プログラミング 8－10
（1996. 8. 28）

チャネルを同期制御に用いるコンカレント言語 Joule

辻村仁志　片山善夫　高橋真　熊谷章
hitosi@pfu.co.jp
（株）PFU　研究所

分散アプリケーションを記述するのに有効なモデルとして、チャネルを同期制御に用いるコンカレント言語 Joule を提案する。チャネルは、制御用ポートを持つメッセージキューであり、データフローモデルに基いてメッセージの同期と配送制御を行う。チャネルは、シンプルで柔軟なメッセージ配送モデルを提供し、分散オブジェクトの透過性を表現する。本稿では、 Joule 言語を紹介し、構成要素と基本概念を説明した後、チャネルとチャネルを用いたメッセージの配送、配管方法を示し、メッセージの配送経路の指定により、並列プログラムを記述できることを示す。また、 Joule 処理系の現状と実装方式についても報告する。

Joule: The Concurrent Programming Language with the Channel Synchronization Mechanism

Hitoshi Tsujimura, Yoshio Katayama, Makoto Takahashi and Akira Kumagai
Research Center, PFU Limited

We propose the concurrent programming language Joule which uses channels for synchronization. A channel is a message queue with a control port, and provides the control mechanism for synchronization and distribution based on a dataflow model. Channels provide a simple and flexible message forwarding model which represents the transparency among distributed objects. In this paper, we introduce Joule programming language and its basic concepts. We explain the components of Joule system, "message plumbing" by channels, and the way of parallel programming by describing message delivery pass. We also report the current status of Joule implementation.

1 はじめに

WWW(WorldWideWeb)の発達で、ネットワーク上に分散したドキュメントの結合は容易になった。しかし、分散アプリケーションの実現には、言語環境の記述能力を高め、アプリケーションで通信制御をしなくても済むようにする必要がある。シーケンシャル言語に1対1ベースの抽象化された通信機能を追加するアプローチもある[1]。しかし、多数の分散オブジェクトを利用するアプリケーション（例えば、ネットワークエージェントやサイバースペース）の記述には、コンカレントなオブジェクト指向言語を用いるのが自然であると考える。（図1）

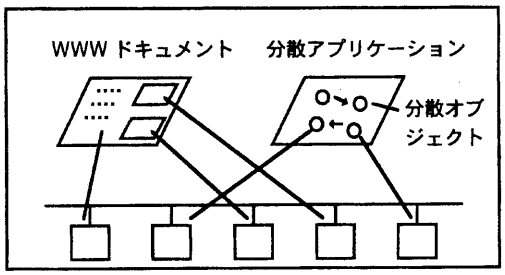

図1　WWWと分散アプリケーション

そこで、我々はこのような観点からコンカレント言語 Joule の開発を行っている。本稿では、 Joule を紹介し、その特徴について述べた後、実装方法についても報告する。

2 Joule 言語

Joule[2] は、米国Agorics社[2] とPFU で開発中の、アクターモデル[3]　に基づくコンカレント言語である。言語仕様の設計は、Dean Tribble による。 Joule の特徴は、データフローに従いメッセージの同期をとる「チャネル」と、チャネルを用いたシンプルで柔軟なメッセージ配送のモデルである。ここでは、Joule の構成要素と基本概念を説明した後、チャネルとそれを利用したプログラミングについて述べる。

2.1 サーバ

Jouleでは、すべてのオブジェクトをサーバと呼ぶ。数値や文字のようなプリミティブから、Joule プログラムを実行する複合サーバ、複数のサーバを含むモジュールやアプリケーションまで、すべてをサーバと呼ぶ。サーバはインスタンスオブジェクトである。 Joule ではクラスとインスタンスの区別は存在しない。サーバは、動的に新しいサーバを生成することができる。Jouleの実行環境中には、複数のサーバが存在し、それらはお互いにメッセージを送信しあう（図2）。サーバは、メッセージを受信するとアクティブになり、サーバ毎に定義された受信処理を行う。サーバによるメッセージ送信は非同期であり、メッセージは受信側のメッセージキューにキューイングされる。このため、すべてのサーバは、非同期に動作する。

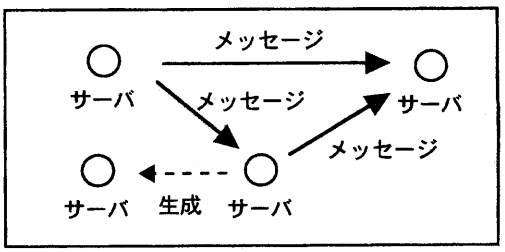

図2　サーバ

2.2 ファセットとポート

すべてのサーバは、1つ以上のファセットを持つ。ファセットは、メッセージの受信口である。 また、ファセットへの参照をポートと呼ぶ。 すべてのメッセージは、ポートへ送信される。ポートへ送信されたメッセージは、対応するファセットへ送信される。メッセージは、必ずポートとファセットを介して宛先サーバへ到達する。メッセージを、直接宛先サーバに届けることはできない。（図3）

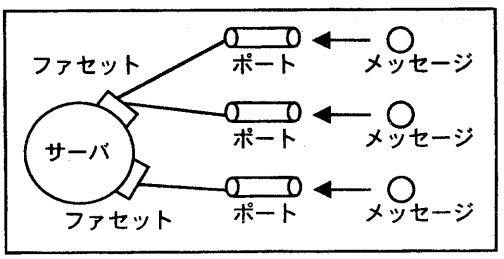

図3　ファセットとポート

サーバが複数のファセットを持つ場合、サーバは届いたメッセージをファセット毎に独立に解釈する。同じメッセージでも、どのファセットで受信するかにより、違う動作をさせることができる。ただし、サーバの内部状態（インスタンス変数）は、共通である。複数ファセットの例としては、public なファセットとprivate なファセットを持つサーバが挙げられる。

2.3 メッセージ

メッセージは、内部に1個以上のポートを含むJoule サーバである。メッセージ内には、オペレーション（＋などの演算子）を示すポートと0個以上の引数ポートを含む。（図4）

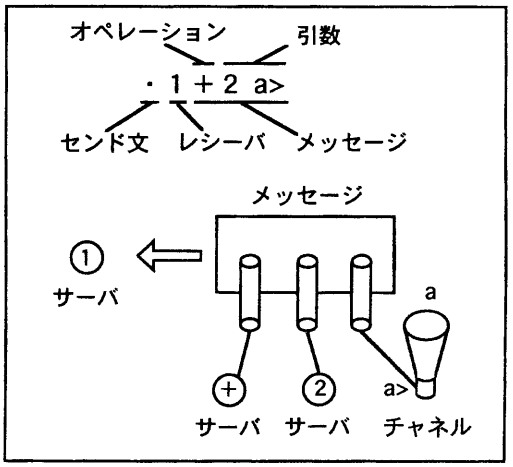

図4　メッセージ式の例

2.4 サーバの内部状態

サーバの内部には、0個以上のインスタンス変数と、ファセットから受信したメッセージを実行するメソッドがある（図5）。インスタンス変数は、他サーバのポートと結合する。メッセージ実行中に再結合（代入）が起こるサーバは、シリアライズサーバとなり、一時点では一つのメッセージを実行する。それ以外のサーバは、アンシリアライズサーバとなり、同時に複数のメッセージを実行できる。メッセージを受信すると、メッセージ中のオペレーションに従い適切なJoule メソッドを呼び出す。しかし、適切なメソッドがなく、かつサーバにotherwise メソッドの定義があれば、otherwise メソッドを実行する。otherwise メソッドでは、受け取ったメッセージをそのまま他サーバへ転送するなどの処理を行う。いずれの場合も、メソッド中では、Joule ステートメントを並列に実行する。

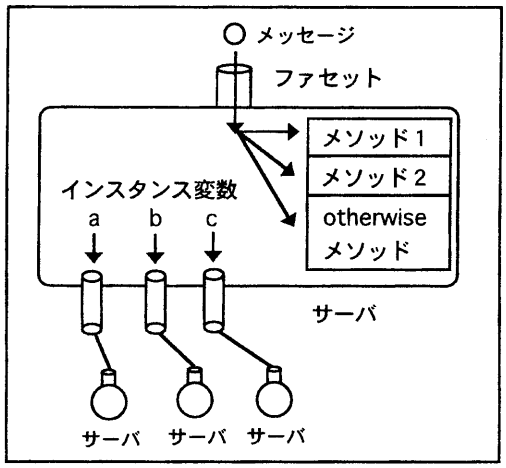

図5　サーバの内部状態

2.5 メッセージの受信処理

サーバにメッセージが到着しアクティブになると、次のいずれか、または両方の処理を行う。

- ・新しいサーバを作成する。
- ・アクセス可能なポートにメッセージ送信する。アクセス可能なポートとは、受信メッセージに含まれるポート、インスタンス変数、そのメソッドで新規作成したサーバへのポート、のいずれかである。

2.6 チャネル

チャネルは、アクセプタおよびディストリビュータと呼ぶ、2つのファセットを持つサーバである。チャネルAのアクセプタは「A」、ディストリビュータは「A>」と表記する。チャネルの機能は、アクセプタに届いたメッセージを、そのまま（解釈せずに）他のサーバへ回送することである。ディストリビュータは、どのサーバへ回送するかを指定するた

めの、制御用 のファセットである。チャネルは、ディストリビュータにフォワードメッセージが届くと、メッセージ引数で指定されたポートへのフォワードリンクを設定する。（図６）

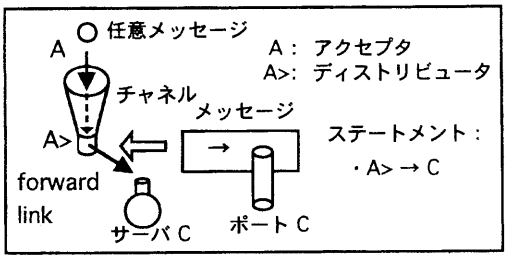

図６　チャネル

アクセプタにメッセージが届いた時、チャネルにフォワードリンクが設定済みであれば、メッセージはそのままフォワード先へ回送される。そうでなければ、メッセージはチャネル内にキューイングされ、ディストリビュータにフォワードメッセージが届いた時点で、フォワード先へと回送される。

Joule　では、チャネルを介した宛先ポートへのメッセージ送信と、チャネルを介せず宛先ポートへ直接送るメッセージ送信とを、等価とみなす。すなわち、チャネルはメッセージ送信に関して透過であり、送信者はチャネルの有無を区別できない（プリミティブサーバを除く）。（図７）

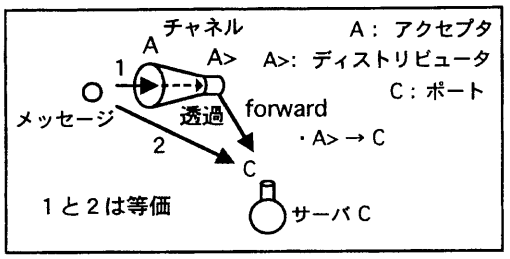

図７　チャネルの透過性

2.7 リビール

クライアントが、サーバに非同期メッセージを送って処理を依頼する場合、処理結果をどのようにして受け取るかという問題がある[4]。アクタでは、通常、依頼メッセージと一緒にコンティニュエーションアクタを送り、サーバは結果をコンティニュエーションに対しメッセージで通知する。これに対し、Joule では、クライアント（に相当する Joule サーバ）が、依頼メッセージと一緒にチャネルのディストリビュータポートを送り、サーバはディストリビュータに対し「結果値へフォワードせよ」というメッセージを送る。クライアント側では、チャネルの透過性により、アクセプタを結果値ポートとみなして処理を継続することができる。結果値をアクセスするには、アクセプタへメッセージを送信すれば良い。この、チャネルを利用した結果値への参照を、リターンに対応する用語として、リビールと呼ぶ。（図８）リビールに使用するディストリビュータポートは、慣用として、メッセージの最終引数にする。

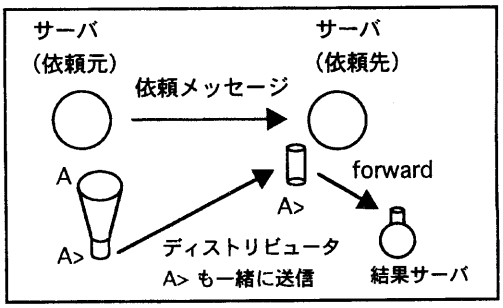

図８　リビール

2.8 メッセージの受信

リビールは、メッセージ引数にチェネルを使用する例であるが、チャネル自身をメッセージとして送ることもできる。メッセージの実体は Joule サーバであるが、サーバが実際に送受信するのは、メッセージサーバへのポートである。そのため、メッセージ本体の完成前にメッセージを送ることもできる。また、メッセージのたらい回し（デリゲーション）も効率良く行える。

2.9 式の展開

図９に、コンパイラが式を中間チャネルを用いた文に展開する例を示す。Def　文では、チャネル a を

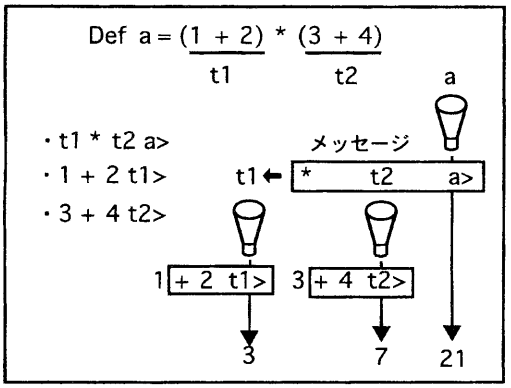

図９　式の展開

```
1: Server  Factorial :: num result>
2:  If num <= 1
3:       . result>  → 1
4:  else
5:       . num * (Factorial  :: num -  1) result>
6:  endIf
7: endServer
```

5: の展開形式
・num * t1 result>
・Factorial :: t2 t1>
・num - 1 t2>

図１０　階乗計算プログラム

作成し、計算結果21へフォワードさせる。コンパイラは、中間チャネル t1, t2 を作成し、この式を３つのメッセージ送信文に変換し、並列実行する。サーバ「1」は、メッセージ「+ 2 t1>」を受け取ると、サーバ「3」を作成し、チェネル t1 を 3 へフォワードさせる。同様に、サーバ「3」は、メッセージ「+ 4 t2>」を受けてサーバ「7」を作成し、チャネル t2 を 7 へフォワードさせる。一方、 t1 には、メッセージ「* t2 a>」が送られるが、このメッセージはチャネル t1 を通過して 3 へ到達する。サーバ「3」は、 t2 が 7 であることを知って 21 を作成し、チャネル a を 21 へフォワードさせる。言語仕様上、t2 は 7 と透過であり、t2 = 7 とみなせるため、値 7 の取得操作は必要ない。実装上は、プリミティブサーバが値の取得を行う。すなわち、サーバ「3」は、引数の加算値がプリミティブでなければ、チャネルをたどり値 7 を取り出す。

2.10 サンプルプログラム

図１０に、階乗計算プログラムの例を示す。1:Factorial サーバは、メッセージ「:: num result>」を受けると、num の階乗を計算し、result> ポートを「計算結果」 にフォワードする。2: num は、比較演算を行い、真なら 3: のステートメントを、偽なら 5: のステートメントを実行する。3: を実行すると、result> を 1 へフォワードする。5: は、中間チャネルを用いた３つの文に展開されるので、5: の実行では、これらの文が並列に実行される。そして、その中で Factorial へ再帰的にメッセージが送信される。

2.11 順序つきメッセージ

同一サーバに複数のメッセージを送信した場合、メッセージの到着順は非決定的である。しかし、プログラム上で順序を指定する必要のある場合もある（図１１）。

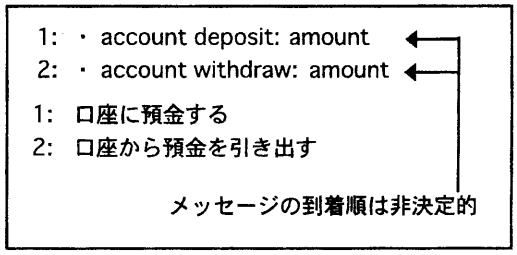

図１１　メッセージの到着順が重要な例

そのため、Joule では、２種類の順序つきメッセージを用意している。１つは、一つのステートメントで直列にメッセージ送信する方法（図１２）、もう１つは、チャネルを利用する方法である（図１３）。後者の例では、deposit: 送信と then: によるチャネルのフォワード設定には前後関係があるが、deposit: と withdraw: には前後関係がない。この例では、new-acount チャネルを、"deposit: メッセージ送信後の account サーバ" とみなすことができる。

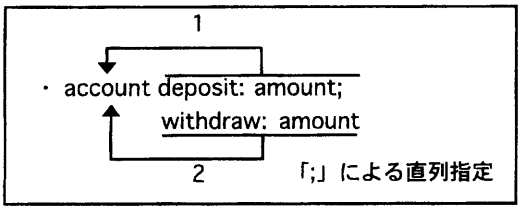

図１２　直列メッセージの例

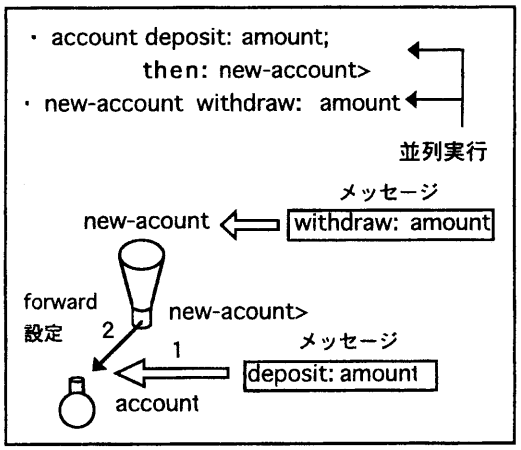

図１３　チャネルによる順序つきメッセージの例

3 実装

Joule 処理系の実装は、Smalltalk および C++ 言語を用いて行っている。 Smalltalk 版では、コンパイラ、ブラウザ、デバッガ、ランタイムを含む開発環境を、C++ 版ではランタイム環境を作成中であり、現在、サンプルプログムが動作している。実装においては、 Joule ソースコードを、マシン非依存な中間コード（バイトコード）にコンパイルし、それを仮想マシンで実行する方式を採用している。

3.1 モジュール

モジュールは、複数のサーバをその中に含む Joule サーバである。コンパイラは、モジュール単位にコンパイルし、バイトコードへ変換する。モジュールは、入れ子にすることができる。ローダは、コンパイルされたモジュールを実行環境へロードし、実行環境にあるデフォルトポートと結合する。その後、モジュールへ初期化メッセージを送る。モジュールは、初期化メッセージを受けると、モジュール内のサーバを生成する Joule コードを実行する。

3.2 JAM バイトコード

中間コードには、スタックモデルで動作するJAM (Joule Abstract Machine) バイトコード を使用する。これは、メッセージの配送を実現する約40種の命令セットであり、四則演算は含まない。演算処理は、すべてプリミティブサーバで行う。図１４に、JAM コードの例を示す。ここで、スタックに積むのはすべてポートであり、opOuter の引数はモジュールで共有するワーク用ポート配列のインデックス、opSeal の引数２はスタックから取り出すメッセージ引数の個数である。opOuter2Send では、スタックにある１にメッセージ mes を送っている。

バイトコード	スタックの状態	
opChannel;	⇒ a a>	チャネル生成
opOuter: 6;	⇒ a a> 2	２へのポートを積む
opOuter: 4;	⇒ a a> 2 +	+へのポートを積む
opSeal: 2;	⇒ a mes	メッセージ作成
opOuter: 7;	⇒ a mes 1	１へのポートを積む
opOuter2Send;	⇒ a	メッセージ送信

図１４　"・１＋２ a>" の一部

4 おわりに

コンカレント言語 Joule では、チャネルを用いたシンプルなモデルを用いて、メッセージの配送経路を指定することにより、並列プログラムを記述できることを示した。処理系もプロトタイプが動作中であり、言語としての整合性の検証も進んでいる。今後は、ネットワーク上に実行環境を実装し、並列処理における有効性について評価する予定である。

参考文献

[1] http://ring.etl.go.jp/openlab/horb/

[2] http://www.webcom.com/~agorics

[3] W. Kim, and F. H. Lochovsky, Object-Oriented Concepts, Databases, and Applications, ACM Press(1989)

[4] Scheme 過去・現在・未来, Guy L. Steele J (訳 井田昌之) bit 1996,4