC で送られてきた情報を使い, そのゲームに対応した認証処理を実行する. 典型的な例としてはそのオンラインゲームのサイトに https で接続して得たワンタイムパスワードを認証サイトに送付する, などであろう. 認証に成功した場合はプレイヤーデータがロードないし作成され, ゲームが開始する. 失敗した場合はクライアントとの接続を切断する.
- (4) yuem は yues にクライアントに対応したオンラインゲームのためのノードのアドレスと失敗を返す. このアドレスはクライアントに返信され, クライアントは新しいアドレスに接続し(1)からやり直す.

8. システムの評価

yue が PaaS として動作するために必要な名前空間のアイソレーションの機能を持ち, オンラインゲームが要求するパフォーマンスを満たすことの評価を行った.

8.1 環境

Amazon EC2(TM) のスモールインスタンスを 10 ユニット用意し, 5 ユニットのサー

バー, 5 ユニットをクライアントとした。サーバー側 5 ユニットのうち 1 台は yues, yuem, 1 台は yues が動作する。クラスタの構成においてマスターノードのアドレスは手動で与えた。クライアント 5 ユニットは yuec が動作する。

8.2 テスト内容

Amazon EC2(TM)上で構成した pfm クラスタ上で, 2 つのオンラインゲーム A, B を初期化しておく。ゲーム A について, クライアントは 1 台あたり 1000 クライアントがサーバーへログインする。サーバーは 1 台あたり 1000 本のクライアントからの TCP コネクションを受け入れることになる。ログイン後, 各クライアントが 128byte のメッセージを送付する 100 回のチャットコマンドを実行完了するまでの時間とコマンドごとの処理完了までの所要時間を 10 回計測し, 最低値と最高値を除いた平均値を計測した。同時にゲーム B に 1 クライアントをログインさせておく。ゲーム A と B のスクリプトはメソッドや変数名として同じ名前を使うが異なる処理を行うようにして, 処理が正しく行われるか確認した。

8.3 結果と考察

まず, B の実行結果が正しいことより, ゲーム A と B の名前空間は正しくアイソレーションがなされていることが確認できた。実行完了するまでの時間から計測された 1 クライアントあたりの平均スループットは 6.23qps, コマンドの平均の実行完了までの遅延は 22.79ms であった。これは 2 章で考察して得られた条件を満たしている。通常オンラインゲームは 3000 から 5000 人程度のプレイヤーが状態を共有するため複数の物理ノードに接続し, チャットや自分の情報を周囲に通知する。したがってこの条件下で目的の通信の頻度と遅延を達成できたため, yue はオンラインゲームが要求するパフォーマンスを達成できていると言える。

9. まとめ

分散 KVS であり, その上でオブジェクトの相互作用を RPC を利用し, 分散透過な形で記述できるアーキテクチャー pfm を提案し, その一実装 yue を用いた性能評価を行った。結果としてそのアーキテクチャーを元にしたフレームワークがオンラインゲームとして十分なパフォーマンスで動作し, PaaS として利用できることが分かった。プログラム作業が十分簡単になっているかどうかは, 個人差もあるが, 分散処理のパラダイムを理解することなく分散処理が記述できることから, 少なくとも従来よりも相当に簡単になっていると言えるだろう。

遅延は一般的なオンラインゲームの 1 リクエストの処理時間より相当長くなっているが, これは分散透過にするための内部での RPC のためである。つまり, 遅延をある程度ゲームプレイに問題ない程度まで大きくすることで, トレードオフとしてプログラミングを簡便にしていると言える。実際インターネットの通信遅延と比較すると,

1ms と 20ms の差はゲームをプレイする側には有意な差として認められないであろう。

以上から, pfm のアーキテクチャーにそって実装されたフレームワークを利用することによって, インフラ・開発両面のコストを削減し, オンラインゲームとして十分なパフォーマンスを提供しつつ開発を行うことができることが示された。

しかし, 今回簡単になったのはオンラインゲームのサーバー側のプログラムでしかない。通常のゲームであれば, クライアント側をスクリプティングや UI の操作だけで簡単に作成できるようなフレームワークが存在する。そういったフレームワークも lua を利用していることが多いため, yue はクライアント側のフレームワークと連携することが比較的容易である。クライアントの開発フレームワークとの連携でさらに開発のコストを下げるのが将来の発展として考えられる。

参考文献

- 1) バイナリシリアライズ形式 msgpack, <http://msgpack.org>
- 2) DBM 実装 tokyocabinet, <http://1978th.net/tokyocabinet/>
- 3) プログラム言語 lua, <http://www.lua.org>