(19)日本国特許庁(JP)　(12)公表特許公報(A)　(11)特許出願公表番号
特表2007-538323
(P2007-538323A)
(43)公表日　平成19年12月27日(2007.12.27)

(51)Int.Cl.	F I	テーマコード（参考）
G06F 9/46 (2006.01)	G06F 9/46 350	

審査請求 未請求　予備審査請求 未請求　（全 52 頁）

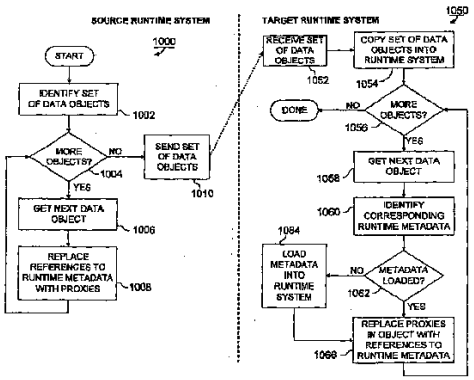
(21)出願番号	特願2007-517096 (P2007-517096)
(86)(22)出願日	平成17年5月20日 (2005.5.20)
(85)翻訳文提出日	平成19年1月22日 (2007.1.22)
(86)国際出願番号	PCT/EP2005/005502
(87)国際公開番号	WO2005/114405
(87)国際公開日	平成17年12月1日 (2005.12.1)
(31)優先権主張番号	10/851,813
(32)優先日	平成16年5月20日 (2004.5.20)
(33)優先権主張国	米国 (US)

(71)出願人　300015447
エスアーペー　アーゲー
ＳＡＰ　ＡＧ
ドイツ連邦共和国，６９１９０　バルドルフ，ディートマル－ホップ－アレー　１６
Ｄｉｅｔｍａｒ－Ｈｏｐｐ－Ａｌｌｅｅ　１６，　６９１９０　Ｗａｌｌｄｏｒｆ，　Ｇｅｒｍａｎｙ
(74)代理人　100064908
弁理士　志賀　正武
(74)代理人　100089037
弁理士　渡邊　隆
(74)代理人　100108453
弁理士　村山　靖彦

最終頁に続く

(54)【発明の名称】ランタイムシステムにおけるオブジェクトを共有するためのプログラム、方法、装置

(57)【要約】
ランタイムシステムにおけるオブジェクトを共有するためのコンピュータシステム及びプログラムプロダクトを含む方法及び装置。一組のオブジェクトが識別され、各オブジェクトは、第１ランタイムシステムにおけるオブジェクトに関するランタイムメタデータに対する参照を有する。各オブジェクトにおける上記ランタイムメタデータに対する参照はプロキシで置き換えられ、そして、上記一組のオブジェクトは、第２のランタイムシステムに伝送され、その第２のランタイムシステムでは、各オブジェクトにおけるプロキシが第２のランタイムシステムにおけるオブジェクトに関するランタイムメタデータに対する参照で置き換えられる。或る実施例では、ランタイムメタデータは、既に利用可能な状態になければ、第２のランタイムシステムにインストールされる。



【特許請求の範囲】
【請求項１】
データ処理装置に、
各データオブジェクトが第１のランタイムシステムにおけるデータオブジェクトに関するランタイムメタデータに対する参照を含んでなる一組のデータオブジェクトを識別させ、
前記一組のデータオブジェクトにおける各データオブジェクトについて、前記データオブジェクトにおける前記ランタイムメタデータに対する前記参照をプロキシで置き換えさせ、
前記一組のデータオブジェクトを第２のランタイムシステムに伝送させるように動作可能なマシン読み取り可能な記憶媒体に格納されたコンピュータプログラム。
【請求項２】
前記一組のデータオブジェクトにおける各データオブジェクトについて、前記データオブジェクトに関する前記ランタイムメタデータに対する前記参照は、前記データオブジェクトがインスタンスであるクラスのランタイム表現に対する参照を含む請求項１記載のコンピュータプログラム。
【請求項３】
前記第１のランタイムシステムは、第１の仮想マシンを含み、且つ、前記第２のランタイムシステムは、第２の仮想マシンを含む請求項１または２記載のコンピュータプログラム。
【請求項４】
前記第１の仮想マシンおよび前記第２の仮想マシンは、Java（登録商標）仮想マシンまたは共通言語ランタイム仮想マシンである請求項３記載のコンピュータプログラム。
【請求項５】
前記第１のランタイムシステムは、第１の物理マシン上に配置され、且つ前記第２のランタイムシステムは、第２の異なる物理マシン上に配置された請求項１ないし４の何れか１項記載のコンピュータプログラム。
【請求項６】
前記一組のデータオブジェクトを識別するステップは、１又は２以上のデータオブジェクトの推移的クロージャを識別するステップを含み、前記参照されたデータオブジェクトのそれぞれは、第１のデータオブジェクトによって参照される請求項１ないし５の何れか１項記載のコンピュータプログラム。
【請求項７】
データ処理装置に、
各データオブジェクトがプロキシを含んでなる一組のデータオブジェクトを第１のランタイムシステムから受信させ、
前記一組のデータオブジェクトを第２のランタイムシステムにコピーさせ、
且つ、
前記一組のデータオブジェクトにおける各データオブジェクトについて、
前記データオブジェクトに関するランタイムメタデータを識別させ、
前記データオブジェクトにおける前記プロキシを前記データオブジェクトに関する前記ランタイムメタデータに対する参照で置き換えさせるように動作可能なマシン読み取り可能な記憶媒体に格納されたコンピュータプログラム。
【請求項８】
前記一組のデータオブジェクトにおける各データオブジェクトについて、
前記データオブジェクトに関する前記ランタイムメタデータが前記第２のランタイムシステムにおいて利用可能かどうかを判定させ、
もし、前記データオブジェクトに関する前記ランタイムメタデータが前記第２のランタイムシステムにおいて利用可能でなければ、前記データオブジェクトに関する前記ランタイムメタデータを前記第２のランタイムシステムにインストールさせるように更に動作可

能な請求項７記載のコンピュータプログラム。
【請求項９】
前記一組のデータオブジェクトにおける各データオブジェクトについて、前記データオブジェクトに関する前記ランタイムメタデータに対する前記参照が、前記データオブジェクトがインスタンスであるクラスのランタイム表現に対する参照を含む請求項７または８記載のコンピュータプログラム。
【請求項１０】
前記命令が、前記データ処理装置に、
前記データオブジェクトがインスタンスである前記クラスの前記ランタイム表現が前記第２のランタイムシステムにおいて利用可能であるかどうかを判定させ、
もし、前記データオブジェクトがインスタンスである前記クラスの前記ランタイム表現が前記第２のランタイムシステムにおいて利用可能でなければ、前記データオブジェクトがインスタンスである前記クラスの前記ランタイム表現を前記第２のランタイムシステムにインストールさせるように更に動作可能な請求項９記載のコンピュータプログラム。
【請求項１１】
前記第１のランタイムシステムは、第１の仮想マシンを含み、且つ、前記第２のランタイムシステムは第２の仮想マシンを含む請求項７ないし１０の何れか１項記載のコンピュータプログラム。
【請求項１２】
前記第１の仮想マシンおよび前記第２の仮想マシンは、Java（登録商標）仮想マシンまたは共通言語ランタイム仮想マシンである請求項１１記載のコンピュータプログラム。
【請求項１３】
前記第１のランタイムシステムは、第１の物理マシン上に配置され、且つ前記第２のランタイムシステムは、第２の異なる物理マシン上に配置された請求項７ないし１２の何れか１項記載のコンピュータプログラム。
【請求項１４】
前記一組データオブジェクトは、第１のデータオブジェクトと１又は２以上の参照されたデータオブジェクトの推移的クロージャから構成され、前記参照されたデータオブジェクトのそれぞれは、前記第１のデータオブジェクトによって参照される請求項７ないし１３の何れか１項記載のコンピュータプログラム。
【請求項１５】
前記ランタイムメタデータを識別し、前記一組のデータオブジェクトにおける各データオブジェクトにおける前記プロキシを置き換えるためのオペレーションは、前記一組のデータオブジェクトが受信された後、実質的に即座に発生する請求項７ないし１４の何れか１項記載のコンピュータプログラム。
【請求項１６】
前記一組のデータオブジェクトにおける各データオブジェクトについて、前記ランタイムメタデータを識別し、前記データオブジェクトにおける前記プロキシを置き換えるためのオペレーションは、前記データオブジェクトが前記第２のランタイムシステムにおいて最初にアクセスされるときに発生する請求項７ないし１５記載のコンピュータプログラム。
【請求項１７】
各データオブジェクトが第１のランタイムシステムにおけるデータオブジェクトに関するランタイムメタデータに対する参照を含んでなる一組のデータオブジェクトを識別し、
前記一組のデータオブジェクトにおける各データオブジェクトについて、前記データオブジェクトにおける前記ランタイムメタデータに対する前記参照をプロキシで置き換え、
前記一組のデータオブジェクトを第２のランタイムシステムに伝送するコンピュータ実施の方法。
【請求項１８】
各データオブジェクトが第１のランタイムシステムにおけるデータオブジェクトに関す

るランタイムメタデータに対する参照を含んでなる一組のデータオブジェクトを識別するための手段と、
前記一組のデータオブジェクトにおける各データオブジェクトにおける前記ランタイムメタデータに対する前記参照をプロキシで置き換えるための手段と、
前記一組のデータオブジェクトを第２のランタイムシステムに伝送するための手段と、を備えた装置。
【請求項１９】
各データオブジェクトがプロキシを含んでなる一組のデータオブジェクトを第１のランタイムシステムから受信し、
前記一組のデータオブジェクトを第２のランタイムシステムにコピーし、
前記一組のデータオブジェクトにおける各データオブジェクトについて、
前記データオブジェクトに関するランタイムメタデータを識別し、
前記データオブジェクトにおける前記プロキシを前記データオブジェクトに関する前記ランタイムメタデータに対する参照で置き換えるコンピュータ実施の方法。
【請求項２０】
各データオブジェクトがプロキシを含んでなる一組のデータオブジェクトを第１のランタイムシステムから受信するための手段と、
前記一組のデータオブジェクトを第２のランタイムシステムにコピーするための手段と、
前記一組のデータオブジェクトにおける各データオブジェクトに関するランタイムメタデータを識別すると共に、各データオブジェクトにおける前記プロキシを前記データオブジェクトに関する前記ランタイムメタデータに対する参照で置き換えるための手段と、を備えた装置。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、データ処理に関する。
【背景技術】
【０００２】
一般に、エンタープライズサーバー或いは他の大型サーバーのような幾つかのサーバーは、概して、ユーザーセッションに属する一般には小さなユーザーリクエストを大量に処理するので、リクエスト処理エンジンと特徴づけられる。通常、リクエストの処理は、サーバー上で実行するランタイムシステム（例えばＪａｖａ（登録商標）仮想マシン）におけるユーザーコード（例えばＪａｖａ（登録商標）サーブレット又はエンタープライズＪａｖａ（登録商標）ビーンズ）の作動を起動する。このようなサーバーにおける拡張性(scalability)は、従来、スレッド(thread)を使用することにより達成され－例えば、多くのユーザーセッションに対応するリクエストを処理するためにマルチスレッドの仮想マシン（ＶＭ）が使用される。しかしながら、システムのロバスト性(robustness)は、ユーザーセッション間の強い分離を必要とし、単一のＶＭ内で多くのユーザーセッションが作動している場合、それを実現することは困難である。
【０００３】
オペレーティングシステムは、処理に対してはほぼ完全な分離を提供することができる。幾つかのオペレーティングシステムでは、クラッシュした処理(crashed process)は他の処理に影響を与えず、そして、割り当てられた資源を取り残すことやリークすることはないであろう。ユーザーセッションを分離することは概念的には可能であり、従って一つのオペレーティングシステム（ＯＳ）処理を各ユーザーセッションに割り当て、割り当てられた処理内でそのユーザーセッションのためのＶＭを作動させることにより、サーバーのロバスト性を向上させることは概念的には可能である。しかしながら、このようなアプローチは、相当に多数の処理間でのスイッチングにおいて発生するＯＳスケジューリングのオーバーヘッドのため、また、このようなアプローチが消費する資源のため、或る状況

（例えば多数のユーザーセッションが存在する状況）では実用的ではないかもしれない。一般に、ＯＳ処理は、ユーザーセッションのようにきめ細かくエンティティ(entity)を作るのに設計されていない。

【発明の開示】

【課題を解決するための手段】

【０００４】

本発明は、ユーザーセッション間の分離(isolation)を提供し、データを共有するための技術を実施するコンピュータプログラムプロダクトを含む方法及び装置を提供する。

【０００５】

一般的な一態様において、本技術は、データ処理装置に、第１のランタイムクラスのインスタンス(instance)である第１のデータオブジェクトの識別(identification)を受信させ、前記第１のランタイムクラスが共有可能であるかどうかを判定させ、前記第１のデータオブジェクトが１又は２以上の参照データオブジェクトを参照するかどうかを判定させるように操作可能なコンピュータプログラムプロダクトを特徴とする。もし、前記第１のデータオブジェクトが１又は２以上の参照データオブジェクトを参照すれば、前記コンピュータプログラムプロダクトは、さらに、前記データ処理装置に、前記１又は２以上の参照データオブジェクトをトラバース(traverse)させ、そして、トラバースされた各データオブジェクトについて、前記トラバースされたデータオブジェクトがインスタンスであるところのランタイムクラスが共有可能であるかどうかを判定させるように操作可能である。

【０００６】

有利な実施例は、以下の１又は２以上の特徴を含むことができる。前記１又は２以上の参照データオブジェクトをトラバースするステップは、前記１又は２以上の参照データオブジェクトの推移的クロージャ(transitive closure)における各データオブジェクトを再帰的にトラバースするステップを含むことができる。

【０００７】

もし、前記第１のランタイムクラスが共有可能であり、且つトラバースされた各データオブジェクトの前記ランタイムクラスが共有可能であれば、前記第１のデータオブジェクト及びトラバースされた各データオブジェクトは、オブジェクトグループにグループ化され、且つ、前記オブジェクトグループは、共有メモリ領域にコピーされることができる。

【０００８】

もし、前記第１のランタイムクラスが共有可能でなければ、または、少なくとも一つのトラバースされたデータオブジェクトの前記ランタイムクラスが共有可能でなければ、消極的ステータスの指示(indication)が生成される。

【０００９】

前記第１のランタイムクラスが共有可能であるかどうかを判定するステップは、前記第１のランタイムクラスが共有可能であることが前もって宣言されているかどうかを判定するステップを含むことができる。

【００１０】

前記第１のランタイムクラスは、１又は２以上のベースクラス(base class)から得ることができ、且つ１又は２以上のフィールド(field)を含むことができる。前記第１のランタイムクラスが共有可能であるかどうかを判定するステップは、前記第１のランタイムクラスが直列化インターフェイス(serialization interface)を実施するかどうかを判定するステップと、前記第１のランタイムクラスのオブジェクトインスタンスの直列化又は非直列化(deserialization)の期間にカスタムコードが実行されるかどうかを判定するステップと、全ての前記ベースクラスが直列化可能であるかどうかを判定するステップと、全ての前記フィールドが直列化されるかどうかを判定するステップと、前記第１のランタイムクラスの前記オブジェクトインスタンスがガーベッジコレクション(garbage collection)に影響を与えるかどうかを判定するステップとを含むことができる。

【００１１】

前記第１のランタイムクラスは、Ｊａｖａ（登録商標）クラスであってもよく、前記直列化インターフェイスは、java（登録商標）.io.Serializableであってもよい。

【００１２】

カスタムコードが実行されるかどうかを判定するステップは、前記第１のランタイムクラスが所定組のメソッドの中のメソッドを含んでいるかどうかを判定するステップを含むことができる。前記所定組のメソッドは、readObject( ),writeObject( ),readExternal( ),writeExternal( ),readResolve( ),writeReplace( )メソッドを含むことができる。

【００１３】

全ての前記ベースクラスが直列化可能であるかどうかを判定するステップは、前記ベースクラスの中の各クラスが前記直列化インターフェイスを実施するかどうかを判定するテップと、もし、前記ベースクラスの中のクラスが前記直列化インターフェイスを実施しなければ、前記クラスが普通のデフォルトのコンストラクター(constructor)を含むかどうかを判定するステップとを含むことができる。

【００１４】

全ての前記フィールドが直列化されるかどうかを判定するステップは、前記フィールドの何れもトランジェントフィールド(transient field)であるかどうかを判定するステップを含むことができる。また、全ての前記シールドが直列化されるかどうかを判定するステップは、前記フィールドの何れもserialPersistentFieldであるかどうかを判定するステップを含むこともできる。

【００１５】

前記第１のランタイムクラスのオブジェクトインスタンスがガーベッジコレクションに影響を与えるかどうかを判定するステップは、前記第１のランタイムクラスが普通の終了化子(finalizer)を含むかどうかを判定するステップを含むことができる。前記第１のランタイムクラスがＪａｖａ（登録商標）クラスであれば、前記第１のランタイムクラスのオブジェクトインスタンスがガーベッジコレクションに影響を与えるかどうかを判定するステップは、また、前記第１のランタイムクラスがjava（登録商標）.lang.ref.Referenceクラスから得られるかどうかを判定するステップを含むことができる。

【００１６】

前記第１のランタイムクラスが共有可能であるかどうかを判定するステップは、前記ランタイムクラスのランタイム表現が共有可能であるかどうかを判定するステップと、前記第１のランタイムクラスのためのクラスローダー(class loader)が共有可能であるかどうかを判定するステップとを更に含むことができる。

【００１７】

前記第１のランタイムクラスの前記ランタイム表現が共有可能かどうかを判定するステップは、前記ランタイム表現が第１の所定の場所に格納されるかどうかを判定するステップを含むことができる。前記第１のランタイムクラスのための前記クラスローダーが共有可能かどうかを判定するステップは、前記クラスローダーが第２の所定の場所に格納されているかどうかを判定するステップを含むことができる。

【００１８】

他の態様において、本技術は、データ処理装置に、ゼロ又はそれ以上の参照データオブジェクトを参照する第１のデータオブジェクトであって第１のランタイムシステムにおける該第１のデータオブジェクトの識別を受信させ、前記参照データオブジェクトの推移的クロージャ(transitive closure)及び前記第１のデータオブジェクトから構成されるデータオブジェクトの共有クロージャ(shared closure)を識別させ、前記データオブジェクトの共有クロージャが第２のランタイムシステムにおいて使用可能かどうかを判定させるように操作可能なコンピュータプログラムプロダクトを特徴とする。

【００１９】

有利な実施例は、次の１又は２以上の特徴を含むことができる。前記第１及び第２のランタイムシステムは、Ｊａｖａ（登録商標）仮想マシン又は共通言語仮想マシン(Common Language Runtime virtual machine)を含む仮想マシンであってもよい。前記データオブ

ジェクトの共有クロージャは、ユーザーコンテキスト情報(user context information)を含むことができる。

【００２０】

前記共有クロージャが前記第２のランタイムシステムにおいて使用可能であるかどうかを判定するステップは、前記データオブジェクトの共有クロージャにおける各データオブジェクトがカスタムコードを実行することなく直列化可能であるかどうかを判定するステップを含むことができる。

【００２１】

前記共有クロージャが前記第２のランタイムシステムにおいて使用可能であるかどうかを判定するステップは、前記共有クロージャにおける各データオブジェクトのランタイムクラスが共有可能であるかどうかを判定するステップを含むことができる。

【００２２】

各データオブジェクトがインスタンスであるところのランタイムクラスは、１又は２以上のベースクラスから得ることができ、１又は２以上のフィールドを有することができ、且つ、前記ランタイムクラスが共有可能であるかどうかを判定するステップは、前記ランタイムクラスが直列化インターフェイスを実施するかどうかを判定するステップと、前記ランタイムクラスのオブジェクトインスタンスの直列化又は非直列化の期間中にカスタムコードが実行されるかどうかを判定するステップと、全ての前記ベースクラスが直列化可能であるかどうかを判定するステップと、全ての前記フィールドが直列化されるかどうかを判定するステップと、前記ランタイムクラスのオブジェクトインスタンスがガーベッジコレクションに影響を与えるかどうかを判定するステップとを含むことができる。

【００２３】

消極的ステータス指示(negative status indication)は、前記データオブジェクトの共有クロージャが前記第２のランタイムシステムにおいて使用可能でなければ、生成されることができる。

【００２４】

前記データオブジェクトの共有クロージャは、共有メモリ領域にコピーされることができる。前記コンピュータプログラムプロダクトは、更に、前記データ処理装置に、前記データオブジェクトの前記共有クロージャの前のバージョンが前記共有メモリ領域に存在するかどうかを判定させ、且つ、バージョンナンバーを前記データオブジェクトの共有クロージャと関連づけさせるように操作可能であってもよい。

【００２５】

他の態様において、本技術は、データ処理装置に、識別子を受信させ、前記識別子と関連する共有クロージャを識別させ、前記共有クロージャをランタイムシステムと関連づけさせるように操作可能なコンピュータプログラムプロダクトを特徴とする。前記共有クロージャは、共有メモリ領域に配置され、且つ、第１のデータオブジェクトと該第１のデータオブジェクトによって参照されるデータオブジェクトの推移的クロージャから構成される。

【００２６】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記ランタイムシステムは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであってもよい。前記共有クロージャを識別するステップは、前記共有クロージャの現在のバージョンを識別するステップを含むことができる。

【００２７】

前記共有クロージャは、前記ランタイムシステムと関連するアドレス空間に前記共有クロージャをコピー又はマッピングすることにより、前記ラインタイムシステムと関連づけられることができる。前記共有クロージャが前記ランタイムシステムと関連するアドレス空間にマッピングされた場合、前記共有クロージャを前記ランタイムシステムに関連づけるステップは、前記共有クロージャに対するライトアクセスを阻止し、または、前記共有クロージャにおけるデータオブジェクトに対する第１のライトアクセスを検出すると、す

ぐに、前記ラインタイムシステムと関連する前記アドレス空間に前記共有クロージャをコピーするステップを更に含むことができる。

【００２８】

共有クロージャは、削除されたものとしてマークされることができる。削除されたものとして共有クロージャをマークするステップは、前記共有クロージャが追加的ランタイムシステムと関連づけられることを防ぐステップを含むことができる。

【００２９】

他の態様において、本技術は、データ処理装置に、一組のデータオブジェクトを識別させるように操作可能なコンピュータプログラムプロダクトを特徴とし、前記一組のデータオブジェクトの中の各データオブジェクトは、第１のランタイムシステムにおける前記データオブジェクトに関するランタイムメタデータに対する参照を有する。前記コンピュータプログラムプロダクトは、更に、前記データ処理装置に、各データオブジェクトにおける前記ランタイムメタデータに対する前記参照をプロキシで置き換えさせ、そして、前記一組のデータオブジェクトを第２のランタイムシステムに伝送させるように操作可能である。

【００３０】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記一組のデータオブジェクトの中の各データオブジェクトについて、前記データオブジェクトに関する前記ランタイムメタデータに対する参照は、前記データオブジェクトがインスタンスであるところのクラスのランタイム表現に対する参照を含むことができる。

【００３１】

前記第１のランタイムシステムは、第１の仮想マシンを含むことができ、前記第２のランタイムシステムは、第２の仮想マシンを含むことができる。前記第１及び第２の仮想マシンは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであることができる。前記第１のランタイムシステムは、第１の物理マシン上に配置されることができ、前記第２のランタイムシステムは、第２の異なる物理マシン上に配置されることができる。

【００３２】

前記一組のデータオブジェクトを識別するステップは、１又は２以上の参照データオブジェクトの推移的クロージャを識別するステップを含むことができ、各参照データオブジェクトは、第１のデータオブジェクトによって参照されるオブジェクトである。

【００３３】

他の態様において、本技術は、データ処理装置に、各データオブジェクトがプロキシを含んでなる一組のデータオブジェクトを第１のランタイムシステムから受信させ、前記一組のデータオブジェクトを第２のランタイムシステムにコピーさせ、前記一組のデータオブジェクトにおける各データオブジェクトについて、前記データオブジェクトに関するランタイムメタデータを識別させて、前記データオブジェクトにおけるプロキシを前記データオブジェクトに関する前記ランタイムメタデータに対する参照で置き換えさせるように操作可能なコンピュータプログラムプロダクトを特徴とする。

【００３４】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記コンピュータプログラムプロダクトは、更に、前記各データオブジェクトに関するランタイムメタデータが前記第２のランタイムシステムにおいて利用可能であるかを判定させ、そのランタイムメタデータが前記第２のランタイムシステムにおいて利用可能でなければ、前記第２のランタイムシステムにおける各データオブジェクトに関する前記ランタイムメタデータをインストールさせるように操作可能であってもよい。

【００３５】

各データオブジェクトについて、前記データオブジェクトに関するランタイムメタデータに対する参照は、前記データオブジェクトがインスタンスであるところのクラスのランタイム表現に対する参照を含むことができる。前記コンピュータプログラムプロダクトは

、更に、各オブジェクトインスタンスの前記クラスのランタイム表現が前記第２のランタイムシステムにおいて利用可能であるかを判定させ、そのランタイム表現が前記第２のランタイムシステムにおいて利用可能でなければ、前記第２のランタイムシステムにおける各オブジェクトインスタンスの前記クラスのランタイム表現をインストールさせるように操作可能であってもよい。

【００３６】

前記第１のランタイムシステムは、第１の仮想マシンを含むことができ、前記第２のランタイムシステムは、第２の仮想マシンを含むことができる。前記第１及び第２の仮想マシンは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであってもよい。前記第１のランタイムシステムは、第１の物理マシン上に配置されることができ、前記第２のランタイムシステムは、第２の異なる物理マシン上に配置されることができる。

【００３７】

前記一組のデータオブジェクトは、１又は２以上の参照データオブジェクトの推移的クロージャと第１のデータオブジェクトから構成され、各参照データオブジェクトは前記第１のデータオブジェクトによって参照されるオブジェクトである。

【００３８】

前記ランタイムメタデータを識別して前記一組のデータオブジェクトの中の各データオブジェクトにおける前記プロキシを置き換えるためのオペレーションは、前記一組のデータオブジェクトが受信された後、実質的に即座に発生することができる。或いは、それらのオペレーションは、各データオブジェクトが前記第２のランタイムシステムにおいてアクセスされるときに発生することができる。

【００３９】

他の態様において、本技術は、データ処理装置に、ユーザーセッションのためにランタイムシステムを初期化させ、共有メモリ領域におけるデータオブジェクトの共有クロージャを生成させ、前記ユーザーセッションに対応するリクエストを受信させ、一組のオペレーションシステム処理の中から第１の処理を選択させ、前記ランタイムシステムを前記第１の処理にバインド(bind)させ、前記データオブジェクトの共有クロージャを前記ランタイムシステムに関連づけさせるように操作可能なコンピュータプログラムプロダクトを特徴とする。前記データオブジェクトの共有クロージャは、１又は２以上の参照データオブジェクトの推移的クロージャと第１のデータオブジェクトから構成され、各参照データオブジェクトは、前記第１のデータオブジェクトによって参照される。

【００４０】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記ランタイムシステムは、仮想マシンを含むことができる。前記仮想マシンは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであることができる。

【００４１】

前記データオブジェクトの共有クロージャは、前記ユーザーセッションに対応するユーザーコンテキスト情報を含むことができる。

【００４２】

前記共有クロージャを前記ランタイムシステムと関連づけるステップは、前記共有クロージャを前記第１の処理にバインドするステップを含むことができる。前記共有クロージャを前記第１の処理にバインドするステップは、前記共有クロージャを前記第１の処理の前記アドレス空間にコピー又はマッピングするステップを含むことができる。もし、前記共有クロージャが前記第１の処理のアドレス空間にマッピングされれば、前記共有クロージャをバインドするステップは、更に、前記共有クロージャに対するライトアクセスを阻止し、または、前記共有クロージャにおける前記データオブジェクトの一つに対する第１のライトアクセスを検出したときに、すぐに、前記共有クロージャを前記第１の処理のアドレス空間にコピーするステップを含むことができる。

【００４３】

前記コンピュータプログラムプロダクトは、更に、前記データ処理装置に、第２のユーザーセッションのために第２のランタイムシステムを初期化させ、前記第２のユーザーセッションに対応する第２のリクエストを受信させ、前記一組のオペレーティングシステム処理の中から第２の処理を選択させ、前記第２のランタイムシステムを前記第２の処理にバインドさせ、前記データオブジェクトの共有クロージャを前記第２のランタイムシステムに関連づけさせるように操作可能であってもよい。

【００４４】

他の態様において、本技術は、データ処理装置に、共有メモリ領域にユーザーセッションに対応するユーザーコンテキストを格納させ、前記ユーザーセッションに対応するリクエストを受信させ、一組のオペレーションシステム処理の中からアドレス空間を有する処理を選択させ、一組のランタイムシステムの中から一つのランタイムシステムを選択させ、前記ランタイムシステムを前記処理にバインドさせ、前記ユーザーコンテキストを前記ランタイムシステムに関連づけさせるように操作可能なコンピュータプログラムプロダクトを特徴とする。

【００４５】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記一組のランタイムシステムにおける各ランタイムシステムは、仮想マシンを含むことができる。仮想マシンは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであることができる。

【００４６】

前記ユーザーコンテキストは、第１のデータオブジェクトと該第１のデータオブジェクトによって参照されるデータオブジェクトの前記推移的クロージャから構成される共有クロージャを含むことができる。

【００４７】

前記ランタイムシステムは、前記共有メモリ領域に格納されることができ、前記ランタイムシステムを前記処理にバインドするステップは、前記ランタイムシステムに対応する前記共有メモリ領域の一部を前記処理のアドレス空間にマッピングするステップを含むことができる。

【００４８】

前記ユーザーコンテキストを前記ランタイムシステムに関連づけるステップは、前記ユーザーコンテキストを前記処理にバインドするステップを含むことができる。前記ユーザーコンテキストを前記処理にバインドするステップは、前記ユーザーコンテキストを前記処理のアドレス空間にコピー又はマッピングするステップを含むことができる。もし、前記ユーザーコンテキストが前記処理のアドレス空間にマッピングされれば、前記ユーザーコンテキストをバインドするステップは、前記ユーザーコンテキストに対する第１のライトアクセスを検出したときに、すぐに、前記ユーザーコンテキストを前記処理のアドレス空間にコピーするステップを更に含むことができる。

【００４９】

前記コンピュータプログラムプロダクトは、前記データ処理装置に、前記ユーザーコンテキストを前記処理からバインド解除(unbind)させるように更に操作可能であってもよい。前記ユーザーコンテキストを前記処理からバインド解除するステップは、前記ユーザーコンテキストを前記共有メモリ領域にコピーするステップを含むことができる。前記ユーザーコンテキストを前記処理からからバインド解除するステップは、前記ユーザーコンテキストの前のバージョンが前記共有メモリ領域に存在するかを判定し、前記ユーザーコンテキストの前のバージョンが存在すれば、前記共有メモリ領域に前記ユーザーコンテキストの新たなバージョンを生成するステップを更に含むことができる。

【００５０】

前記処理がブロックされたことを検出すると、すぐに、前記ランタイムシステムと前記ユーザーコンテキストの双方は、前記処理からバインド解除されることができる。前記処理がブロックされたことを検出するステップは、前記処理が、入力／出力（Ｉ／Ｏ）イベ

ントが完了されたことを待っていることを検出するステップを含むことができる。Ｉ／Ｏイベントが完了されたことを検出すると、すぐに、利用可能な処理が前記一組のオペレーティングシステム処理の中から選択されることができ、そして前記ランタイムシステム及び前記ユーザーコンテキストは、前記利用可能な処理にバインドされることができる。

【００５１】

前記コンピュータプログラムプロダクトは、前記データ処理装置に、前記ランタイムシステムを前記処理からバインド解除させるように更に操作可能であってもよい。

【００５２】

前記一組のオペレーティングシステム処理は、２又はそれ以上の物理マシンに分散されることができ、そして、前記処理は、前記２又はそれ以上の物理マシンの中の第１のマシン上で実行されることができる。前記ユーザーコンテキストは、プロキシを備えた第１のデータオブジェクトを含むことができる。前記ユーザーコンテキストを前記ランタイムシステムに関連づけるステップは、前記プロキシを、前記第１のデータオブジェクトに関するランタイムメタデータに対する参照で置き換えるステップを含むことができ、前記ランタイムメタデータは、前記第１のマシンに格納される。

【００５３】

他の態様において、本技術は、一組の処理と、一組のランタイムシステムと、複数のユーザーコンテキストを格納するための共有メモリ領域と、配信コンポーネント(dispatcher component)を備えるコンピュータサーバーを特徴とする。前記配信コンポーネントは、前記複数のユーザーコンテキストにおけるユーザーコンテキストに対応するリクエストを受信し、前記一組の処理の中から利用可能な処理を選択し、前記一組のランタイムシステムの中から利用可能なランタイムシステムを選択し、前記利用可能なランタイムシステムと前記ユーザーコンテキストの識別を、前記リクエストの処理のための前記利用可能な処理に伝送するように操作可能である。

【００５４】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記一組のランタイムシステムにおける各ランタイムシステムは、仮想マシンを含むことができる。前記仮想マシンは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであることができる。前記一組の処理における処理の数は、前記一組のランタイムシステムにおけるランタイムの数以下に設定されることができる。

【００５５】

他の態様において、本技術は、データ処理装置に、一組の処理の中の各処理におけるランタイムシステムを初期化させ、共有メモリ領域にユーザーセッションに対応するユーザーコンテキストを格納させ、前記ユーザーセッションに対応するリクエストを受信させ、前記一組の処理の中から一つの処理を選択させ、前記リクエストを処理するために、前記選択された処理における前記ランタイムシステムに前記ユーザーコンテキストを関連づけさせるように操作可能なコンピュータプログラムプロダクトを特徴とする。

【００５６】

有利な実施例は、１又は２以上の次の特徴を含むことができる。前記選択された処理における前記ランタイムシステムに前記ユーザーコンテキストを関連づけるステップは、前記ユーザーコンテキストを前記選択された処理にバインドするステップを含むことができる。前記各処理におけるランタイムシステムは、仮想マシンを含むことができる。前記仮想マシンは、Ｊａｖａ（登録商標）仮想マシン又は共通言語ランタイム仮想マシンであることができる。

【００５７】

本明細書で述べられる本技術は、１又は２以上の次の利点を実現するために実施できる。本技術は、概して、或るランタイムシステムにおける複合データ構造体を分解(disassemble)し、そして、別のランタイムシステムの本来のフォーマットでそれを再構築(reassemble)するために使用できる。更に詳しくは、本技術は、ランタイムシステム（例えばＶＭ）間でオブジェクトを共有するために使用できる。共有可能なクラス及び共有可能なオ

ブジェクトインスタンスは、自動的に識別される。共有可能なものとして自動的に識別されないクラスは、それでもそれらが共有可能かどうかを判定するために、手動で分析されることができる。共有可能なオブジェクトは、共有クロージャと呼ばれる組にグループ化され、この共有クロージャは、複数のランタイムシステムと関連づけられることができる。複数のタイプのオブジェクトが共有されることができ（例えば、ユーザーコンテキストを構成するオブジェクトを含む）、そして複数のバージョンのオブジェクトが生成され共有されることができる。共有されたオブジェクトは、複数のメカニズムを通じてアクセスされることができ、この複数のメカニズムは、１又は２以上のランタイムシステムと関連するアドレス空間に共有メモリ領域から前記共有オブジェクトをコピーしマッピングするステップを含む。他のランタイムシステムにおいて共有オブジェクトを無効又は使用不能とする方法で前記共有オブジェクトにおけるデータをランタイムシステムがオーバーライトすることを防止するために、前記共有オブジェクトに対するアクセスを制限することができる。また、共有オブジェクトは、削除されガーベッジコレクトされたものとしてマークされることもできる。

【００５８】

オブジェクトは、異なる物理マシン上のランタイムシステム（例えばクラスターアーキテクチャ）間で共有されることができる。ソースランタイムシステムにおけるランタイムメタデータに対する参照は、ターゲットランタイムシステムにおけるランタイムメタデータに対する参照で置き換えられ識別されることができる。ランタイムメタデータは、ターゲットシステムにロードされることができ、そしてこのようなランタイムメタデータに対する参照は、しきりに又はゆっくりと（例えば要求に応じて）プラグイン(plug-in)されることができる。前記オブジェクトの共有は、資源の消費を低減させることができ（例えば、オブジェクトが複数のランタイムシステムによって共有されるので、使用するメモリが少なくてすみ）、同様に、時間の消費も少なくてすむ（例えば、共有されたオブジェクトが複数のランタイムシステムにおいて生成又は初期化される必要がないので、少ない時間ですむ）。

【００５９】

本明細書において述べられる本技術は、また、拡張可能な方法でユーザーセッション間の分離を提供するために使用されることができ、これにより、多くのユーザーセッションに対応するリクエストをサーバーがしっかりと処理することを可能にしている。ユーザーセッションは、各ユーザーセッションに対し一つのＶＭを割り当て、そしてリクエストを処理するために前記ＶＭをＯＳワーク処理にバインドすることにより、分離されることができる。ワーク処理の数は、スループットを増加させるために調整されることができる（例えば、ワーク処理の数は、サーバーに割り付けられたプロセッサの数と一致するように設定されることができる）。前記ＶＭは、資源及び時間の消費を低減するために、ユーザーコンテキストを含んでオブジェクトを共有することができる。

【００６０】

或いは、各ユーザーセッションに対して一つのＶＭを割り当てることよりも、ＶＭは、また、ユーザーセッション間で共有されることもできる。従って、サーバーは、固定された数のＶＭで実施されることができる。サーバーは、また、可変ではあるが制限された数（例えば最大値）のＶＭで実施されることもできる。ユーザーセッション間でＶＭを共有することは、ユーザーセッションごとのオーバーヘッドを低減させることができ、特に、ユーザーコンテキストが小さい環境において上記オーバーヘッドを低減させることができる。ＶＭを共有するにもかかわらず、それでも、ユーザーセッションは、たった一つのＶＭと一つのユーザーコンテキストを各ワーク処理に取り付け、そして、上記取り付けられたユーザーコンテキストのユーザーセッションに対応するリクエストを処理するために上記取り付けられたＶＭを使用することにより、分離された状態を維持することができる。それでも、スループットは最適化されることができる。なぜなら、ユーザーコンテキストとＶＭの双方が、バインドされ、そしてワーク処理が利用可能な限り処理で実行されることができるからである。

【００６１】
上述した分離される方法でＶＭを共有するために、ユーザーコンテキストは、上記ＶＭから分離されることができる。上記ユーザーコンテキストは、複数のＶＭによって、例えば、サーバーにおける全てのＶＭによってアクセス可能な共有メモリ領域にユーザーコンテキストを格納することによって、共有されアクセスされることができる。リクエストは、ＶＭをワーク処理にバインドし、関連するユーザーコンテキストを前記ＶＭと関連づけ、そして前記関連づけられたユーザーコンテキストと共に前記ワーク処理における前記ＶＭを実行することにより処理されることができる。たった一つのＶＭと一つのユーザーコンテキストを同時に処理と関連づけることにより、ユーザーセッションは相互に分離され、従ってＶＭ又はワーク処理がクラッシュしても、前記関連づけられたユーザーセッションのみが影響を受ける。

【００６２】
複数のメカニズムが、ユーザーコンテキストをＶＭに関連づけるために使用されることができ、そして複数のタイプのアクセスが、前記ユーザーコンテキストにおけるデータに対して提供されることができる。スループットは、ユーザーコンテキストをＶＭと関連づけるための低コストのメカニズムの使用（例えば、ユーザーコンテキストとＶＭを処理にマッピングすること）を通じて最適化されることができ、同様に、ブロッキングＶＭをそれらのワーク処理から取り外し、他のリクエストを処理するためにそれらのワーク処理を使用することにより、最適化されることができる。ユーザーコンテキストは、クラスターアーキテクチャにおいて共有されることができ、ここで、ＶＭは、複数の物理マシンに分散される。配信処理(dispatcher process)又は配信コンポーネントは、全体の作業負荷を分散させるように、または、マシン間でユーザーコンテキストを共有する必要性を最小化するように、ＶＭ間でリクエストを分散するために使用されることができる。ユーザーコンテキストは、たとえＶＭが永久的にワーク処理にバインドされても、ＶＭ間で共有されることができる（そして、ＶＭは、それ故に、ユーザーセッション間で共有されることができる）。

【００６３】
各ユーザーセッションについて個別のＶＭを割り当てること、或いは、上述した分離法でＶＭを共有することは、資源消費の観点から責任(accountability)を生じる。それは、特定のＶＭまたはオペレーティングシステム処理のための資源の消費をモニターするには分かりやすい方法ではあるが、ＶＭ又は処理において動作しているプログラムのどの部分が特定の資源の消費（例えば、メモリ、処理時間、或いはガーベッジコレクションに費やされる時間）の原因となっているかを判定することは極めて困難である。しかしながら、もし、ＶＭにおいて動作するユーザーコンテキストが一つしかなければ、資源はユーザーセッションベースで説明されることができ、それは、多くの場合は望ましい。

【００６４】
本発明の一実施は、上述の利点の全てを提供する。

【００６５】
これらの一般的および具体的な態様は、コンピュータプログラム、方法、システム、装置、または、コンピュータプログラム、方法、システムの任意の組み合わせを用いて実施されることができる。本発明の１又は２以上の実施形態の詳細は、添付の図面を参照しながら以下に説明される。本発明の他の特徴、目的、および利点は、以下の説明、図面、請求項から明らかになるであろう。

【００６６】
図１は、クライアント／サーバーシステムのブロック図である。

【００６７】
図２は、図１のクライアント／サーバーシステムにおけるサーバーのブロック図である。

【００６８】
図３は、図２におけるサーバーの他のブロック図である。

【００６９】
図４は、図２のサーバーにおけるリクエストの処理を説明するための処理フローチャートである。
【００７０】
図５は、サーバーにおける共有オブジェクトの使用を説明するためのブロック図である。
【００７１】
図６は、オブジェクトインスタンスが共有可能であるかどうかを判定するための処理を説明するためのフローチャートである。
【００７２】
図７は、クラスが共有可能であるかどうかを判定するための処理を説明するためのフローチャートである。
【００７３】
図８及び図９は、共有オブジェクトを生成し使用するための処理を説明するためのフローチャートである。
【００７４】
図１０は、複数のマシンでオブジェクトを共有するための処理を説明するためのフローチャートである。
【００７５】
図１１は、サーバーにおける共有オブジェクトを使用するための処理を説明するためのフローチャートである。
【００７６】
図１２は、図１のクライアント／サーバーシステムにおける他のサーバーのブロック図である。
【００７７】
図１３は、図１２におけるサーバーの他のブロック図である。
【００７８】
図１４は、図１２のサーバーにおけるリクエストの処理を説明するためのブロック図である。
【００７９】
図１５は、図１２のサーバーにおけるリクエストの処理を説明するための処理フローチャートである。
【００８０】
図１６は、図１のクライアント／サーバーシステムのうちの他のサーバーにおけるリクエストの処理を説明するための処理フローチャートである。
【００８１】
各図面における同様の参照番号及び記号は、同様の要素を示す。
【発明を実施するための最良の形態】
【００８２】
ユーザーセッションの分離(Isolating User Session)
【００８３】
図１は、クライアント／サーバーシステム１００を示し、同図において、ネットワーク１５０を介して、サーバー２００はクライアントシステム１０２，１０４，１０６と接続される。サーバー２００は、本発明による装置、プログラム、方法の実施に適したプログラム可能なデータ処理システムである。サーバー２００は、ユーザーリクエストを処理する１又は２以上のランタイムシステムのためのコアオペレーティング環境を提供する。サーバー２００は、プロセッサ２０２とメモリ２５０を備える。メモリ２５０は、オペレーティングシステムと、ネットワーク１５０を介して通信するためのＴＣＰ／ＩＰ(Transmission Control Protocol/Internet Protocol)スタックと、プロセッサ２０２によって実行されるマシン実行可能な命令(instruction)を格納するために使用されることができる

。幾つかの実施例では、サーバー２００は、複数のプロセッサを備えることができ、そのそれぞれは、マシン実行可能な命令を実行するために使用されることができる。メモリ２５０は、共有メモリ領域(shared memory area)２５５（後述の図に示される）を備え、この共有メモリ領域２５５は、サーバー２００において実行する複数のオペレーティングシステムによってアクセス可能となっている。クライアント／サーバーシステム１００で使用されることができる適切なサーバーの例としては、ドイツのＳＡＰ（登録商標）ＡＧ（ＳＡＰ）によって開発されたウェブアプリケーションサーバー、或いは、ＮＹ，アーモンクのＩＢＭ（登録商標）社によって開発されたウェブスフェアアプリケーションサーバーのような、Ｊａｖａ（登録商標）２プラットフォーム、エンタープライズ版（Ｊ２ＥＥ）コンパチブルのサーバーがある。

【００８４】

クライアントシステム１０２，１０４，１０６は、複数のアプリケーション又はアプリケーションインターフェイスを実行することができる。アプリケーションインターフェイス又はアプリケーションの各インスタンスは、ユーザーセッションを構成することができる。各ユーザーセッションは、サーバー２００によって処理されるべき１又は２以上のリクエストを生成することができる。リクエストは、サーバー２００上のランタイムシステム（例えばＶＭ３３０）上で実行されるべき命令を含むことができる。

【００８５】

ランタイムシステムは、ユーザーリクエストにおけるコードまたは命令を実行するコード実行環境(code execution environment)であり、そのコードにランタイムサービスを提供する。コアランタイムサービス(core runtime services)は、処理(process)、スレッド(thread)、メモリマネージメント（例えば、サーバーメモリ２５０におけるオブジェクトの割り付け、オブジェクトの共有、オブジェクトに対する参照のマネージメント、オブジェクトのガーベッジコレクション）のような機能性(functionality)を含むことができる。機能強化されたランタイムサービスは、エラー処理や、セキュリティ及び接続性の確立のような機能性を備えることができる。

【００８６】

ランタイムシステムの一例として、仮想マシン(Virtual Machine ;VM)がある。仮想マシン（ＶＭ）は、実際のマシンまたはプロセッサのように、命令セット、一組のレジスタ(register)、スタック(stack)、ヒープ(heap)、メソッド領域(method area)を備えることができる概念上のマシンである。ＶＭは、本質的には、プログラムコードが実行されるべき実際のプロセッサ又はハードウェアプラットフォームとプログラムコードとの間のインターフェイスとして振る舞う。プログラムコードは、ＶＭの資源をコントロールするＶＭ命令セットからの命令を含む。ＶＭは、ＶＭが動作するプロセッサ又はハードウェアプラットフォーム上で命令を実行し、そして、プログラムコードの命令を発効させるために、これらプロセッサ又はハードウェアプラットフォームの資源をコントロールする。このようにして、同一のプログラムコードが、各プロセッサ又はハードウェアプラットフォームのために再コンパイル又は再書き込みされる必要なく、複数のプロセッサ又はハードウェアプラットフォーム上で実行されることができる。その代わり、ＶＭは各プロセッサ又はハードウェアプラットフォームのために実施され、そして同一のプログラムコードが各ＶＭにおいて実行されることができる。ＶＭは、プロセッサ又はハードウェアプラットフォームによって認識されるコードの形態で実施されることができる。或いは、ＶＭは、プロセッサに直接的に組み込まれるコードの形態で実施されることができる。

【００８７】

一例として、Ｊａｖａ（登録商標）ソースプログラムは、バイトコードとして知られるプログラムコードにコンパイルされることができる。バイトコードは、任意のプロセッサ又はプラットフォーム上で動作するＪａｖａ（登録商標）ＶＭ上で実行されることができる。Ｊａｖａ（登録商標）ＶＭは、一回に一つの命令のバイトコードを解釈(interpret)することができる、または、バイトコードは、ジャストインタイム(just-in-time ;JIT)で実際のプロセッサ又はプラットフォームのために更にコンパイルされることができる。

【００８８】
Ｊａｖａ（登録商標）ＶＭに加え、ＶＭの他の例は、アドバンスドビジネスアプリケーションプログラミング言語(Advanced Business Application Programming language ;ABAP)と、共通言語ランタイム(Common Language Runtime ;CLR)ＶＭを含む。ＡＢＡＰは、ＳＡＰ Ｒ／３システムのための開発アプリケーション用のプログラミング言語であり、ＳＡＰ（登録商標）により開発されたビジネスアプリケーションシステムに広くインストールされている。共通言語ランタイムは、ＷＡ，レッドモンドのマイクロソフト（登録商標）社により開発された管理コード実行環境(managed code execution environment)である。説明の簡略化のため、本明細書における議論は、仮想マシンに焦点を当てるが、本明細書で述べられる本技術は、他のタイプのランタイムシステムでも使用できることが理解されるべきである。

【００８９】
ユーザーセッションを互いに分離し、これによりシステムのロバスト性を増すために、サーバー２００は、各ユーザーセッションにそれぞれＶＭが提供されるように実施されることができる。更に詳しくは、各ユーザーセッションには、それぞれ、ＰＡＶＭ(Process-Attachable Virtual Machine)が提供され、ＰＡＶＭは、ＯＳ処理に取り付けられる(attach)と共にＯＳ処理から取り外される(deattach)ことが可能なＶＭである。

【００９０】
図２は、処理取り付け可能なＶＭ(process-attachable VMs)を使用するサーバー２００の実施例を示す。処理取り付け可能なＶＭのインスタンスが作成され－例えば、ＶＭ３０１は、そのインスタンスが作成されてユーザーセッション１のために使用され、そして、ＶＭ３０３は、そのインスタンスが作成されてユーザーセッション２のために使用される。通常、もし、Ｕ個のユーザーセッションが存在すれば、サーバー２００は、Ｕ個の対応するＶＭを備える。これは、図２においてユーザーセッションＵに対応するＶＭ３０９によって概念的に示されている。

【００９１】
また、図２に示されるサーバー２００の実施例は、共有メモリ領域２５５を備え、この共有メモリ領域は、処理取り付け可能なＶＭ、ワーク処理のプール(pool)４００、配信処理４１０を格納するために使用される。ワーク処理のプール４００は、サーバー２００に割り付けられた多数のオペレーティングシステム処理－例えばワーク処理４０１，４０３，４０９を含む。概して、図２においてワーク処理４０９によって概念的に示されるように、Ｐ個のワーク処理をサーバー２００に割り付けることができる。同期オーバーヘッドとコンテキストスイッチの個数を低減させるために、サーバー２００に割り付けられたオペレーティングシステム処理Ｐの数は、概ね、サーバー２００が動作するコンピュータ上で利用可能な処理の数と等しくなければならない。しかしながら、増強された分離(isolation)とロバスト性(robustness)の利益は、サーバー２００がユーザーリクエストを処理するために、たった一つのオペレーティングシステム処理しか備えない場合でさえ達成されることができる。

【００９２】
オペレーションにおいて、図２のサーバー２００がリクエストを受信すると、特殊な処理（配信処理４１０）が、ワーク処理プール４００における利用可能なワーク処理の一つに対してリクエストを配信する。そのリクエストがユーザーセッション１に対応し（それがＶＭ３０１に対応し）、且つ、配信処理４１０がそのリクエストをワーク処理４０１に配信すると仮定すると、サーバー２００は、そのリクエストを処理するために、ワーク処理４０１にＶＭ３０１を取り付け(attach)またはバインド(bind)する。そのリクエストが処理された後、サーバーは、ワーク処理４０１からＶＭ３０１を取り外すことができる。そして、ワーク処理４０１は、他のリクエストを処理するために使用されることができる。例えば、もし、サーバー２００がユーザーセッション２に対応する新たなリクエストを受信し、配信処理４１０が、そのリクエストをワーク処理４０１に配信すれば、対応するＶＭ３０３は、この新たなリクエストを処理するためにワーク処理４０１に取り付けられ

ることができる。この例の説明を続けると、もし、ユーザーセッション２に対応するリクエストを処理するためにワーク処理４０１が依然として使用されている間にサーバー２００がユーザーセッション１に対応する他のリクエストを受信すれば、配信処理４１０は、ユーザーセッション１からワーク処理４０３に新たなリクエストを配信し、そしてユーザーセッション１のためのＶＭ３０１が、ユーザーセッション１からの新たなリクエストを処理するためにワーク処理４０３に取り付けられることができる。

【００９３】

サーバー２００は、配信処理４１０を備える必要はない。例えば、代替の実施例では、ユーザーリクエストは、サーバー２００に割り付けられた処理に連続的に割り当てられることができる。各処理は、リクエストのキュー(queue)を保持し、そしてそのリクエストに対応するユーザーセッションの処理取り付け可能なＶＭを、そのリクエストを処理するために、そのキューの前(front)に取り付けることができる。

【００９４】

上述したように、処理取り付け可能なＶＭは、ＯＳ処理に取り付け可能であるとともにＯＳ処理から取り外し可能なＶＭである。処理からＶＭを取り外すために（そして、そのＶＭを他の処理に取り付けるために）、ＶＭと該ＶＭが動作する処理との間の親和性(affinity)は除去される必要がある。ＶＭが処理から取り外される場合、ＶＭのステートは存続される必要がある。ＶＭが他の処理に取り付けられる場合、このＶＭのステート(state)は存続されてはならない。従って、ＶＭのステートは、そのステートを存続させることを可能とすると共にそのステートを存続させないことを可能とするような方法で保持され発生される必要がある。さらに、処理からＶＭを取り外すときにＶＭのステートを存続すること、及びＶＭを別の処理に取り付けるときにＶＭのステートを存続させないことは、望ましくは、低コストのオペレーションであるべきである。このことは、サーバーに割り当てられたＯＳ処理にアクセス可能である共有メモリ領域にＶＭのステートを格納することにより達成される。

【００９５】

また、共有メモリに配置されたＶＭがユーザーセッションに対応するユーザーコンテキストをアクセスするために、ユーザーコンテキスト－ユーザーヒープ(user heap)とユーザースタックの両方を含む－は、共有メモリに配置されなければならない。（或いは、ユーザーコンテキストは、ＶＭがバインドされた処理のアドレス空間にコピーされることができ－このような実施例は以下で更に詳しく説明される。）従って、サーバー２００の一実施例において、ユーザーコンテキストは、また、共有メモリに格納される。共有メモリにユーザーヒープを格納することは正攻法である－上記ヒープ用のメモリは、共有メモリセグメントから単に割り当てられることができる。上記ユーザースタックを格納することはより困難である。なぜなら、Ｊａｖａ（登録商標）ＶＭと同様に、いくつかのインスタンスでは、ユーザースタック及びＶＭスタックは混合(intermix)されるからである。この場合、一つの解法は、ユーザースタックを含んで、ＶＭのスタックの全てを共有メモリに格納することである。一実施例において、このことは、ＶＭをオペレーティングシステムコルーチン(operating system co-routine)として実施することにより達成される。

【００９６】

典型的なサブルーチンは階層的関係を呈する。例えば、サブルーチンＡは、サブルーチンＢを呼び出すときにサスペンド(suspend)し、そしてサブルーチンＢが終了すると制御をサブルーチンＡに戻し、そしてサブルーチンＡはサスペンドした点から実行を再開する。一方、コルーチンは、階層的関係というよりも、むしろ並列的関係を有する。従って、例えば、コルーチンＡは、コルーチンＢを呼び出すとサスペンドするが、コルーチンＢは、また、制御をコルーチンＡに戻したときにサスペンドする。コルーチンＢに対しては、この制御の返還はコルーチンＡの起動のようなものである。その後、コルーチンＡがコルーチンＢを読み出してサスペンドすると、コルーチンＢは、コルーチンＡの前回の起動がリターンされるように振る舞い、そして、それは、その起動の点から実行を再開させる。従って、制御が二つのコルーチンの間で往来し、それぞれが、前回の中止位置で再開する

。

【００９７】

いくつかの実施例において、コルーチンのそれぞれが固有のスタックを備え、ヒープを共有(share)するので、コルーチンはスレッドと比較されることができる。しかしながら、スレッドとコルーチンとの一つの違いは、オペレーティングシステムがスレッド間のスケジューリングを処理するのに対し、プログラマーがコルーチン間のスケジューリングを管理しなければならないことである。コルーチンは、後述するように、スレッドを模擬するために使用される。例えば、Ｌｉｎｕｘ　ｇｌｉｂｃ　ライブラリー内の一組の機能は、setcontext( ),getcontext( ),makeconetxt( ),swapcontext( )の各機能を備え、コルーチン間で切り替えるために、そして重要なことには、コルーチンのスタック用のメモリを提供するために、処理内の新たなコルーチンをスタートさせるために使用されることができる。この最後の特徴は、共有メモリからＶＭのスタックを割り当てるために（そして、共有メモリにスタックを格納するために）使用されることができる。

【００９８】

サーバー２００の一実施例において、ユーザーセッションが開始するとき、このユーザーセッションに対応する処理取り付け可能なＶＭが生成され初期化される。“セッションメモリ”と呼ばれる、共有メモリのプライベートブロックはＶＭに割り当てられる。ＶＭのヒープとスタックは、共有メモリのこのプライベートブロックから直接的に割り当てられる。オペレーティングシステム処理は、単にＶＭのセッションメモリをそのアドレス空間にマッピング(map)することができるだけであるから、ＶＭのステートを格納するために共有メモリを使用することは、ＶＭをＯＳ処理に取り付け又はバインドする処理を実質的にノンオペレーション(non-operation)にする。同様に、ＯＳ処理からＶＭを取り外す処理は、単に、ＯＳ処理のアドレス空間からＶＭのセッションメモリをマッピング解除(unmapping)することを必要とする。データは、実際には移動またはコピーされない。

【００９９】

また、処理からＶＭを取り外すこと（そして、ＶＭを他の処理に取り付けること）を可能とするためには、ファイルハンドル(file handle)やソケット(socket)のような、ＶＭによって使用される入力／出力（Ｉ／Ｏ）資源も存続される必要がある。ＶＭによって使用されるＩ／Ｏ資源は、資源を存続させること及び存続させないことを可能にするような方法で生成され維持される必要がある。このことは、Ｉ／Ｏ資源をアクセスするためのインディレクション(indirection)の追加的レベルを使用することにより達成されることができる。例えば、ＶＭがファイル又はソケットディスクリプターとして見ることは、実際には丁度、ファイル又はソケットへのハンドル(handle)である。ハンドルは、それ自体、存続可能(persistable)であり、例えば、ファイルハンドルは、ＶＭのセッションメモリに格納されることができる。或いは、ハンドルは、資源マネージャの使用を通じて存続可能とされることができ－例えば、ソケットの場合、デスクリプターパッシング(descriptor passing)が、特定のソケット用のＩ／Ｏリクエストのための Ｉ／Ｏリクエストに関するソケットマネージャを通知するために使用されることができる；そしてソケットマネージャは、Ｉ／Ｏリクエストが完了されると配信手段を通知することができる。

【０１００】

いくつかのＶＭは、マルチスレッドであってもよい。スレッドは、本質的には、単一プログラムの使用の範囲内で複数のサービスリクエスト又はコンカレントユーザーをプログラムが処理することを可能にするプレースホルダー情報(placeholder information)である。プログラムの観点から、スレッドは、一人の個人ユーザー又は特定のサービスリクエストを扱うのに必要な情報である。もし、複数のユーザーがプログラムを一斉に使用し、又はプログラムが同時に多数のリクエストを受信すれば、スレッドは、このような各ユーザー又はリクエストのために生成されて維持される。このスレッドは、プログラムが、異なるユーザー又はリクエストの代わりに交互に再入力reenter)されるので、どのユーザー又はリクエストが扱われているのかをプログラムが知ることを可能にする。

【０１０１】

本来のＯＳスレッド(native OS thread)は、容易に共有メモリに存続(persist)されることができる。従って、マルチスレッドのＶＭを処理から取り外すことを可能にするためには（そして、ＶＭを他の処理に取り付けることを可能にするためには）、本来のスレッドは、ユーザーレベルのメカニズムにより模擬(simulate)されることができる。本来のスレッドを模擬するためにユーザーレベルのメカニズムを導入することは、“グリーンスレッド(green thread)”機能性を提供すると呼ばれる。グリーンスレッドは、オペレーティングシステムよりもユーザーによってスケジュールされるユーザーレベルのスレッドである。

【０１０２】

本来のスレッドを模擬するために使用することができる一つのメカニズムは、オペレーティングシステムコルーチン(operating system so-routines)である。上述したように、コルーチンは、（オペレーティングシステムよりは）プログラマーがコルーチン間のスケジューリングを管理することを除いては、スレッドに似ている。従って、ＶＭスレッドを実施するために使用できる本来のＯＳスレッドは、コルーチンにマッピングされることができる。スレッドのコールスタック(call stacks)、ミューテックス(mutexes)、Ｊａｖａ（登録商標）モニターのための条件変数(condition variables)を含んで、スレッドマネージメント及びスケジューリングに関連する全てのデータ構造は、ＶＭのセッションメモリに保持されることができる。Ｊａｖａ（登録商標）ＶＭの場合では、それは、例えば動的メソッド起動(dynamic method invocation)のＪＮＩ(Java（登録商標） Native Interface)実施のために、ＶＭ実施によって使用されるＣスタック及びＪａｖａ（登録商標）スタックの両方を含むことができる。

【０１０３】

通常、オペレーティングシステムは、フェアネス(fairness)を最大化するために（すなわち、或る時点で稼動する機会を各スレッドに与えるために）、先制してスレッドをスケジューリングする。一方、プログラマーによって扱われるコルーチンスケジューリングは、通常、先制的ではない。しかしながら、それは、必ずしも欠点ではなく、サーバーとの関連で、リクエストスループットは、しばしばフェアネスよりも重要である。リクエストスループットは、拡張性にとって主要目的であり、コルーチンをスケジューリングするためのバッチ処理法を用いることにより最大化されることができる。バッチ処理法において、各ＶＭ内では、各コルーチンは、ウェイトステートに入るとき（例えば、Ｊａｖａ（登録商標）モニターのようなモニター又はＩ／Ｏ上でのブロッキングのとき）、スケジューラー（時々、スレッドスケジューラと呼ばれる）に協力的に服従する。ブロッキングＩ／Ｏコールとスレッドスケジューラとの間の調整(coordination)は、上述したＩ／Ｏ出力先変更メカニズム(I/O redirection mechanism)の一部として含まれることができる。ミューテックス及び条件変数は、セッションメモリにおけるスケジューラー制御の変数として基本命令(primitives)を固定(lock)するオペレーティングシステムを用いることなく実施されることができる。

【０１０４】

コルーチンスケジューリングは、その全てのコルーチンがウェイトステートに入るまで、ＰＡＶＭのために継続することができ、つまり、ユーザーリクエストが完了したか、またはＩ／Ｏを待っている。何れの場合においても、ＰＡＶＭは、ワーク処理から取り外されることができる。配信処理４１０は、次のリクエストがユーザーセッションから到来したとき、またはＩ／Ｏリクエストが完了したときに、利用可能なワーク処理にＰＡＶＭを再取り付けすることができる。

【０１０５】

上述したように、ＶＭスレッドを実施するために使用される本来のスレッドは、コルーチンにマッピングされることができる。一方、例えばガーベッジコレクションなどのために、ＶＭにおいて内部的にのみ使用される本来のスレッドは、同期ファンクションコール(synchronous function calls)で置き換えられることができる。同期ファンクションコールを使用することは、内部的な本来のスレッドを模擬する手段として考えられる。或いは

、もし、内部的な本来のスレッドによって実行されるべきファンクションが非同期的に実行されるのであれば、このファンクションは、サーバー内の指定された優先度の低い処理上で実行されスケジューリングされることができる。さらに他の代替は、いくつかのファンクションを一緒に削除(omit)することである。例えば、ユーザーセッションがアクティブである期間中にガーベッジコレクションを遂行するよりも、むしろ、ユーザーセッションが終了したときに全てのセッションメモリを単に開放することの方が可能である。

【０１０６】

図３は、Ｊａｖａ（登録商標）２プラットフォーム，エンタープライズ版（Ｊ２ＥＥ）アプリケーションサーバー２００の例を示し、アプリケーションサーバー２００は、共有メモリ２５５とワーク処理４０１，４０３を備える。Ｊ２ＥＥは、大企業に特有なメインクレーム規模のコンピューティングのために設計されたＪａｖａ（登録商標）プラットフォームである。Ｊ２ＥＥは、標準化された再利用可能なモジュラーコンポーネントを生成し、そして層(tier)がプログラミングの多くの局面を自動的に処理することを可能とすることにより、階層的(tiered)で薄いクライアント環境においてアプリケーション開発を簡略化する。Ｊ２ＥＥは、Ｊａｖａ（登録商標）２ディベロップメントキット（ＪＤＫ）のような、Ｊａｖａ（登録商標）２プラットフォーム，標準版（Ｊ２ＳＥ）の多くのコンポーネントと、ＣＯＲＢＡ(Common Object Request Broker Architecture)、ＪＤＢＣ(Java（登録商標） Database Connectivity 2.0)、セキュリティモデルのためのサポートを含む。また、Ｊ２ＥＥは、ＥＪＢ(Enterprise Java（登録商標） Beans)、ＡＰＩ(Java（登録商標） servlet application programming interface)、ＪＳＰ(Java（登録商標） Server Pages)のためのサポートを含む。

【０１０７】

各ワーク処理４０１，４０３は、コンテナー２０４と、ＲＭＩ(Remote Method Invocation)インターフェイス２０６及びデータベースマネージメントシステム（ＤＢ）インターフェイス２０８のようなサービスインターフェイスを備えることができる。ＲＭＩは、異なるコンピュータ上のオブジェクトが分散型ネットワーク上で相互通信することを可能にするオブジェクト指向のプログラミング技術である。ＲＭＩは、一般にＲＰＣ(Remote Procedure Call)として知られているＪａｖａ（登録商標）版のプロトコルであるが、リクエストと一緒に１又は２以上のオブジェクトを通過する追加的能力を備えている。

【０１０８】

コンテナー(container)２０４は、本来のＯＳによって提供されるフレームワークよりもサーバーアーキテクチャに適したフレームワークを提供するＯＳインターフェイスを備えることができる。このＯＳインターフェイスは、スケジューリング、メモリ管理、処理アーキテクチャ、クラスタリング、ロードバランシング、ネットワーキングなどのように、本来のＯＳから一定のファンクションに対する責務を引き継ぐことができる。このようなファンクションを引き継ぐことにより、コンテナー２０４は、リクエストスループットのような一定の対象を最適化するように、プロセッサやメモリのような重要な資源の使用を制御することができる。また、コンテナー２０４は、サーバー上で動作するアプリケーションからサーバーの本来のオペレーティングシステムの詳細を隠す役目を果たすことができる。

【０１０９】

加えて、コンテナー２０４は、インターフェイスとしての役目も果たすことができ、このインターフェイスを通じて、処理取り付け可能なＶＭが処理に取り付けられ、その処理において実行され、そして、その処理から取り外される。従って、図３に示されるサーバーの実施例において、コンテナー２０４は、（例えば、ＶＭに割り当てられた共有メモリブロックを処理のアドレス空間にマッピングすることにより）ＶＭのステートを存続させることや存続させないこと、およびＶＭのスレッド又はコルーチンのスケジューリングのような、処理取り付け可能なＶＭの使用のための幾つかの機能性を提供する。

【０１１０】

図３の例において、第１のユーザーセッション（ユーザーセッション１）は、ワーク処

理４０１にバインドされている。ユーザーセッション１は、対応するユーザーコンテキスト５０１及びＶＭ３０１を備える。共有メモリ２５５のブロック２５７は、ユーザーセッション１に割り付けられている。ユーザーコンテキスト５０１およびＶＭ３０１のステートは、共有メモリブロック２５７に格納される。この例では、ユーザーセッション１に対応するリクエストは、ワーク処理４０１に配信され受信されている。このリクエストを処理するために、ユーザーセッション１に対応するＶＭ３０１は、ワーク処理４０１にバインドされる。従って、ユーザーコンテキスト５０１とＶＭ３０１のステートの両方は、共有メモリブロック２５７からワーク処理４０１のアドレス空間にマッピングされている。そして、ＶＭ３０１は、リクエストを処理するためにワーク処理４０１によって実行される。リクエストが処理されると、ユーザーセッション１は、ワーク処理４０１からバインド解除(unbind)されることができ、つまり、対応するＶＭとユーザーコンテキストは、（例えば、ワーク処理４０１のアドレス空間からＶＭ３０１のステートとユーザーコンテキスト５０１をマッピング解除(unmapping)することにより）ワーク処理４０１から取り外されることができる。

【０１１１】

また、図３は、ワーク処理４０３にバインドされたユーザーセッション２を示す。ユーザーセッション２は、対応するユーザーコンテキスト５０３とＶＭ３０３を備える。共有メモリ２５５のブロック２５９は、ユーザーセッション２に割り当てられる。ユーザーコンテキスト５０３及びＶＭ３０３のステートは、共有メモリブロック２５９に格納され、そしてワーク処理４０３のアドレス空間にマッピングされる。ユーザーセッション２からのリクエストが処理されると、ユーザーセッション２は、ワーク処理４０３からバインド解除(unbind)されることができ、つまり、（例えば、ワーク処理４０３のアドレス空間からユーザーコンテキスト５０３とＶＭ３０３のステートをマッピング解除することにより）ワーク処理４０３から、対応するＶＭとユーザーコンテキストが取り外されることができる。

【０１１２】

ＶＭ３０３がウェイトステート(wait state)にあるか、または、ＶＭ３０３がユーザーセッション２からのリクエストの処理を完了したために、ＶＭ３０３がワーク処理４０３から取り外されたと仮定すると、もし、新たなリクエストがユーザーセッション１から到着すれば、この新たなリクエストは、例えばワーク処理４０３に配信されることができる。そして、ユーザーセッション１に対応するユーザーコンテキスト５０１及びＶＭ３０１は、ワーク処理４０３にバインドされることができる。一実施例において、このことは、如何なるデータの移動やコピーを必要とせず－むしろ、共有メモリブロック２５７（又はその適切な一部）は、ワーク処理４０３のアドレス空間に単にマッピングされるだけである。そして、ワーク処理４０３は、ユーザーセッション１からの新たなリクエストを処理するためにＶＭ３０１を実行することができる。

【０１１３】

従って、ユーザーセッションは、異なる処理にバインドされることができ、異なる処理間で移動することができる。このように、リクエストは、ワーク処理が利用可能になるとすぐに処理されることができる。さらに、ユーザーセッションを異なる処理にマッピングすることは、通常、非常に安価なオペレーションである。その結果、リクエストスループットが最適化される。また、リクエストスループットが、サーバーにさらに多くの処理を割り付けることにより最適化されることができるので、処理取り付け可能なＶＭは、サーバーに拡張性をもたらす。また、さらに多くの処理をより良く扱うために、潜在的コンピュータ(underlying computer)に更に多くの処理を加えることが可能である。

【０１１４】

また、処理取り付け可能なＶＭの使用はサーバーを堅牢にする。このことは、ＯＳが、処理（例えば、サーバー処理プール４００におけるワーク処理）間で提供する分離(isolation)によるものである。また、一つのユーザーセッションのメモリとＶＭのみを１度にワーク処理にマッピングすることにより、ユーザーセッションに関連するステータスとメ

モリを保護することが可能である。

【０１１５】

サーバー２００の別の実施例において、２又はそれ以上のＰＡＶＭが、一つの処理に取り付けられて、この一つの処理内で実行される。例えば、ＡＢＡＰ ＶＭ及びＪａｖａ（登録商標）ＶＭが一つの処理内で実行されることができる。一つの処理内でＡＢＡＰ ＶＭとＪａｖａ（登録商標）ＶＭを動作させることは、同一のアプリケーションにおいてＡＢＡＰ ＶＭとＪａｖａ（登録商標）ＶＭの両方の使用を可能にする。従って、アプリケーションは、ＡＢＡＰ ＶＭとＪａｖａ（登録商標）環境の両方からの有用なコンポーネントを活用するように開発されることができる。後述する技術は、一つの処理内で複数のＶＭを実行するために使用されることができる。例えば、処理は、Ｊａｖａ（登録商標）ＶＭ、およびＡＢＡＰ ＶＭ、およびＣＬＲ ＶＭを実行するために使用されることができる。

【０１１６】

一つの処理内での複数のＶＭの実行は、コルーチンの使用を通じて達成することができる。ＡＢＡＰ ＶＭとＪａｖａ（登録商標）ＶＭが一つの処理内で動作する例では、二つのコルーチンが使用されることができる－一つはＡＢＡＰ ＶＭのために使用され、もう一つはＪａｖａ（登録商標）ＶＭのために使用される。Ｊａｖａ（登録商標）ＶＭ又はＡＢＡＰ ＶＭの何れかを備えることに加え、各コルーチンは、また、スケジューリングのようなファンクションと、サーバー外部のアプリケーションだけではなく二つのＶＭ間の通信とを処理するために、コンテナー（またはコンテナーの一部）を備えることができる。

【０１１７】

ＡＢＡＰ／Ｊａｖａ（登録商標）の例を続けると、もし、ＡＢＡＰ ＶＭが、アプリケーションが使用するであろう主要なＶＭであれば、ＡＢＡＰ ＶＭを実行するコルーチンは、上述したようなコンテナーを備えることができる。ＡＢＡＰ ＶＭ内で実行するリクエストが最初にＪａｖａ（登録商標）オブジェクトをコールしたとき、そのコンテナーは、Ｊａｖａ（登録商標）ＶＭスタック（Ｊａｖａ（登録商標）ユーザースタックを含む）のためにメモリを割り当て、そして、Ｊａｖａ（登録商標）ＶＭを実行するために新たなコルーチンを開始させる。この新たなコルーチンに受け渡されるパラメータは、新たなコルーチンのためのスタートファンクションだけでなく（この場合、Ｊａｖａ（登録商標）ＶＭそのもの）、割り当てられるメモリのサイズと場所を含む。そして、Ｊａｖａ（登録商標）ＶＭは、メインクラス、例えば、Ｊａｖａ（登録商標）サーブレット、エンタープライズＪａｖａ（登録商標）ビーンズ、またはＪａｖａ（登録商標）サーバーページのような、Ｊ２ＥＥコンポーネントを動作させるために使用できるＪ２ＥＥコンテナーの実施を開始させることができる。それが実行している間、Ｊａｖａ（登録商標）ＶＭは、例えば、メッセージエリアからリクエストを読み出し、そしてレスポンスをメッセージエリアに書き込むためのコンテナー機能性を起動させることができる。スケジューリングのために、Ｊａｖａ（登録商標）ＶＭは、リクエストをメッセージエリアから読み出し、レスポンスをメッセージエリアに書き込んだ後、制御をＡＢＡＰ ＶＭに戻すことができる。そして、Ｊａｖａ（登録商標）ＶＭは、新たなリクエストまたはそれ自体のリクエストの何れかが到着すると、すぐに、再スケジューリングされることができる。

【０１１８】

図４は、図２に示されるサーバー実施例においてリクエストを処理するための処理４５０の例を説明するためのフローチャートである。処理４５０は、処理取り付け可能なＶＭの使用を通じて拡張可能な方法でユーザーセッション間の分離を提供する。ユーザーセッションが開始すると、そのユーザーセッションのために、処理取り付け可能なＶＭが生成されて初期化される(４５２)。このオペレーションは、ＶＭ用のメモリのブロックの割り当てを含む。割り当てられたメモリは、（ＶＭがサーバーにおける複数の処理によってアクセスされることを可能にするための）共有メモリであることができ、また、プライベートとして指定されることができ、つまり、メモリのブロックが新たに生成されたＶＭに単

独で属する。

【０１１９】

メモリのブロックがＶＭに割り当てられた後、ＶＭは、メモリブロックに格納されることができる。（ＶＭスタック及びヒープを含む）ＶＭの計算ステート(computational state)と、（ユーザースタックとヒープを含む）対応ユーザーセッションのためのユーザーコンテキストは、全てメモリブロックに格納されることができる。ユーザーコンテキストは、ファイルのようなＩ／Ｏ資源に対するハンドルと、ソケット(socket)のようなＩ／Ｏ資源のためのプロキシに対するハンドル（例えば、資源マネージャ）を含む。

【０１２０】

ＶＭを初期化することは、システムクラスにおける多数のスタティックイニシャライザーの実行だけでなく、ロード(loading)、検証(verifying)、多くのクラス（例えば、Ｊａｖａ（登録商標）システムクラス）の分解(resolving)を起動するので、高価なオペレーションである。このような初期化オーバーヘッドは、予め初期化された“マスター”ＶＭの使用を通じて低減されることができる。スクラッチ(scratch)からの新たなＶＭを初期化するよりも、むしろ、マスターＶＭのメモリブロックは、新たなＶＭのメモリブロックに単にコピーされることができる。マスターＶＭのメモリブロックのテンプレートイメージを新たなＶＭのメモリブロックにコピーすることは、既に初期化されたステートで新たなＶＭが動作を開始することを可能にする。もし、或る初期化オペレーションが、新たなＶＭが実際に起動したときに実行されるのみであれば、部分的に初期化されたＶＭのテンプレートイメージは使用されることができ、従って、起動した後、すぐに、新たなＶＭは、起動の最中に実行される必要のあるそれらのオペレーションを実行すればよいだけである。

【０１２１】

（実行と同様に）初期化オーバーヘッドは、他の最適化：全てのＶＭによってアクセスされることができる共有メモリのセクションにタイプ情報（例えば、ロードされたクラスのランタイム表現）を格納することを通じてさらに低減することができる。この技術は、クラスローディング、検証、各ＶＭによって負担された分解能(resolution)のためのオーバーヘッドを低減することができ、そして、あらゆるユーザーコンテキストによって使用されそうなシステムクラスのバイトコードを共有するために使用されれば、特に有用である。類似の技術は、ユーザークラスのバイトコードを共有するために使用されることができる。また、コンパイルされたコードは、ジャストインタイム（ＪＩＴ）コンパイラーが使用される実施例で共有されることができる。

【０１２２】

ＶＭが初期化された後、ＶＭは、対応するユーザーセッションからのユーザーリクエストを処理するために使用されることができる。対応するユーザーセッションからのユーザーリクエストが受信されると（４５４）、サーバーに割り付けられた処理のプールからの利用可能な処理が、リクエストを処理するために選択される（４５６）。そして、リクエストを送ったユーザーセッションのＰＡＶＭは、選択された処理にバインドされる（４５８）。

【０１２３】

もし、ＶＭを格納するために使用されるメモリブロックが、サーバーの全ての処理によってアクセスされることができる共有メモリであれば、バインドは、実質的にノンオペレーションであることができる－例えば、メモリブロックまたはその一部が、選択された処理のアドレス空間に単にマッピングされることができる。マッピングされた部分は、ＶＭの計算ステート(computational state)を含むことができる。或いは、ＶＭは、別の方法で存続されないようにできる。例えば、ＶＭの計算ステートはファイルからコピーされることができる。しかしながら、このようなオペレーションの性能は悪く、特に、共有メモリを処理アドレス空間にマッピングする効率的オペレーションと比較して悪く、それは、通常、データのコピーや移動を必要としない。

【０１２４】

選択された処理にＶＭがバインドされた後、ＶＭは、ユーザーリクエストを処理するために処理によって実行されることができる（４６０）。このことは、ＶＭ内のスレッドを模擬するための２又はそれ以上のコルーチンの実行を起動することができる。また、それは、内部スレッド（例えば、ガーベッジコレクションを実行するためのスレッド）の実行を模擬するための同期ファンクションコールの実行を起動することができる。もし、このような内部スレッドまたは他のファンクションが非同期に実行されれば、サーバーに割り当てられた処理のうちの一つが、優先度の低い処理として指定され、そして、ＶＭのためのファンクションを実行するために使用される。この技術は、例えば、ＶＭにおけるガーベッジコレクションを実行するために使用されることができる。さらに重要なファンクションは、優先度が通常の処理または優先度が高い処理の何れかにおいて動作するようにスケジューリングされることができる。

【０１２５】

ＶＭがユーザーリクエストを処理した後、ＶＭは、選択された処理から取り外される（４６２）。そして、この選択された処理は、サーバーによって受信される新たなリクエストを処理するために使用されることができるように、利用可能な処理のプールに戻されることができる（或いは、使用可能であるようにマークされる）。バインドと同様に、処理からＶＭを取り外すことは、単純で低コストのオペレーションである：共有メモリのＶＭのブロックは、単に処理アドレス空間からマッピング解除される。或いは、ＶＭを存続させることは、ファイルにＶＭの計算ステートをセーブするなどのように、より複雑で高価なオペレーションを起動してもよい。ＶＭを処理にバインドすること又は取り付けることと、その処理からＶＭをバインド解除すること又は取り外すことは、お互いに鏡像の関係にある－すなわち、ＶＭを存続させないために実行されるオペレーションは、ＶＭを存続させることと逆のオペレーションである。そして、サーバーは、ユーザーセッションからの他のリクエストを待つことができる（４５４）。

【０１２６】

ユーザーセッションが終了すると、その対応するＶＭもまた終了され、そして、その割り当てられたメモリを含むその資源の全てが開放されることができる。もし、ＶＭのメモリが、ユーザーセッションの終わりで開放されれば、ＶＭが存在する間にガーベッジコレクションを省略(omit)することが可能である。ユーザーセッションの終わりでＶＭを終了させることの代替手段は、ＶＭを再使用することであり、すなわち、ＶＭを他のユーザーセッションと関連づけ、そして、そのユーザーセッションに対応するリクエストを処理するためにそのＶＭを使用することである。この技術は、保持されるのに非常に小さなユーザーコンテキスト（またはユーザーコンテキストなし）を必要とするアプリケーションにとっては特に有用であり、非常に短いユーザーセッションを有するアプリケーションについても同様である。（保持されるのにユーザーコンテキストを必要としないアプリケーションは、時折、ステートレスアプリケーションと呼ばれる。）このような全てのアプリケーションについて、連続的に再使用可能なＶＭのプールを使用することは、ＶＭを生成し、初期化し、保持し、終了することに関連したオーバーヘッドを最小化するのに役立つことができる。

【０１２７】

処理取り付け可能なＶＭの使用と実施例に関する追加的な詳細は、N.Kuck, H.Kuck, E.Lott, C.Rohland, and O.Schmidt, SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers(August 1, 2002)(非公開の要約が提出され、そして2nd USENIX Java（登録商標）仮想マシン研究技術シンポジウム（Ｊａｖａ（登録商標）ＶＭ‘０２）に途中報告として示されている)において述べられている。

【０１２８】

共有オブジェクトの生成および使用(Creating and Using Shared Object)

【０１２９】

各ユーザーセッションに一つのＶＭを割り当てることは、各ユーザーセッションは、そ

れ自身のＶＭヒープ（例えばＪａｖａ（登録商標）ヒープ）を備えることを意味する。従って、シングル、マルチスレッドＶＭを使用するサーバー２００の実施例において一度だけ存在するだけのＶＭヒープにおけるデータオブジェクトは、各ユーザーセッションのために一度だけ複製(replicate)される。このことは、資源の消費（例えば、複製されたデータオブジェクトを格納するためのメモリ）と時間の消費（例えば、複製されたデータオブジェクトを構築して初期化するための時間）の両方をもたらす。

【０１３０】

簡単な例として、アプリケーションの起動時に解析される拡張マークアップ言語（ＸＭＬ）構成ファイル(configuration file)を表す大規模なＪａｖａ（登録商標）データ構造を考える。上記アプリケーションに対応する各ユーザーのためのこのようなデータ構造の複製処理において、サーバー２００は、ＣＰＵ時間（ＸＭＬファイルを解析し、データ構造を構築するための時間）とメモリ（データ構造を格納するためのメモリ）の両方を第１のＶＭとユーザーセッションとを除く全てのために消費する。

【０１３１】

この問題を解決するため、サーバー２００の一実施例は、データオブジェクトがＶＭ間で共有されることを可能にする。サーバー２００のこの実施例において、共有メモリ領域またはヒープが、複数のＶＭによってアクセス可能なデータオブジェクトを格納するために使用される。

【０１３２】

共有メモリヒープにおけるデータオブジェクトは、通常、如何なるプライベートヒープ（例えば、個々のＶＭのプライベートヒープ）に対しても参照(references)又はポインタ(pointers)を有しない。これは、もし、共有メモリヒープにおけるオブジェクトが一つの特定のＶＭにおけるプライベートオブジェクトに対する参照と共にメンバー変数を有しているとすれば、その参照は、その共有オブジェクトを使用する他の全てのＶＭにとって無効(invalid)であるからである。さらに正式には、この制限は次のように考えられる：何れの共有オブジェクトについても、初期オブジェクトによって参照されるオブジェクトの推移的クロージャ(transitive closure)は、常に共有オブジェクトを含むだけである。

【０１３３】

従って、サーバー２００の一実施例において、オブジェクトは、それ自身によって共有メモリに格納される－正確に言えば、オブジェクトは、“共有クロージャ”として知られるグループにおける共有メモリヒープに格納される。

【０１３４】

共有クロージャを通じたオブジェクトの共有は、図５に概念的に示され、同図において、共有クロージャ６００は、第１のオブジェクト６０１によって参照される全てのオブジェクトの推移的クロージャと共に第１のオブジェクト６０１をグループ化することにより第１のＶＭ３０１において識別(identify)される。共有クロージャ６００が識別された後、ＶＭ３０１は、例えば、共有クロージャ６００を共有メモリ領域２５５にコピーすることにより、共有メモリ領域２５５に共有クロージャを生成することができる。共有クロージャ６０１が共有メモリ領域２５５に生成された後、それは、サーバー（例えば、ＶＭ３０１，３０３，３０５）におけるＶＭによってアクセスされることができる。ＶＭは、例えば、共有メモリ領域２５５からＶＭが稼動している処理のアドレス空間に共有クロージャ６００をマッピングまたはコピーすることにより、共有メモリ領域２５５から共有クロージャ６００をアクセスすることができる。共有クロージャの生成及び使用は、以下において更に詳細に説明される。

【０１３５】

共有クロージャ内で使用可能であるためには、オブジェクトは、“共有可能”でなければならない。通常、一つのランタイムシステム（例えばＪａｖａ（登録商標）ＶＭ）における複合データ構造体（例えば、ヒープ、またはその一部）は、もし、データ構造の内部の一貫性または機能性を阻害することなく、データ構造を分解(disassemble)し、そして第２のランタイムシステムの本来のフォーマットで再構成(reassemble)することができれ

ば、第２のランタイムシステムで共有されることができる。

【０１３６】

サーバー２００の一実施例において、オブジェクトは、共有クロージャを共有メモリにコピーし又は該共有メモリからコピーすることを通じて共有される。この実施例においてオブジェクトが共有可能であるためには、オブジェクトは、オブジェクトの機能性または内部的一貫性を阻害することなく、他のＶＭのアドレス空間に対する透過的ディープコピー(transparent deep-copy)に耐えることができなければならない。このような実施例のための共有可能性要件(shareability requirement)、は、以下で更に詳しく説明される。

【０１３７】

通常、共有可能性のほとんどの態様はオブジェクトのクラスのプロパティであるにもかかわらず、オブジェクトインスタンスの共有可能性(shareability)は、このクラスのプロパティに依存しなのみならず、そのオブジェクトインスタンスのメンバー変数のタイプにも依存しない。メンバー変数が、実行時まで決定できないランタイムタイプを有することができる場合、共有クロージャ内のオブジェクトインスタンスの共有可能性は実行時に決定されなければならない。

【０１３８】

従って、オブジェクトインスタンスがランタイムタイプを有するサーバー実施例において、共有可能なクラスと共有可能なオブジェクトインスタンスとの間に区別が設けられる。クラスは、以下に例が述べられるような共有可能性の基準(criteria)を満足すれば、共有可能なクラスである。オブジェクトインスタンスは、そのランタイムタイプが共有可能である場合、および、参照する全てのオブジェクトが共有可能なオブジェクトインスタンスである場合に、共有可能なオブジェクトインスタンスである。換言すると、オブジェクトインスタンスは、次の両方の条件が満たされれば、共有可能なオブジェクトインスタンスである：（ｉ）オブジェクトのランタイムクラスが共有可能なクラスであること、および（ｉｉ）オブジェクトインスタンスのゼロでない参照タイプメンバー変数が共有可能なオブジェクトインスタンスであること。

【０１３９】

第１条件（オブジェクトのランタイムクラスが共有可能なクラスであること）は、ランタイムクラスのインスタンスが共有に対処することが意味的(semantically)に可能であることを確保することを意味している。クラスが共有可能であるかどうかを判定するための基準の例が以下に提示される。ランタイムクラスが共有可能であるかどうかの判定は、各クラスについて一度だけなされればよいが、このような特性は、引き継ぐことができない。なぜなら、導出クラス(derived class)は、共有と互換性のない機能性を付け加えるからである。

【０１４０】

第２条件（オブジェクトインスタンスのゼロでない参照タイプメンバー変数が、それ自体、共有可能なオブジェクトインスタンスであること）は、共有クロージャにおける全てのオブジェクトが共有可能であることを確保することを意味している。この条件が満足されるか否かは、オブジェクトインスタンスにおける参照の再帰的検査によって判定されることができる。“共有可能なクラス”特性の非引継性(non-inheritability)のために、オブジェクトの全てのメンバー変数の宣言されたタイプを単に検査することは、宣言されたタイプが共有可能であり、ランタイムタイプが共有可能でないとしても、十分ではない。

【０１４１】

図４は、オブジェクトインスタンスが共有可能なオブジェクトインスタンスであるかどうかを判定するために使用できる処理６５０の例のフローチャートを示す。図６に示される処理において、最初に、オブジェクトインスタンスの識別が受信される（６５２）。オブジェクトインスタンスのランタイムクラスが識別され、そしてこのランタイムクラスが共有可能なクラスであるかどうかの判定が行われる（６５４）。もし、ランタイムクラスが共有可能なクラスでなければ（判定６５６のブランチ“ｎｏ”）、処理は、オブジェクトインスタンスが共有可能でない旨の指示(indication)で終わる（６５８）。このような

否定的な指示は、例えば、“共有不能(not shareable)”の例外を起こすことによりなされることができる。

【０１４２】

もし、オブジェクトインスタンスのランタイムクラスが共有可能なクラスであれば（判定６５６のブランチ“ｙｅｓ”）、オブジェクトインスタンスによって参照されるオブジェクトが識別される（６６０）。そして、処理６５０は、参照されるオブジェクトをトラバース(traverse)して、その参照されるオブジェクトが共有可能なオブジェクトインスタンスであるかどうかを判定する。もし、参照されるオブジェクトがさらに存在すれば（判定６６２のブランチ“ｙｅｓ”）、残りの参照されるオブジェクトの一つが選択される（６６４）。そして、参照されるオブジェクトが共有可能なオブジェクトインスタンスであるかどうかの判定がなされる（６６６）。もし、参照されるオブジェクトが共有可能なオブジェクトインスタンスでなければ（判定６６８のブランチ“ｎｏ”）、処理６５０は、初期オブジェクトインスタンスが共有可能でない旨の指示(indication)で終わる（６５８）。これは、初期オブジェクトインスタンスによって参照されるオブジェクトの一つが共有可能なオブジェクトインスタンスではないためであり、そして、前述したように、オブジェクトが共有可能であるためには、初期オブジェクトによって参照されるオブジェクトの全てが共有可能なオブジェクトインスタンスでなければならないからである。

【０１４３】

もし、参照されるオブジェクトが共有可能なオブジェクトインスタンスであれば（判定６６８のブランチ“ｙｅｓ”）、処理６５０は、分析すべき参照されるオブジェクトがさらに存在するかどうかを把握するためにチェックする。もし、参照されるオブジェクトがさらに存在すれば（判定６６２のブランチ“ｙｅｓ”）、処理６５０は、残りの参照されるオブジェクトの一つを選択し、そしてこれまでのように処理を進める。もし、もはや参照されたオブジェクトが存在せず（判定６６２のブランチ“ｎｏ”）、且つ、処理がまだ終了していなければ、それは、参照されるオブジェクトの全てが分析され、そして共有可能なオブジェクトインスタンスであると判定されたことを意味する。従って、処理は、初期オブジェクトインスタンスが共有可能である旨の指示で終了する（６７０）。

【０１４４】

参照されるオブジェクトが共有可能なオブジェクトインスタンスであるかどうかの判定は、再帰的になされる－すなわち、処理６５０は、図６の破線で示されるように、参照されるオブジェクトで再び起動されることができる。換言すれば、参照されるオブジェクトの推移的クロージャにおける各オブジェクトに関して判定がなされるように、参照されるオブジェクトが再帰的にトラバースされることができる。

【０１４５】

もし、参照されるオブジェクトの推移的クロージャにおける全てのオブジェクトと初期オブジェクトが共有可能なオブジェクトインスタンスであれば、このオブジェクトは、共有クロージャ(shared closure)にグループ化され、そして、他のランタイムシステムで共有されることができる（例えば、共有メモリ領域に共有クロージャをコピーすることにより）。

【０１４６】

結局のところ、処理６５０は、共有クロージャにおける各オブジェクトのランタイムクラスが共有可能なクラスであるかどうかを判定するものと考えることができる。前述したように、共有クロージャを共有メモリにコピーすることと共に該共有メモリからコピーすることを通じてオブジェクトが共有するところの実施例では、通常、オブジェクトは、オブジェクトの機能性と内部的な一貫性を阻害することなく、他のＶＭのアドレス空間への透過的ディープコピーに耐えることができれば、共有可能であると見なされる。このような実施例では、通常、クラスのオブジェクトインスタンスの直列化(serialize)又は非直列化(deserialize)において、ＶＭが如何なるカスタムコードも実行しなければ、クラスは共有可能であると見なされる。このルールの理論的根拠は、もし、ＶＭが如何なるカスタム直列化または非直列化コードを実行する必要がなければ、共有クロージャを共有ヒー

プにコピーするために（または共有ヒープからＶＭのアドレス空間にコピーするために）使用されるディープコピーオペレーションが、意味的に、共有クロージャにおけるオブジェクトの直列化および非直列化と等価である、ということにある。従って、もし、共有クロージャが共有ヒープにコピーされていれば、共有クロージャを自身のアドレス空間にマッピングまたはコピーする任意のＶＭは、オブジェクトを非直列化するために必要な追加的アクションを何ら要することなく、共有クロージャにおけるオブジェクトをアクセスすることができるべきである。

【０１４７】

図７は、特定のクラスが共有可能なクラスであるかどうかを判定するために使用できる処理７５０の例のフローチャートを示す。処理７５０は、例えば、特定のランタイムクラスの共有可能性に関してなされる判定(６５４)を呼び出す場合に使用されることができる。図７に示される処理では、まず、クラス（例えば、オブジェクトインスタンスのランタイムクラス）の識別が受信される（７５２）。そして、クラスが共有可能かどうかを判定するために、多数の基準が適用される。具体的には、クラスは、共有可能であるためには次の条件の全てを満足しなければならない。

【０１４８】

第１に、クラスは、直列化可能でなければならない（７５４）。Ｊａｖａ（登録商標）クラスの場合、これは、クラスがマーカーインターフェイスjava（登録商標）.io.Serializableを実施するかどうかをチェックすることにより判定することができる。このjava（登録商標）.io.Serializableインターフェイスの実施例は、通常、クラスのオブジェクトインスタンスが有意義に他のアドレス空間にコピーされることを示す。従って、もし、クラスがjava（登録商標）.io.Serializableインターフェイスを実施すれば、第１条件が満足される。

【０１４９】

第２に、クラスは、如何なるカスタム直列化または非直列化コードも含んではならない（７５６）。Ｊａｖａ（登録商標）クラスの場合、これは、クラスが次のメソッドの何れかを実施するかどうかによって判定できる。

```
Private void readObject(ObjectInputStream);
Private void writeObject(ObjectOutputStream);
Private void readExternal(ObjectInput);
Private void writeObject(Objectoutput);
{any access} object readResolve( );
{any access} object writeReplace( );
```

【０１５０】

上述のメソッドは、直列化または非直列化の期間中に実行されるカスタムコードを構成する。このようなカスタムコードは、共有クロージャの生成中に実行されるディープコピーオペレーションと自動的に等価であることにはならない。従って、ディープオペレーションが共有クロージャを生成するために使用される場合、上述の何れかのファンクションの実施は、クラスが、処理７５０において共有可能なクラスと自動的に見なされることを排除する。

【０１５１】

第３に、未解決のクラスの全てのベースクラス(base classes)は直列化可能でなければならない（７５８）。Ｊａｖａ（登録商標）クラスの場合、これは、全てのベースクラスが、java（登録商標）.io.Serializableを実施するか、或いは普通のデフォルトコンストラクター(default constructor)を有するかどうかをチェックすることにより判定されることができ－もしそうであれば、第３条件は満足される。もし、どのベースクラスもjava（登録商標）.io.Serializableを実施しなければ、そのデフォルトコンストラクターは、非直列化の期間中に実行される。もし、デフォルトコンストラクターが普通(trivial)であれば－すなわち、もし、コンストラクターがベースクラスの普通のデフォルトコンストラクターを起動するか又は空(empty)であれば、それはデフォルトコンストラクターの再

帰的検査を通じて判定されることができる－デフォルトコンストラクターの起動は、何ら非直列化に影響を及ぼさない。非通常(non-trivial)のデフォルトコンストラクターは、クラスが処理７５０において共有可能なクラスであると自動的に見なされることを防ぐ。なぜなら、デフォルトコンストラクターは、ディープコピーオペレーションと等価ではないカスタムコードを含んでもよいからである。

【０１５２】

第４に、問題としているクラス(class at issue)の全てのメンバーフィールドは直列化されなければならない（７６０）。Ｊａｖａ（登録商標）クラスの場合、このことは、クラスが、何らかのトランジェントフィールドまたはserialPersistentFieldsフィールドを有するかどうかをチェックすることにより判定されることができる。トランジェントフィールドは、非直列化の間にそれらのデフォルト値に設定されるフィールドである。従って、トランジェントフィールドでクラスのオブジェクトインスタンスを非直列化することは、オブジェクトインスタンスのディープコピーと等価ではない。従って、クラスにおけるトランジェントフィールドの存在は、クラスが処理７５０において共有可能なクラスであると自動的に見なされることを防ぐ。serialPersistentFieldsフィールドを有するクラスもまた排除される。なぜなら、このようなクラスは、トランジェントフィールドでクラスを示す他の方法であるに過ぎないからである。

【０１５３】

第５に、クラスは、何らかのガーベッジコレクションの副次的効果(garbage collection side effects)を有しなければならない（７６２）。共有されるオブジェクトは、それを使用するＶＭのライフサイクルとは異なるライフサイクルを有してもよく、従って、ＶＭ内で実行するガーベッジコレクションアルゴリズムに影響を与えてもよい。ガーベッジコレクションの副次的効果は、クラスが処理７５０において共有可能なクラスであると自動的に見なされることを防ぐ。なぜなら、副次的効果は、ガーベッジコレクションアルゴリズムの適切なオペレーションを妨ぐからである。Ｊａｖａ（登録商標）クラスの場合、処理７５０は、クラスが普通の終了化子(finalizer)を有することをチェックし、そしてクラスがclass java（登録商標）.lang.ref.Referenceから得られたものでないことをチェックすることにより、この第３条件が満足されると判定することができる。普通の終了化子は、ベースクラスの普通の終了化子を起動する終了化子、又は空にする終了化子である。

【０１５４】

もし、上述した５つの全ての条件が満足されれば、処理７５０は、問題としているクラスが共有可能なクラスであることの指示で終了する（７６６）。一方、もし、何れかの条件が満足されなければ、処理７５０は、問題としているクラスが共有可能なクラスでないことの指示で終了する（７６４）。

【０１５５】

サーバー２００の一実施例において、もし、自動的に適用される処理（例えば処理７５０）を通じてクラスが共有可能であることが分かれば、または、もし、クラスが共有可能であると予め宣言されていれば、そのクラスは共有可能である見なされる。すなわち、たとえ、自動的に適用されたクラスの分析が、そのクラスが共有可能であることを示すことができなくても、そのクラスは共有可能であるかもしれない。

【０１５６】

もし、クラスが（例えばそのソースコードを手動で調査することにより）検査されて、そして共有に適していることが判明すれば、そのクラスは共有可能であると宣言されることができる。例えば、共有クロージャを共有メモリにコピーし、該共有メモリから共有クロージャをコピーすることを通じてオブジェクトが共有される実施例において、もし、意味検査(semantic inspection)が、上述した共有可能性基準の全ての違反が問題とならないことを立証すれば、クラスは共有に適している。共有可能性基準の違反にもかかわらず、もし、共有クロージャを共有ヒープにコピーするために（または、共有ヒープからＶＭのアドレス空間にコピーするために）使用されるディープコピーオペレーションが、共有

クロージャにおけるオブジェクトの直列化および非直列化と意味的に等価であることが示されれば、通常、これらの違反は問題にならない。

【０１５７】

上述した共有可能性基準を満足しないが共有には適しているクラスの一つの簡単な例は、class java（登録商標）.lang.Stringである（そのクラスは、Ｊａｖａ（登録商標）２プラットフォーム、標準版１．３において規定されている）。java（登録商標）.lang.Stringクラスは、上述した第４条件に違反する。なぜなら、それは、serialPersistentFieldsフィールドを含んでいるからである。クラスにおけるコードの手動の調査は、直列化の期間中にクラスのオブジェクトインスタンスの特別な処理を実施するためにそのフィールドが含まれていることを示す。そしてそれは、直列化プロトコルの要件である。にもかかわらず、ディープコピーの効果がクラスのための直列化と等価であることが容易に示される。したがって、java（登録商標）.lang.Stringクラスは、共有可能であると宣言されることができる。

【０１５８】

上述の共有可能性基準を満たさないが共有に適した更に複雑な例は、class java（登録商標）.util.Hashtable（そのクラスは、Ｊａｖａ（登録商標）２プラットフォーム、標準版１．３で規定されている）である。java（登録商標）.util.Hashtableクラスは、上述の第２および第４条件に違反する。なぜなら、それは、カスタム直列化メソッドおよびトランジェントフィールドを含むからである。そのクラスにおけるコードのレビューは、カスタム直列化メソッドとトランジェントフィールドが必要とされることを示している。なぜなら、ハッシュコード(hushcodes)は、直列化中に保存されず、それは、直列化中にハッシュテーブルにそれらのコンテンツを再構成させるからである。しかしながら、ディープコピーオペレーションはハッシュコードを保存するので、ディープコピーオペレーションは、直列化及び非直列化と等価であることが示されることが出来る。結局、class java（登録商標）.util.Hashtableは、また、共有可能であると宣言されることができる。

【０１５９】

図５に概念的に示される共有クロージャの生成および使用は、図８および図９に示される処理８５０および処理９５０により実現できる。

【０１６０】

処理８５０は、共有クロージャを生成するために使用できる処理の例を示す。処理８５０において、第１のランタイムシステム（例えば、ＶＭ）における初期オブジェクトの識別が受信される（８５２）。そして、共有クロージャ－すなわち、初期オブジェクトによって参照される全てのオブジェクトの推移的クロージャと初期オブジェクト－が識別され（８５４）、そして共有クロージャが他のランタイムシステム（例えば、他のＶＭ）によって共有され、または他のランタイムシステムで使用されることができるかどうかに関して判定がなされる（８５６）。この判定は、例えば、共有クロージャにおけるオブジェクトが共有可能であるかどうかを判定することにより（あるいは、より正確には、共有クロージャにおける各オブジェクトインスタンスが共有可能なオブジェクトインスタンスであるかどうかを判定することにより）、行われる。一実施例において、共有クロージャを識別し、そして共有クロージャにおけるオブジェクトが共有可能なオブジェクトインスタンスであるかどうかを判定する（８５４，８５６）ためのオペレーションは、図６に示される処理６５０によって実施される。

【０１６１】

もし、共有クロージャが他のランタイムシステムにおいて使用可能でなければ（判定８５８のブランチ“ｎｏ”）、処理８５０は、例外(exception)を引き起こし、または、或るタイプの否定的な指示を発生させる。例えば、もし、共有クロージャにおけるオブジェクトが全て共有可能なオブジェクトインスタンスでなければ、処理は、初期オブジェクトとその共有クロージャが共有可能でないことを示す例外を引き起こす。

【０１６２】

もし、共有クロージャが他のランタイムシステムにおいて使用可能であれば（判定８５

８のブランチ“ｙｅｓ”）、処理８５０は、共有クロージャを他のランタイムシステムに利用可能とするためのメカニズムを起動する。例えば、もし、オブジェクトが共有メモリの使用を通じて共有されれば、共有クロージャは、共有メモリ領域にコピーされることができる（８６２）。他の実施例において、共有クロージャは、メッセージまたは他の通信手段の使用を通じて１又は２以上のランタイムシステム（例えば、他のＶＭ）に伝送されることができる。

【０１６３】

また、共有クロージャを生成する処理は、特定のネーム(name)または他の識別子(identifier)を共有クロージャに関連づけることができる（８６４）。その後、このような識別子は、アクセスされるべき共有クロージャを識別するために他のランタイムシステムによって使用されることができる。

【０１６４】

いくつかの実施例では、また、共有クロージャを生成する処理は、バージョニング(versioning)の使用を起動する。処理８５０において、バージョニングは、共有メモリに格納された共有クロージャに関連づけられたバージョンナンバー(version numbers)の使用を通じて行われる。共有クロージャが所定のネームで生成されれば、そのネームを有する共有クロージャが共有メモリに既に存在するかどうかに関して判定がなされる。もし、このようなクロージャが存在すれば（判定８６６のブランチ“ｙｅｓ”）、共有クロージャに関連する現在のバージョンナンバーが増加され（８６８）、そして新たな現在のバージョンナンバーが、新たに生成された共有クロージャと関連づけられる（８７２）。もし、共有メモリに所定のネームを有する共有クロージャが存在しなければ（判定８６６のブランチ“ｎｏ”）、新たな共有クロージャのための現在のバージョンナンバーは、最初のバージョンを示すナンバー（例えば、０または１）に設定され（８７０）、そして、新たに生成された共有クロージャと関連づけられる（８７２）。

【０１６５】

バージョニングは、共有クロージャを更新するために使用されることができる－例えば、共有クロージャの新たな更新されたバージョンは、共有クロージャに対して前に与えられたネームと同一のネームで生成されることができる。一実施例において、ネームが付された新たなバージョンの共有クロージャが生成された場合、この新たなネームが付された共有クロージャをＶＭに関連づけるためのその後の全てのオペレーションは、新たなバージョンの共有クロージャを使用する。既に共有クロージャをアクセスしているＶＭ（例えば、それらのアドレス空間にマッピングされた共有クロージャの前のバージョンを有するＶＭ）は、新たなバージョンに影響されない－それらは、単に、古いバージョンに対する全てのオブジェクト参照を保持する。この実施例では、複数のバージョンの共有クロージャは、もはや古いバージョンが如何なるＶＭにも参照されなくなるまで、共有メモリに共存することができ、従ってガーベッジコレクションが行われることができる。

【０１６６】

図９は、共有クロージャをアクセスし使用するための処理例のフローチャートを示す。処理９５０において、ネーム又は他の識別子が受信され（９５２）、そして、関連する共有クロージャを識別するために使用される（９５４）。上記ネーム又は他の識別子は、共有クロージャが生成されたときに該共有クロージャに関連づけられた識別子に対応する（例えば処理８５０のオペレーション８６４）。バージョニングが実施される場合、もし、２以上のバージョンの特定の共有クロージャが存在すれば、ごく最近生成されたバージョンの特定の共有クロージャが識別される。

【０１６７】

そして、識別された共有クロージャは、ランタイムシステム（例えば、ＶＭ）と関連づけられる（９５６）。一実施例において、共有クロージャは、次の２つのうちの一つの方法によりランタイムシステムと関連づけられることができる－一つは、共有メモリ領域からランタイムシステムのアドレス空間に共有クロージャをマッピングすることであり、もう一つは、共有メモリ領域からランタイムシステムのアドレス空間に共有クロージャをコ

ピーすることである。共有クロージャがランタイムシステムと関連づけられた後、共有クロージャ内のオブジェクトは、通常のオペレーション（例えば、通常のＪａｖａ（登録商標）オペレーション）を用いてアクセスされることができる（９６２）。

【０１６８】

いくつかの実施例では、共有クロージャにおけるオブジェクトへのアクセスは、この共有クロージャがランタイムシステムにどのように関連づけられているかに依存してもよい。例えば、一実施例において、もし、共有クロージャがＶＭのアドレス空間にマッピングされれば（判定９５８のブランチ“ｍａｐｐｅｄ”）、共有クロージャにおけるオブジェクトへのアクセスは、リードのみのアクセスに制限される（９６０）。この制限のため、共有クロージャにおける多くのオブジェクトインスタンスに対するライトの試行はエラーに終わるであろう。この制限は、共有オブジェクトインスタンスにおける参照ナンバー変数をＶＭのプライベートヒープへの参照でオーバーライトすることにより、或いは逆に、共有オブジェクトの機能性または内部的一貫性をブレークすることにより、ＶＭが共有オブジェクトインスタンスを“ブレーク(brake)”することを防止するのに有用である。

【０１６９】

他方、もし、共有クロージャがＶＭのアドレス空間にコピーされれば（判定９５８のブランチ“ｃｏｐｉｅｄ”）、ＶＭは、コピーされたオブジェクトに対する完全なリード－ライトが認められる。このような実施例では、従って、共有クロージャにおけるオブジェクトは、共有クロージャをＶＭのアドレス空間にコピーし、共有クロージャにおけるオブジェクトのコンテンツを修正し、そして、共有メモリにおける新たなバージョンの共有クロージャを生成することにより（例えば、図８に示される処理８５０を用いて）更新されることができる。

【０１７０】

他のアプローチは、共有クロージャをランタイムシステムと関連づけるために使用されることができ、そして、ランタイムシステムから共有クロージャにおけるオブジェクトに対するアクセスを提供するために使用されることができる。例えば、コピーオンデマンドアプローチ(copy-on-demand approach)が使用されることができる。このような一実施例では、共有クロージャは、共有クロージャへのアクセスをリードのみのアクセスに制限することなくＶＭのアドレス空間にマッピングされる。その代わり、共有クロージャへのアクセスはモニターされ、そして、共有クロージャに対する最初の試行ライトアクセスを検出すると、すぐに、共有クロージャは、ＶＭのアドレス空間にコピーされ、これにより、マッピングされた共有クロージャからコピーされた共有クロージャに共有クロージャを伝送する。そして、試行ライトアクセスは、完了することを許され、そしてその後、コピーされた共有クロージャに対するリード及びライトアクセスが通常どおり続行する。もし、マッピングされた共有クロージャから、コピーされた共有クロージャへの伝送が発生したときに、共有クロージャのヒープアドレスが変化すれば、ヒープに対する既存の参照が、新たに生成された共有クロージャのコピーにリダイレクト(redirect)されなければならない。或いは、基礎をなすＯＳ特性(underlying OS feature)は、ヒープアドレスが変化することなくＯＳがコピーオンデマンド機能性を提供することを可能にする方法で、共有クロージャをマッピングするために使用されることができる。

【０１７１】

共有クロージャを生成し、マッピングし、コピーするための機能に加えて、ＡＰＩ(Application Programming Interface)は、共有オブジェクトを管理するための追加的機能を含むことができる。例えば、ＡＰＩは、また、“デリート(delete)”機能を含むこともできる。“デリート”機能の一実施例は、ネーム又は他の識別子を入力パラメータと見なし、そして共有メモリにおける関連する共有クロージャを、デリートされたものとしてマークする。デリートされたものとして共有クロージャをマークすることは、共有クロージャを既にアクセスしているＶＭに影響を与えないが、その後に（例えば、共有クロージャをそれらのアドレス空間にマッピングまたはコピーすることにより）共有クロージャをアクセスしようとするＶＭは、そうすることを阻止される。

【０１７２】
共有クロージャがＶＭのアドレス空間にマッピングされることができるサーバー２００の実施例では、デリートされた又は古いバージョンの共有クロージャのためのガーベッジコレクションは、アドレス空間に共有クロージャがマッピングされたＶＭの数の経過を追うことにより実施されることができる。ＶＭが共有クロージャをそのアドレス空間にマッピングする度にカウントがインクリメントされることができる。自身のガーベッジコレクションの最中に、ＶＭが、もはやそれが前にマッピングされた共有クロージャへの参照を何ら含んでいないと判定したときに、そのカウントはデクリメントされることができる。特定の共有クロージャと関連する上記カウントがゼロに到達すると、その共有クロージャは共有メモリからデリートされることができる。

【０１７３】
Ｊａｖａ（登録商標）では、オブジェクトインスタンスは、通常、オブジェクトインスタンスのクラスのランタイム表現に対する参照を含み、そしてクラスランタイム表現は、クラス用のクラスローダーに対する参照を含む。従って、Ｊａｖａ（登録商標）オブジェクトインスタンスに関連するクラスローダー及びランタイム表現は、オブジェクトインスタンスの共有クロージャに含まれ、そのことは、オブジェクトインスタンスが共有可能であるためには、ランタイム表現及びクラスローダーが、それ自体、共有可能でなければならないことを意味する。従って、クラスランタイム表現とクラスローダーを含むサーバーの実施例では、二つの追加的基準が、特定のクラスが共有可能であるかどうかを判定するために使用されることができる：クラスは、共有可能なランタイム表現と共有可能なクラスローダーを備えるべきである。

【０１７４】
種々の技術が、クラスランタイム表現及びクラスローダーを取り扱うために（すなわち、クラスランタイム表現およびクラスローダーを“共有可能”とするために）使用されることができる。一つの技術として、クラスランタイム表現とクラスローダーを実際に共有することが挙げられる。すなわち、オブジェクトインスタンスが共有メモリにコピーされたときに、オブジェクトインスタンスのクラスに対応するクラスローダーとランタイム表現は、また、全てのＶＭによってアクセスされることができるように、共有メモリにコピーされる（例えば、オペレーションをマッピングすることを通じて）。この技術の種々の最適化が可能である。例えば、クラスのランタイム表現を共有メモリにコピーする前に、共有メモリは、そのクラスのためのランタイム表現が既に共有メモリに存在しているかどうかを判定するためにチェックされることができ－もし、そうであるならば、共有メモリにコピーされているオブジェクトインスタンスにおけるランタイム表現に対する参照は、単純に、既に共有メモリに存在するランタイム表現を参照するように設定されることができる。

【０１７５】
ランタイム表現及びクラスローダーを取り扱うための第２の技術は、それらを共有するためではなく、それらが、各ＶＭにおける固定場所に位置されることを確実にするためである。換言すれば、各クラスのためのクラスローダー及びランタイム表現は、各ＶＭにおいて同一の固定されたアドレスに位置されなければならない。そして、各オブジェクトインスタンスにおけるランタイム表現は、オブジェクトインスタンスのクラスのためのランタイム表現に対応する場所に設定されることができる。このアプローチを用いれば、ランタイム表現に対する参照は、オブジェクトインスタンスが共有メモリからマッピングされたか、または、アドレス空間にコピーされたかどうかとは関係なく、有効(valid)である。

【０１７６】
しかしながら、各クラスのためのランタイム表現の場所を固定することは、実際的ではないかもしれない。従って、ランタイム表現及びクラスローダーを取り扱うための第３の技術は、オブジェクトインスタンスがＶＭにコピーされたときに各オブジェクトインスタンスのためのランタイム表現に対する参照を調整することである。従前の技術と同様に、

ランタイム表現及びクラスローダーはこの技術で共有される－すなわち、各ＶＭは、各クラスのためのそれ自身のクラスローダー及びランタイム表現を格納する。しかしながら、従前の技術とは違って、この技術は、各ランタイム表現及びクラスローダーの場所が各ＶＭにおいて固定されることを要しない。その代わり、クラスローダー及びランタイム表現の場所は、各ＶＭにおいて異なることができる。オブジェクトインスタンスが特定のＶＭにコピーされると、適切なクラスランタイム表現の場所が決定され、そして、オブジェクトインスタンスにおける対応参照がその場所に設定される。

【０１７７】

ランタイム表現を取り扱うための第３の技術－各ＶＭにおけるランタイム表現に対する参照を調整すること－は、オブジェクトインスタンスが、多数のＶＭに同時にマッピングされることを阻止する。この理由は、前述したように、共有されたオブジェクトインスタンスは、如何なるプライベートヒープへの参照も有することができないためである。この第３の技術は、各ＶＭにおけるプライベートランタイム表現を参照するための参照を調整するので、この技術は、オブジェクトがＶＭのアドレス空間にコピーされるとき、あるいは、オブジェクトが１回に一つのＶＭによってのみアクセスされる他の環境において（例えば、オブジェクトインスタンスが一つのＶＭにマッピングされるときに他のＶＭがそのオブジェクトインスタンスをマッピングしないように、オブジェクトインスタンスが“排他的に”マッピングされることができる環境において）のみ使用されることができる。

【０１７８】

上述の第３の技術は、ＶＭが多数の物理マシン上で実行されることができるところのクラスターアーキテクチャにおいて有用である。ランタイム表現は、通常、物理マシン間で共有されず、したがってランタイム表現に対する参照は、オブジェクトインスタンスが物理マシン間で共有されるときに（例えば、一つの物理マシン上のＶＭにおいて使用されているオブジェクトインスタンスが、そのマシン上のＶＭにおいて使用されるべき第２の物理マシンに伝送されるときに）調整されなければならない。

【０１７９】

図１０は、多数のマシン間でオブジェクトを使用するための技術を示す。この技術は、上述したランタイム表現を取り扱うための第３の技術の一般化されたバージョンであり、ソースランタイムシステムで実行されることができる幾つかのオペレーションと、ターゲットランタイムシステムで実行されることができる幾つかのオペレーションとの組み合わせを含む。図１０において、処理１０００は、ソースランタイムシステムにおいて実行されることができるオペレーションの例を示し、処理１０５０は、ターゲットランタイムシステムにおいて実行されることができるオペレーションの例を示す。通常、この技術は、オンザフライでのスタブ(stubs)の生成および使用を含む－ランタイムメタデータに対する参照は、ソースランタイムシステムにおけるプロキシで置き換えられ、そしてプロキシは、ターゲットランタイムシステムにおけるランタイムメタデータに対する参照で置き換えられる。

【０１８０】

さらに詳しくは、処理１０００において、ターゲットランタイムシステムにおいて使用され又は共有されるべき一組のオブジェクトは、最初に、ソースランタイムシステムにおいて識別される（１００２）。このオペレーションは、例えば、図６の処理６５０のように、ソースランタイムシステムにおけるオブジェクトの共有クロージャの識別と、共有クロージャにおける全てのオブジェクトが共有可能なオブジェクトインスタンスであることの検証を起動する。しかしながら、このオペレーションは、また、非常にシンプルである－例えば、単一のオブジェクトが、ターゲットランタイムシステムにおいて使用されるために選択されることができる。（しかしながら、単一のオブジェクトが選択された場合、例えば参照されるオブジェクトを両方のランタイムシステムにおいて同一アドレスに配置することにより、または、上述したように、選択されたオブジェクトにおける参照を調整することにより、特別な処理(arrangement)が行われなければ、ソースランタイムシステムにおける１又は２以上の参照されるオブジェクトに対する選択されたオブジェクトにお

ける何れの参照もターゲットランタイムシステムにおいて有効ではない。）

【０１８１】

そして、識別された一組のオブジェクトにおけるオブジェクトがトラバースされる。もし、一組のオブジェクトにさらにオブジェクトが存在すれば（判定１００４のブランチ“ｙｅｓ”）、次のオブジェクトが読み出され（１００６）、そして、ランタイムメタデータに対するオブジェクトにおける任意の参照（例えば、オブジェクトのクラスのランタイム表現に対する参照）がプロキシで置き換えられる（１００８）。全てのオブジェクトがトラバースされた後（判定１００４のブランチ“ｎｏ”）、一組のオブジェクトが、ターゲットランタイムシステムに送られる（１０１０）。もちろん、処理１０００におけるオペレーションは、別のシーケンスで実行されることが可能である－例えば、プロキシがオブジェクトに配置され、そしてこのオブジェクトは、一組のオブジェクト全体が識別される前に、ターゲットランタイムシステムに送られることができる。このようなオペレーションは、例えば、共有クロージャを識別する再帰的処理の一部としてオンザフライで実行されることができる。

【０１８２】

処理１０５０は、ターゲットランタイムシステムにおいて（例えば、ソースＶＭとは異なるマシン上で実行するターゲットＶＭにおいて）使用されることができる対応処理を示す。一組のデータオブジェクトが受信され（１０５２）、そして、ターゲットランタイムシステムにおいてコピーされる（１０５４）。そして、受信されたオブジェクトの組の中のオブジェクトがトラバースされる。もし、受信されたオブジェクトの組に更なるオブジェクトが存在すれば（判定１０５６のブランチ“ｙｅｓ”）、次のオブジェクトが受信される（１０５８）。そして、オブジェクトに対応する関連ランタイム情報(relevant runtime information)が識別される（１０６０）。このような情報は、例えば、オブジェクトのランタイムクラスを含むことができる。そして、このオブジェクトのプロキシは、上記ランタイム情報に基づいてターゲットランタイムシステムにおけるランタイムメタデータに対する参照で置き換えられることができる。例えば、一旦、オブジェクトのランタイムクラスが識別されると、このオブジェクトにおけるランタイム表現のためのプロキシは、ターゲットランタイムシステムにおけるクラスのランタイム表現に対する参照で置き換えられることができる。

【０１８３】

或る環境では、オブジェクトにおける１又は２以上のプロキシがターゲットランタイムシステムにおけるランタイムメタデータに対する参照で置き換えられる前に、まず、このようなメタデータがターゲットランタイムシステムにロードされたかどうかを判定するためにチェックが行われる。もし、メタデータがターゲットランタイムシステムにロードされていなければ（判定１０６２のブランチ“ｎｏ”）、メタデータは必要に応じてロードされることができる（１０６４）。一例として、クラスローダーは、プロキシがそのランタイム表現に対する参照で置き換えられる前にＶＭにクラスのランタイム表現をロードするために起動されることができる。

【０１８４】

処理１０００と同様に、処理１０５０におけるオペレーションは、別のシーケンスで実行されることができる－例えば、プロキシは、オブジェクトが受信されたときに（すなわち、一組のオブジェクトの全体が受信される前に）ランタイムメタデータに対する参照で置き換えられることができる。更なるバージョンもまた可能である。例えば、プロキシは、オブジェクトが受信されたとき、または、“オンデマンド(on demand)”ベースで（例えば、オブジェクトが最初にターゲットランタイムシステムにおいてアクセスされたとき）、ランタイムメタデータに対する参照で置き換えられることができる。

【０１８５】

共有クロージャの生成および使用と、ランタイムメタデータに対する参照の取り扱いを含む上述の技術は、ひとつのランタイムシステムにおける複合データ構造を分解し、他のランタイムシステムの本来のフォーマットでデータ構造を再構成する一般的概念の特別な

実施例である。本技術は、例えば、サーバー環境においてオブジェクトを共有するために使用されることができ、その環境では、複合データ構造はヒープ（または、ヒープ内の一組のオブジェクトのような、その一部）であり、ランタイムシステムはＶＭである。上述の技術のこのような使用例は、以下に更に詳細に説明される。

【０１８６】

図１１は、サーバーにおける共有オブジェクトを使用するための処理１１５０の例のフローチャートである。処理１１５０は、共有オブジェクトが、処理取り付け可能なＶＭとともにどのように使用されることができるかを示している。処理１１５０において、ＶＭは、ユーザーセッションのために生成され（１１５２）、そしてオブジェクトの共有クロージャが生成され、そして共有メモリに格納される（１１５４）。サーバーがユーザーセッションに対応するユーザーリクエストを受信すると（１１５６）、それは、サーバーに割り当てられた処理のプールから利用可能な処理を選択し（１１５８）、そして、ユーザーセッションのために生成されたＶＭを、選択された処理にバインドする（１１６０）。

【０１８７】

もし、ＶＭが共有クロージャにおけるオブジェクトをアクセスする必要がなければ（判定１１６２のブランチ“ｎｏ”）、ＶＭは、単にそのリクエストを処理することができ（１１６４）、その後、ＶＭは、その処理からバインド解除されることができる（１１６６）。そして、処理は、利用可能な処理のプールに戻されることができ、そしてそのサーバーは、新たなリクエストを受信したときに、再びＶＭを処理にバインドすることができる(１１５６)。

【０１８８】

一方、もし、ＶＭが共有クロージャにおけるオブジェクトをアクセスする必要があれば（判定１１６２のブランチ“ｙｅｓ”）、共有クロージャはＶＭと関連づけられる。共有クロージャは、種々の方法でＶＭと関連づけられることができる－例えば、共有クロージャは、共有クロージャを、選択された処理のアドレス空間にマッピングまたはコピーすることにより、選択された処理にバインドされることができる。

【０１８９】

共有クロージャがＶＭと関連づけられる的確な方法は、ＶＭによって必要とされるアクセスのタイプに依存する。上述したように、一つのサーバー実施例では、他のＶＭにおいて有効でない値にメンバー変数を設定することにより、マッピングＶＭが共有クロージャにおける共有オブジェクトインスタンスをブレークすることを防止するために、マッピングされた共有クロージャは、常にリードのみのアクセスに制限される。このような実施例では、もし、処理１１５０においてユーザーセッションに関連するＶＭが、オブジェクトに対するリードアクセスだけが必要ならば（すなわち、判定１１６８のブランチ“ｎｏ”によって示されるように、もし、ＶＭが、共有クロージャにおけるオブジェクトの何れをも修正する必要がないのであれば）、共有オブジェクトは、選択された処理のアドレス空間にマッピングされることができる（１１７０）。そして、ＶＭは、リクエストを処理し（１１６４）、必要に応じて共有オブジェクトから情報を読み出す。

【０１９０】

もし、ＶＭが共有オブジェクトに対するリードライトアクセスを必要とすれば（すなわち、判定１１６８のブランチ“ｙｅｓ”で示されるように、もし、ＶＭが１又は２以上のオブジェクトを修正する必要があれば）、共有クロージャは、選択された処理のアドレス空間にコピーされる（１１７２）。そして、ＶＭは、共有オブジェクトに対する完全なリードライトアクセスを有し、そしてそれは必要に応じてオブジェクトを修正し、リクエストを処理することができる。ＶＭが共有オブジェクトの修正を終了すると、それは、共有クロージャを共有メモリにコピーすることができる（１１７６）。上述のように、一つのサーバー実施例では、共有クロージャは、共有クロージャを再生成することにより（例えば、“生成(create)”機能を起動し、前に使用された同じネームを共有クロージャに割り当て、そしてそのネームを有する共有クロージャの古いバージョンが依然として共有メモリに存在すれば新たなバージョンの共有クロージャを生成することにより）、共有メモリ

にコピーされることができる。これまでのように、リクエストが処理された後は、ＶＭは処理からバインド解除されることができ（１１６６）、処理は、利用可能な処理のプールに戻されることができ、そして、サーバーは、新たなリクエストを受信するために再び待機することができる（１１５６）。

【０１９１】

共有クロージャにおけるオブジェクトは、以下の例に示すように、ユーザーコンテキスト情報を含んで、任意のタイプのオブジェクトを含むことができる。さらに、処理１１５０は、共有クロージャをアクセスする一つのＶＭを示しているだけではあるが、共有クロージャは、また、他のＶＭによってアクセスされることもできる（例えば、他のＶＭは、共有クロージャをマッピングまたはコピーすることができ、これにより共有クロージャにおけるオブジェクトをアクセスする）。

【０１９２】

共有仮想マシン(Shared Virtual Machines)

【０１９３】

図２および３に示されるサーバー実施例は、ユーザーセッションが大規模なユーザーコンテキストに対応する環境に良好に適合する。しかしながら、このような実施例は、ユーザーコンテキストが相当に小さい環境（例えば、ユーザーコンテキストがそれぞれ１０キロバイトよりも小さい環境）では最適でないかもしれない。ＶＭ間でオブジェクトを共有する能力があるとしても、このような環境において各ユーザーセッションに個別のＶＭを割り当てることは、ユーザーセッションごとに大きなオーバーヘッドを招くので、実際的ではないかもしれない。

【０１９４】

図１２は、ユーザーコンテキストが相当に小さいと予想される環境において、より実用的なサーバー２００の別の実施例を示す。各ユーザーセッションのために一つのＶＭを生成する代わりに、図１２に示されるサーバー２００の実施例は、共有されたＶＭ（例えば、ＶＭ３０１，３０３，３０５，３０７，３０９）のプール３００を使用する。サーバー２００は、各ユーザーセッションのためのユーザーコンテキスト（例えば、ユーザーコンテキスト５０１，５０３，５０５，５０７，５０９，５１１，５１３，５１５，５１９）を生成する。ユーザーコンテキストは、ＶＭのプール３００におけるＶＭがそうであるように、共有メモリ領域２５５に格納される。前述した実施例と同様に、サーバー２００は、ワーク処理（例えば、ワーク処理４０１，４０３，４０４，４０９）のプール４００、および配信処理４１０を使用する。

【０１９５】

オペレーションにおいて、図１２に示されるサーバー２００の実施例は次のように動作する。サーバー２００がユーザーリクエストを受信すると、配信処理４１０は、ワーク処理のプール４００から利用可能なワーク処理（例えば、ワーク処理４０４）を選択し、そしてユーザーリクエストを、選択されたワーク処理４０４に配信する。そして、ワーク処理４０４は、ＶＭのプール３００から利用可能なＶＭ（例えば、ＶＭ３０５）を選択する。選択されたＶＭ３０５は、例えば、共有メモリ２５５から、選択されたワーク処理４０４のアドレス空間にＶＭ３０５をマッピングすることにより、選択されたワーク処理４０４にバインドされる。

【０１９６】

次に、ワーク処理４０４は、ユーザーリクエストに対応するユーザーセッションと、関連するユーザーコンテキスト（例えば、ユーザーコンテキスト５０１）とを識別する。そして、識別されたユーザーコンテキスト５０１は、例えば、共有メモリ領域２５５からワーク処理４０４のアドレス空間にユーザーコンテキスト５０１をコピーすることにより、ワーク処理４０４にバインドされる。いまや、ユーザーリクエストは、ユーザーコンテキスト５０１に関連してワーク処理４０４においてＶＭ３０５を実行することにより処理されることができる。

【０１９７】

関連するユーザーコンテキストを識別し、そしてそのコンテキストを選択されたＶＭに関連づけるための他の実施例が可能である。例えば、配信処理４１０は、関連するユーザーコンテキストを識別し、そしてそのコンテキストを識別する情報を、選択されたＶＭに受け渡すことができ、そして、ユーザーコンテキスト（例えば、そのコンテキストを作り出したオブジェクトの共有クロージャ）は、選択されたＶＭに代わって、選択されたワーク処理のアドレス空間にマッピングまたはコピーされることができる。概念的には、これは、識別されたユーザーコンテキストが選択されたＶＭにプラグインされ、そして選択されたＶＭが選択されたワーク処理にプラグインされるというシーケンスとして考えることができ、技術的な観点からは、この概念的シーケンスは、識別されたユーザーコンテキストと選択されたＶＭを表す共有メモリのセクションを、選択されたワーク処理のアドレス空間に単にマッピングまたはコピーすることにより実行されることができる。

【０１９８】

リクエストが処理されると、ＶＭ３０５とユーザーコンテキスト５０１は、（例えば、ワーク処理４０４からＶＭ３０５をマッピング解除し、そしてユーザーコンテキスト５０１を共有メモリ領域２５５にコピーし直すことにより）ワーク処理４０４からバインド解除される。そして、ＶＭ３０５およびワーク処理４０４は、再び利用可能であるようにマークされ、そしてそれらは、サーバー２００によって受信された追加のリクエストを処理するために使用されることができる。もちろん、ＶＭ３０５とワーク処理４０４は、新たなリクエストの処理において共に対にされなくてもよい－例えば、もし、新たなリクエストがワーク処理４０４に配信され、且つＶＭ３０５が他のリクエストの処理で忙しければ、ワーク処理４０４は、ＶＭのプール３００から他のＶＭを選択しなければならない（または、ＶＭのプール３００におけるＶＭが利用可能になるのを待つ）。

【０１９９】

この方法では、図１２に示されるサーバー実施例におけるＶＭは共有されるが（すなわち、サーバーは、ユーザーセッションごとに一つのＶＭを割り当てない）、各ＶＭが一回で一つのユーザーセッションのみに専念するので、互いに分離されている。さらに、各ＶＭは、分離された処理において動作するので、もし、ＶＭがクラッシュすれば、このようなクラッシュは、（オペレーティングシステムが互いから処理をどの程度良好に分離しているかに依存して）クラッシュしたＶＭにおいて処理されていたリクエストに対応するユーザーセッションにしか影響を与えない。従って、共有されながらも専用化されるＶＭのプールを使用することの上記アプローチは堅牢なサーバーをもたらすが、前述した実施例よりもオーバーヘッドが少ない。

【０２００】

ＶＭが各ユーザーセッションに割り当てられるサーバー実施例において、種々の最適化が同一の利益－オーバーヘッドの低減－を達成するために使用されることができる。例えば、対応するユーザーセッションの終わりでＶＭを終了させる代わりに、ＶＭは再使用されることができる。すなわち、ＶＭは別のユーザーセッションに関連づけられ、そして新たなユーザーセッションに対応するリクエストを処理するために使用されることができる。有用ではあるが、にもかかわらず、このアプローチは、サーバーが各ユーザーセッションのために少なくとも一つのＶＭを例示化する必要があるので、多数の同時的ユーザーセッションが存在する場合には依然として大きなオーバーヘッドを必要とする。

【０２０１】

一方、図１２に示されるサーバー実施例は、どんなに多くのＶＭがＶＭプール３００に割り当てられても、ＶＭの数を制限する。前述したように、この実施例は、多くの同時的ユーザーセッションと小さなユーザーコンテキストを有する環境に良好に適合する。このような環境は、通常、数式Ｐ＜＝Ｖ＜＜Ｕにより特徴づけられる。ここで、Ｐはワーク処理４００のプールに割り付けられたワーク処理の数であり、Ｖは、ＶＭのプールに割り付けられたＶＭの数であり、Ｕは、サーバーによって処理されているユーザーセッションの数である。この数式の左部分は、ＶＭのプールに割り付けられたＶＭの数が通常ワーク処理のプールにおけるワーク処理の数に等しいかこれよりも大きくなければならないことを

示している（そうでなければ、全てのＶＭが使用されているときでさえ、幾つかの待機状態(idle)の使用されない処理が存在するからである）。上記数式の右部分は、ＶＭの数が、ユーザーセッションの数よりも十分に小さいことを示している。繰り返すが、このようなことは可能である。なぜなら、同時ではないにしても各ＶＭは異なるユーザーによって共有可能であるからである。

【０２０２】

図１２に示されるサーバー２００の実施例は、ＶＭ間でのオブジェクトの共有により可能とされる。図３に示されるサーバー実施例では、各ユーザーコンテキスト（例えば、ユーザーコンテキスト３０１）は、ユーザーセッションに対応する寿命期間の間、一つのＶＭ（例えば、ＶＭ５０１）と関連づけられ、そしてそのユーザーコンテキストとその関連づけられたＶＭは、対応するユーザーセッションからユーザーリクエストが受信されたときに、両方ともワーク処理にマッピングされる。従って、図３における実施例では、各ＶＭは、一つのユーザーセッションと関連づけられ、そしてＶＭは、上記関連づけられたユーザーコンテキストをアクセスすることしかできない－ＶＭが他のセッションに対応するユーザーコンテキストをアクセスすることは可能ではない。例えば、ＶＭ３０１は、ユーザーコンテキスト５０１をアクセスすることしかできず、そしてＶＭ３０３は、ユーザーコンテキスト５０３をアクセスすることしかできない。従って、各ＶＭは、その関連づけられたユーザーコンテキストに対応するリクエストを処理することしかできない。

【０２０３】

しかしながら、上述の共有化技術は、ユーザーコンテキストを作成するオブジェクトを含んで、ＶＭがオブジェクトを共有することを可能にする。例えば、ユーザーコンテキストは、共有クロージャとして共有メモリヒープに格納されることができる。そして、共有クロージャ－以下、ユーザーコンテキスト－は、多数のＶＭによってアクセスされることができる。

【０２０４】

従って、図１２に示されるサーバー実施例は、次のように説明される：第１に、ＶＭ及びユーザーコンテキストは互いに分離され；第２に、ユーザーコンテキストは、ＶＭプールにおける全てのＶＭによってアクセスできる共有メモリヒープに格納され；最後に、ユーザーリクエストは、対応するユーザーコンテキストと利用可能なＶＭとを利用可能なワーク処理にマッピングすることにより処理される。

【０２０５】

上述の構想(scheme)が図１３に概念的に示されている。図３と同様に、図１３におけるＶＭとユーザーコンテキストは、共有メモリ領域２５５に格納される。しかしながら、図３と異なり、図１３におけるＶＭは、もはや特定のユーザーコンテキストと関連づけられない－任意の利用可能なＶＭが任意のユーザーコンテキストと連動することができる。例えば、図３において、ＶＭ３０１は、その関連するユーザーコンテキスト５０１と連動することしかできない。これに対し、図１３では、ＶＭは、共有メモリにおける任意のユーザーコンテキスト、例えば、ユーザーコンテキスト５１９と連動することができる。

【０２０６】

幾つかの実施例では、一旦、ＶＭがユーザーコンテキストと対にされると、そのＶＭは、対応するユーザーコンテキストが処理されるまで、そのユーザーコンテキストと対にされたままでなければならない。特に、たとえＶＭが、現在、現在のユーザーリクエストの処理において（例えば、Ｉ／Ｏイベントを待っているために）ブロックされているとしても、そのＶＭは、利用可能なものとしてマークされることはできず、且つ、他のユーザーリクエストを処理するために使用されることはできない。

【０２０７】

例えば、Ｊａｖａ（登録商標）実施例では、リクエストが処理されている間に、（ユーザーヒープとユーザースタックを含む）ユーザーコンテキストから（ＶＭヒープとＶＭスタックを含む）ＶＭを分離することは容易ではない。なぜなら、ＶＭスタックは、ユーザースタックと混ざっているからである。従って、ＶＭは、対応するリクエストが処理され

てユーザースタックが空になるまで、ユーザーコンテキストと対にされたままでいなければならない。

【０２０８】

図１４は、上述の構想に対応するユーザーコンテキストとＶＭを対にする方法(pairing)を示す。サーバーがユーザーコンテキスト５０１に対応するリクエストを受信する前に、ユーザーコンテキスト５０１は共有メモリに格納され、そしてＶＭ３０１は、任意のユーザーコンテキストに対応するリクエストを処理するために利用可能である。ユーザーコンテキスト５０１に対応するリクエストが受信されたとき（１４０２）、ユーザーコンテキスト５０１とＶＭ３０１は、両方とも、ワーク処理４０１にバインドされる。従って、ユーザーコンテキスト５０１とＶＭ３０１は、ＶＭがユーザーコンテキスト５０１と共にワーク処理４０１において動作するように互いに対にされる。

【０２０９】

リクエストの処理における或る時点で、ＶＭにおける全てのスレッドまたはコルーチンは、（例えば、もし、それら全てが、Ｉ／Ｏイベントの完了を待っていれば）ブロックしてもよい。それが起これば（１４０４）、ＶＭ３０１とユーザーコンテキスト５０１は、ワーク処理４０１からバインド解除され、これにより、他のリクエストに取りかかるためにワーク処理４０１を開放する。しかしながら、ＶＭ３０１は、利用可能なものとしてマークされない－それは、ユーザーコンテキスト５０１と対にされたままとされ、そして、少なくとも一つのスレッドが再び動作するまで（例えば、指示されたＩ／Ｏイベントが完了したとき）アイドル状態(idle)を保つ。それが起これば（１４０６）、ＶＭ３０１とユーザーコンテキスト５０１は、そのＶＭがそのユーザーコンテキストと共に実行を継続するよう、ワーク処理に再びバインドされることができる。ＶＭ３０１とユーザーコンテキスト５０１は、それらが前にバインドされた同じワーク処理４０１にバインドされなくてもよい－例えば、もし、他のＶＭが、現在、ワーク処理４０１にバインドされていれば、ＶＭ３０１とユーザーコンテキスト５０１は、利用可能な他のワーク処理にバインドされる。

【０２１０】

最後に、リクエストが処理されて応答(response)が送られると（１４０８）、ＶＭ３０１とユーザーコンテキスト５０１は、ワーク処理からバインド解除される。ユーザーコンテキスト５０１は、共有メモリに再び格納され、そしてＶＭ３０１とワーク処理は、両方とも、他のリクエストに取りかかるために利用可能なものとしてマークされる。

【０２１１】

図１５は、図１２に示されるサーバー実施例においてリクエストを処理するための処理１５５０の例を示すフローチャートである。処理１５５０において、オペレーティングシステムのプールとＶＭのプールは、最初に初期化される（１５５２）。ユーザーコンテキストは、各ユーザーセッションのために生成され、そしてサーバーの共有メモリ領域に格納される（１５５４）。各ユーザーコンテキストは、例えば、オブジェクトの共有クロージャとして格納されることができる。

【０２１２】

サーバーがユーザーセッションからリクエストを受信すると（１５５６）、それは、対応するユーザーコンテキストを識別し（１５５８）、そして、ワーク処理のプールから利用可能な処理を選択すると共に(１５６０)、ＶＭのプールから利用可能なＶＭを選択する（１５６２）。そして、選択されたＶＭは、（例えば、共有メモリ領域からのＶＭを、選択された処理のアドレス空間にマッピングすることにより）選択された処理にバインドされる（１５６４）。ＶＭがリクエストを処理するために使用される前に、ユーザーコンテキストはＶＭと関連づけられなければならない（１５６６）。これは、例えば、ユーザーコンテキストを選択された処理にバインドすることにより行われることができる。上述したように、一実施例において、共有クロージャにおけるオブジェクトへのアクセスは、もし、共有クロージャが処理にコピーされていれば制限されないが、しかし、もし、共有クロージャがその処理にマッピングされていれば、リードのみのアクセスに制限される。こ

のような実施例では、ＶＭはリクエストの処理中にユーザーコンテキストを修正する必要があり得るので、ユーザーコンテキストは、処理にマッピングされるというよりは、むしろコピーされる。しかしながら、他の実施例が可能である－例えば、ユーザーコンテキストが処理にマッピングされ、マッピング処理におけるＶＭがそれらのオブジェクトを使用し或いは修正する間に他の処理またはＶＭがユーザーコンテキストにおけるオブジェクトをアクセスすることを防ぐようにロッキング変数(locking variable)が設定されることができる。ユーザーコンテキストに処理排他的アクセスと選択されたＶＭとを与えるための追加的バリエーションが可能である。

【０２１３】

一旦、選択されたＶＭが選択された処理にバインドされ、そしてユーザーコンテキストがそのＶＭと関連づけられると、リクエストは、ユーザーコンテキストと共に処理におけるＶＭを実行することにより実行されることができる。上述したように、一実施例において、もし、ＶＭがブロックし、或る理由で処理を継続することができなくなれば（例えば、もし、ＶＭがＩ／Ｏの発生を待っていれば）、ＶＭはワーク処理からバインド解除されることができる。そして、ワーク処理は、利用可能なものとしてマークされ、他のリクエストに取りかかるために使用されることができる。また、もし、ユーザーリクエストの処理が完了されるまでにユーザーコンテキストからＶＭを分離することが可能でなければ、ＶＭは、利用可能なものとしてマークされ、そして他のリクエストを処理するために使用されることができる。後者のシナリオでは、ＶＭは、ユーザーコンテキストと対にされたままでなければならない。もはやＶＭがブロックされなければ、ＶＭとユーザーコンテキストは、利用可能なワーク処理に再びバインドされることができ、そして、ＶＭは、ユーザーリクエストの処理を続けることができる。

【０２１４】

ユーザーリクエストが処理されたとき、ＶＭをワーク処理にバインドし、ユーザーコンテキストをＶＭと関連づけるオペレーションが用意されることができる－例えば、ユーザーコンテキストは、ＶＭから分離されることができ（１５７０）、そしてＶＭは、ワーク処理からバンドが解除されることができる（１５７２）。ユーザーコンテキストがＶＭから分離される方法は、ユーザーコンテキストがＶＭと関連づけられる方法に依存する。上述したように、一実施例において、ユーザーコンテキストは、ＶＭにバインドされた処理にコピーされ、この場合、ユーザーコンテキストは、このユーザーコンテキストを共有メモリ領域にコピーし戻すことによりＶＭから分離されることができる。また、前述したように、幾つかの実施例において、ユーザーコンテキストを共有メモリ領域にコピーすることは、もし、前のバージョンのユーザーコンテキストが依然として共有メモリ領域に存在すれば、新たなバージョンのユーザーコンテキストの生成をもたらすことができる。ＶＭをワーク処理にバインドすることと同様に、ＶＭのバインドをワーク処理から解除することは、また、簡単で低コストのオペレーションである：共有メモリのＶＭのブロックは、単純に、ワーク処理のアドレス空間からマッピング解除されることができる。

【０２１５】

ユーザーコンテキストがＶＭから分離され、ＶＭのバインドがワーク処理から解除された後、ＶＭとワーク処理は、両方とも、利用可能なものとしてマークされ、そして、他のリクエストを処理するためにサーバーによって使用されることができる。

【０２１６】

処理１５５０におけるオペレーションは、別のシーケンス（例えば、利用可能な処理の選択は、利用可能なＶＭの選択と同時、または、その前後に発生することができる）において、サーバーの異なるコンポーネントによって実行されることができる。例えば、配信処理（例えば、図１２の配信処理４１０）は、利用可能なワーク処理を選択し、単にユーザーリクエストを選択された処理に伝送して、それを利用可能なＶＭを選択するための処理に任せ、そのＶＭを処理にバインドし、ユーザーリクエストに対応するユーザーコンテキストを識別し、そしてユーザーコンテキストを選択されたＶＭに関連づけることができる。あるいは、配信処理は、幾つかのそれらのオペレーション自体を実行することができ

る－例えば、配信処理は、利用可能なＶＭを選択し、関連するユーザーコンテキストを識別し、そして、選択されたＶＭの識別と、識別されたユーザーコンテキストをワーク処理に送ることができる。他の代替が可能であり、それは、配信処理、ワーク処理、またはサーバーにおける追加的コンポーネントがオペレーションを実行する。

【０２１７】

上述した技術は、ユーザーコンテキストとオブジェクトの共有を含み、他の環境においても使用されることができる。例えば、この技術は、ＶＭが処理に取り付けられ、そして処理から取り外されるところのサーバー実施例において使用されることができる。図１６は、このようなサーバーにおいてリクエストを処理するための処理１６５０の例を示すフローチャートである。処理１６５０において、ワーク処理のプールはサーバーに割り当てられ、そしてＶＭはワーク処理のそれぞれにおいて初期化される（１６５２）。各ＶＭはＶＭの寿命期間の間にそのワーク処理にバインドされ、このことは、ＶＭがワーク処理からとりはずされて他のワーク処理に取り付けられることができないことを意味する。

【０２１８】

処理１５５０におけるのと同様に、ユーザーコンテキストは、各ユーザーセッションのために生成され、そしてサーバーにおける共有メモリ領域に格納される（１６５４）。サーバーがユーザーセッションからリクエストを受信すると（１６５６）、それは、対応するユーザーコンテキストを識別し（１６５８）、そしてワーク処理のプールから利用可能な処理を選択する（１６６０）。そして、ユーザーコンテキストは、（例えば、ユーザーコンテキストを選択された処理にバインドすることにより）選択された処理におけるＶＭと関連づけられる（１６６６）。そして、リクエストは、ユーザーコンテキストと共に、選択された処理におけるＶＭを実行することにより処理されることができる（１６６８）。ユーザーリクエストが処理されると、ユーザーコンテキストは、（例えば、ユーザーコンテキストを、選択された処理から共有メモリ領域にコピーし戻すことにより）ＶＭから分離されることができる（１６７０）。そして、ワーク処理（およびその関連するＶＭ）は、利用可能なものとしてマークされ、そして他のリクエストを処理するためにサーバーによって使用されることができる。

【０２１９】

図１５および図１６の比較から理解されるように、処理１５５０と処理１６５０は、後者の処理がワーク処理からのＶＭのバインドとその解除を含まないことを除いて、同様である。処理１６５０は、より少ないオペレーションを含むにもかかわらず、その処理－すなわち、ＶＭがワーク処理に取り付けられたり取り外されることのない処理－を実施するサーバーは、処理取り付け可能なＶＭを使用するサーバーと同様に効率的ではないようである。なぜなら、ＶＭが前者のサーバーにおいてブロックしたときに、ＶＭが稼動しているワーク処理は、他のリクエストを処理するために使用できないからである。これに対し、処理取り付け可能なＶＭを備えるサーバーにおいては、ＶＭがブロックしたとき、そのＶＭは、そのワーク処理からデリートされることができ、そして他のＶＭがその場所に取り付けられて他のリクエストに取り組むために使用されることができる。

【０２２０】

この明細書で述べられる本技術の種々の代替および最適化が可能である。例えば、資源消費および性能は、全てのＶＭに対して共通のデータを共有メモリに格納し、そしてこのようなデータを全てのワーク処理にマッピングすることにより、さらに改善されることができる。もし、タイプ情報（例えば、ロードされたＪａｖａ（登録商標）クラスのランタイム表現）が、この方法で共有されれば、クラスローディングのためのオーバーヘッド、検証、および各ＶＭのためのレゾリューション(resolution)が低減されることができる。

【０２２１】

他の例として、代替メカニズムがデータと情報（ＶＭとユーザーコンテキストの両方を含む）を共有するために使用されることができ－例えば、情報は、共有メモリよりはむしろ、ファイルまたはメッセージングシステムの使用を通じて共有されることができる。しかしながら、このようなメカニズムは、共有されるべき情報を存続させ、その存続を解除

するためには、一層複雑で高価なオペレーションを必要とするかもしれないので、共有メモリほど望ましくはないかもしれない。

【０２２２】

さらに他の例として、本明細書で述べた処理は、望ましい結果を達成するためには、示された特定のオーダー、またはシーケンシャルオーダーを必要としない。例えば、図１５および１６に示された処理において、ユーザーコンテキストを生成し、識別し、ＶＭと関連づけるためのオペレーションは、処理全体の内の多くの異なる場所で実行されることができる。本明細書で述べられた処理の或る実施例では、マルチタスクおよび並列処理が好ましい。

【０２２３】

さらに、種々のコンポーネントとエンティティ(entities)が、本明細書で述べた処理におけるオペレーションを実行するために使用されることができる。例えば、処理１５５０におけるオペレーションは、サーバー内の別のコンポーネントによって実行されることができる。例えば、一実施例において、配信処理（例えば、図１２における配信処理４１０）は、利用可能なワーク処理を選択し、単に、選択された処理にユーザーリクエストを伝送して利用可能なＶＭを選択するための処理に任せ、そのＶＭを処理にバインドし、ユーザーリクエストに対応するユーザーコンテキストを識別し、そしてユーザーコンテキストを、選択されたＶｍに関連づける。あるいは、配信処理は、幾つかのそれらのオペレーション自体を実行することができる－例えば、配信処理は、利用可能なＶＭを選択し、関連するユーザーコンテキストを識別し、そしてワーク処理に識別されたユーザーコンテキストと選択されたＶＭの識別を送る。他の代替が可能であり、それは、配信処理、ワーク処理、またはサーバーにおける追加的または別のコンポーネントがオペレーションを実行する。例えば、サーバーは、配信処理を用いることなく実施されることができ、他のメカニズムが、ワーク処理間でリクエストを分配するために使用されることができる。

【０２２４】

本発明と本明細書において述べられた機能的オペレーションの全ては、本明細書で開示された構造的手段、その構造的等価物、またはそれらの組み合わせを含み、デジタル電子回路、またはコンピュータソフトウェア、ファームウェア、またはハードウェアで実施されることができる。本発明は、１又は２以上ののコンピュータプログラムプロダクト、すなわち、例えば、プログラマブルプロセッサ、コンピュータ、または複数のコンピュータなどのデータ処理装置によって実行されるための、または上記データ処理装置のオペレーションを制御するための、例えば、マシン読み取り可能な記憶装置または伝搬信号において、明白に具現化される１又は２以上のコンピュータプログラムとして実施されることができる。コンピュータプログラム（プログラム、ソフトウェア、ソフトウェアアプリケーション、またはコードとして知られる）は、コンパイルまたはインタプリットされた言語を含んで、どのような形式のプログラム言語で記述されることもでき、そしてそれは、スタンドアローンプログラムまたはモジュール、コンポーネント、サブルーチン、またはコンピューティング環境における使用に適した他のユニットなどを含んで、どのような形式でも展開されることができる。コンピュータプログラムは、必ずしもファイルに対応しない。プログラムは、他のプログラムまたはデータを保持するファイルの一部に、問題になっているプログラム専用の単一のファイルに、または、複数のコーディネートされたファイル（例えば、１又は２以上のモジュール、サブプログラム、コードの一部を格納するファイル）に格納されることができる。コンピュータプログラムは、展開されて、通信ネットワークによって相互接続されて複数のサイトに分散された、または、一つのサイトでの複数のコンピュータ上または一つのコンピュータ上で実行されることができる。

【０２２５】

本発明の方法ステップを含む処理とロジックフローは、入力を処理し出力を生成することにより本発明の機能を実行するための１又は２以上のコンピュータプログラムを実行する１又は２以上のプログラマブルプロセッサによって実施されることができる。また、処理およびロジックフローは、例えば、ＦＰＧＡ(field programmable gate array)または

ＡＳＩＣ(application-specific integrated circuit)などの特定用途のロジック回路によって実行されることができ、そして、本発明の装置は、上記特定用途のロジック回路として実施されることができる。

【０２２６】

コンピュータプログラムの実行に適したプロセッサは、一例として、汎用と特定用途の両方のプロセッサ、および１又は２以上の任意の種類の任意のデジタルコンピュータを含む。通常、プロセッサは、読み出し専用メモリまたはランダムアクセスメモリまたはその両方から命令およびデータを入力する。コンピュータの本質的要素は、命令を実行するためのプロセッサと、命令とデータを格納するための１又は２以上のメモリデバイスである。通常、コンピュータは、また、例えば、磁気的、磁気光ディスク、または光ディスクなど、データを格納するための１又は２以上の大容量記憶装置を備え、または、上記大容量記憶装置からのデータの受信または上記大容量記憶装置へのデータの送出、あるいはその両方のために動作可能に結合されている。コンピュータプログラム命令とデータを具現化するのに適した情報キャリアは、全ての形式の不揮発性メモリを含み、一例として、例えば、ＥＰＲＯＭ、ＥＥＰＲＯＭ、およびフラッシュメモリデバイスなどの半導体メモリ；例えば、内臓ハードディスクまたはリムーバブルディスクなどの磁気ディスク；磁気光ディスク；およびＣＤ－ＲＯＭおよびＤＶＤ－ＲＯＭディスクがある。プロセッサおよびメモリは、特定用途のロジック回路によって補完されることができ、または特定用途のロジック回路に組み込まれることができる。

【０２２７】

本発明は、コンピューティングシステムにおいて実施されることができ、このコンピューティングシステムは、バックエンドコンポーネント（例えば、データサーバー）、ミドルウェアコンポーネント（例えば、アプリケーションサーバー）、またはフロントエンドコンポーネント（例えば、ユーザーが本発明の実施例と情報をやり取りすることができるグラフィカルユーザーインターフェイスまたはＷｅｂブラウザを備えたクライアントコンピュータ）、またはこのようなバックエンド、ミドルウェア、およびフロントエンドコンポーネントの任意の組み合わせを含む。本システムのコンポーネントは、例えば、通信ネットワークなど、デジタルデータ通信の任意の媒体または形式によって相互接続されることができる。通信ネットワークの例は、ＬＡＮ(Local Area Network)およびＷＡＮ(Wide Area Network)、例えばインターネットを含む。

【０２２８】

コンピューティングシステムはクライアントとサーバーを含むことができる。クライアントとサーバーは、通常、互いに離れており、代表的には、通信ネットワークを通じて情報のやり取りを行う。クライアントとサーバーとの関係は、各自のコンピュータ上で稼動して互いにクライアント－サーバー関係を有するコンピュータプログラムのおかげで発生する。

【０２２９】

本発明は、特定の実施例の観点で説明されたが、他の実施例が本願特許請求の範囲内で可能である。例えば、上述したように、本発明のオペレーションは、別のオーダーで実施されることができ、それでも望ましい結果を達成することができる。他の実施例は、本願特許請求の範囲内である。

【図面の簡単な説明】

【０２３０】

【図１】クライアント／サーバーシステムのブロック図である。

【図２】図１のクライアント／サーバーシステムにおけるサーバーのブロック図である。

【図３】図２におけるサーバーの他のブロック図である。

【図４】図２のサーバーにおけるリクエストの処理を説明するための処理のフローチャートである。

【図５】サーバーにおける共有オブジェクトの使用を説明するためのブロック図である。

【図６】オブジェクトインスタンスが共有可能であるかどうかを判定するための処理を説

明するためのフローチャートである。
【図７】クラスが共有可能であるかどうかを判定するための処理を説明するためのフローチャートである。
【図８】共有されたオブジェクトを生成し使用するための処理を説明するためのフローチャートである。
【図９】共有されたオブジェクトを生成し使用するための処理を説明するためのフローチャートである。
【図１０】複数のマシンでオブジェクトを共有するための処理を説明するためのフローチャートである。
【図１１】サーバーにおける共有オブジェクトを使用するための処理を説明するためのフローチャートである。
【図１２】図１のクライアント／サーバーシステムにおける他のサーバーのブロック図である。
【図１３】図１２におけるサーバーの他のブロック図である。
【図１４】図１２のサーバーにおけるリクエストの処理を説明するためのブロック図である。
【図１５】図１２のサーバーにおけるリクエストの処理を説明するための処理のフローチャートである。
【図１６】図１のクライアント／サーバーシステムの中の他のサーバーにおけるリクエストの処理を説明するための処理のフローチャートである。
【符号の説明】
【０２３１】
１００；クライアント／サーバーシステム
１０２，１０４，１０６；クライアントシステム
１５０；ネットワーク
２００；サーバー
２０２；プロセッサ
２５０；メモリ
３３０；仮想マシン(ランタイムシステム)

【図１】

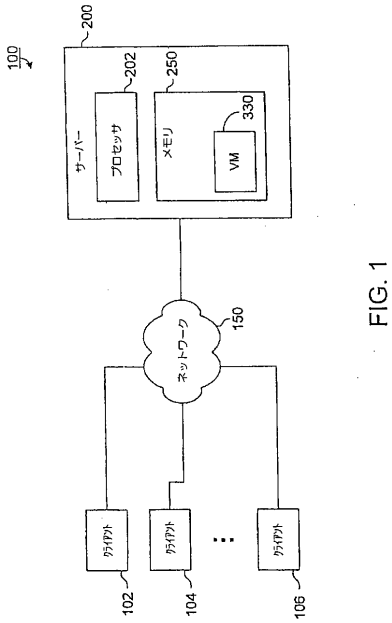


FIG. 1

【図２】

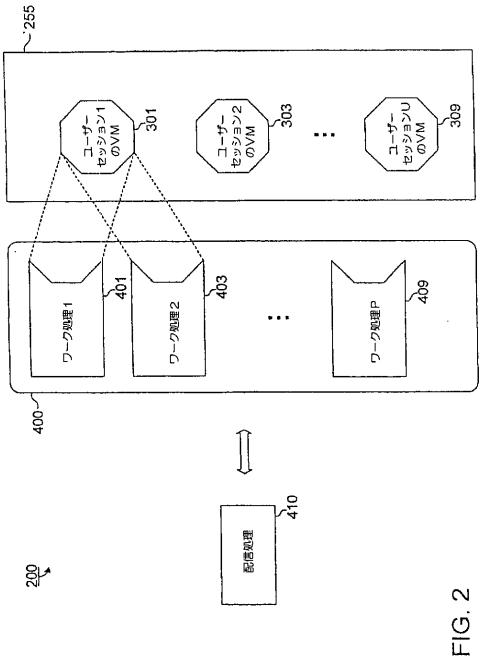


FIG. 2

【図３】

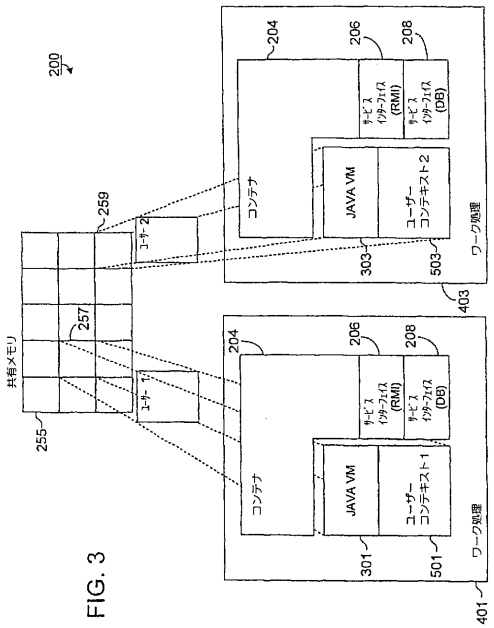


FIG. 3

【図４】

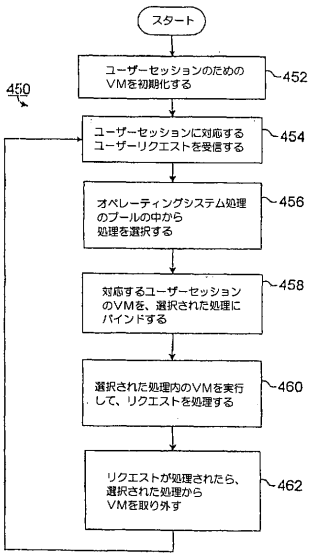


FIG. 4

【図５】

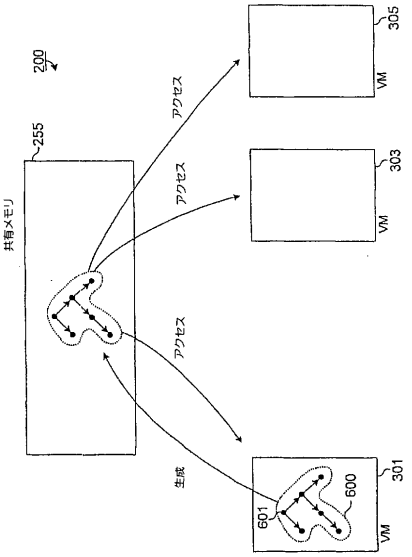


FIG. 5

【図６】

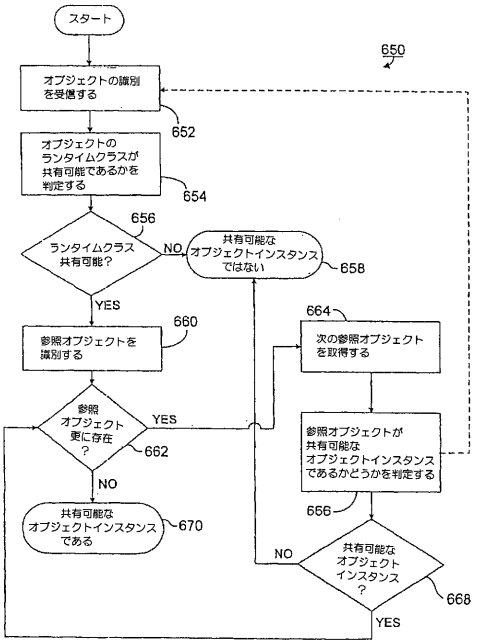


FIG. 6

【図７】

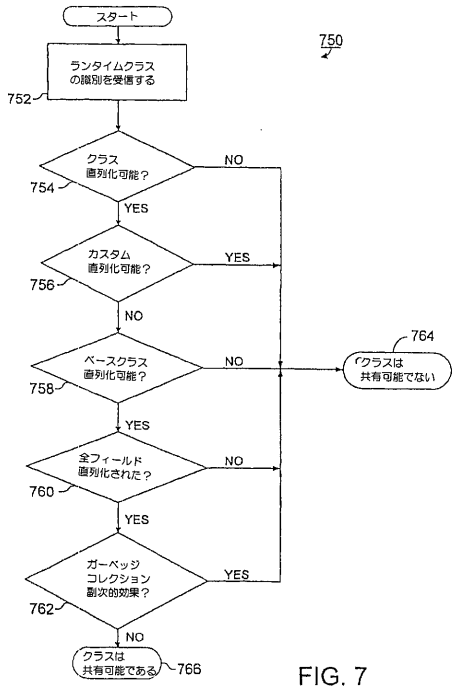


FIG. 7

【図８】

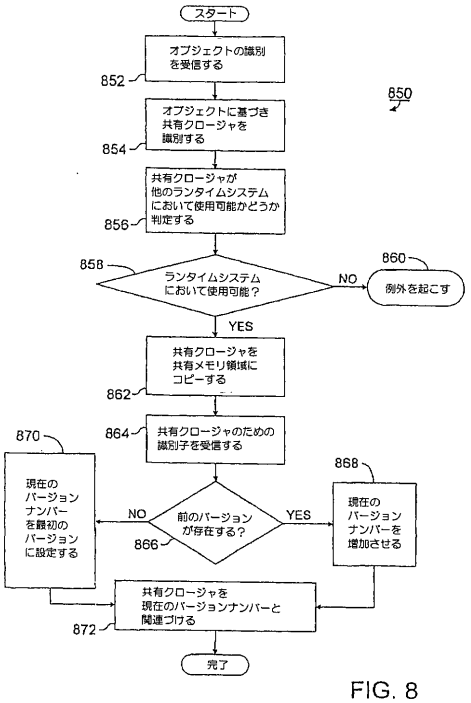


FIG. 8

【図９】

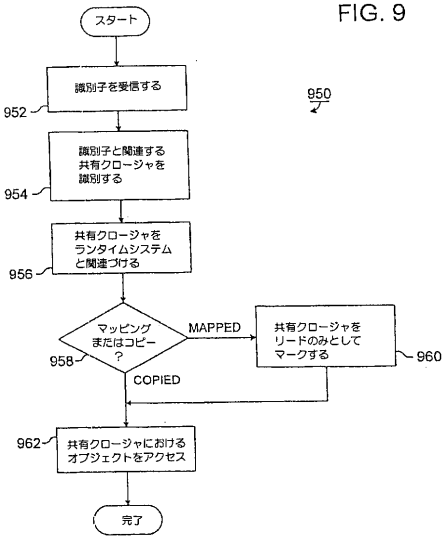


FIG. 9

【図１０】

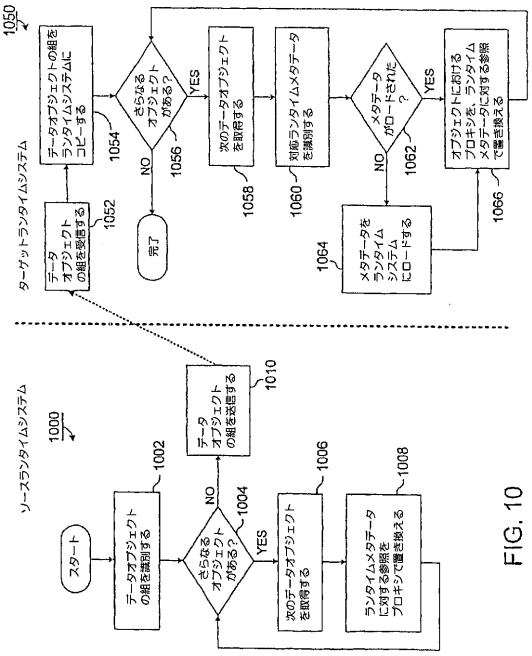


FIG. 10

【図１１】

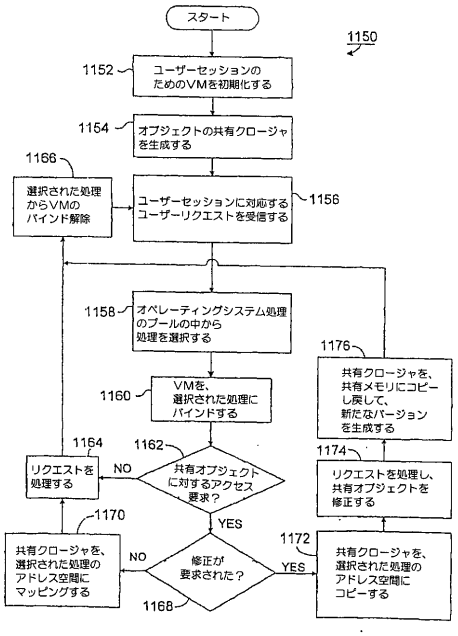


FIG. 11

【図１２】

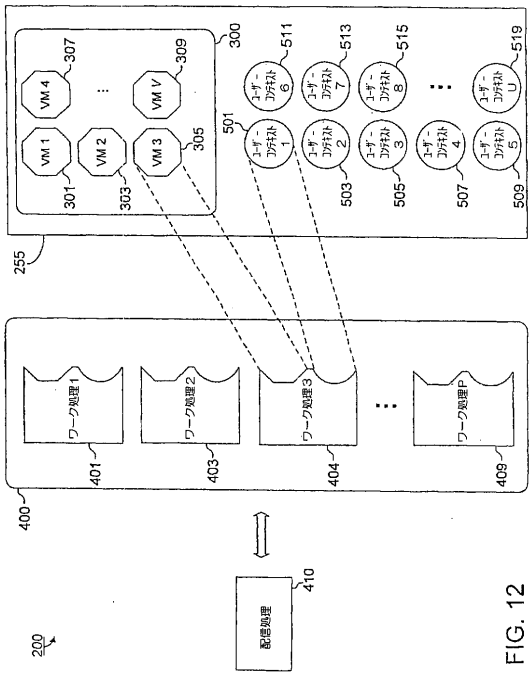


FIG. 12

【図１３】

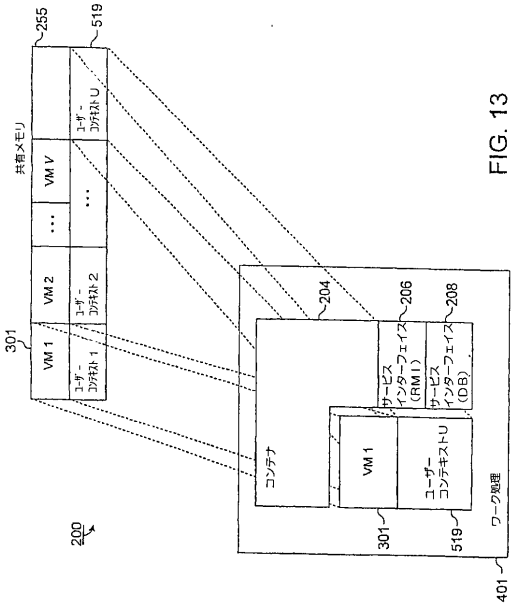


FIG. 13

【図１４】

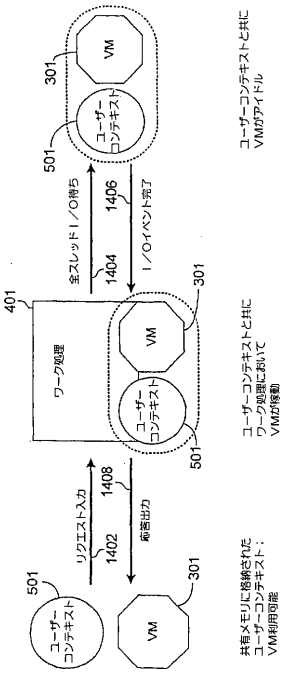


FIG. 14

【図１５】

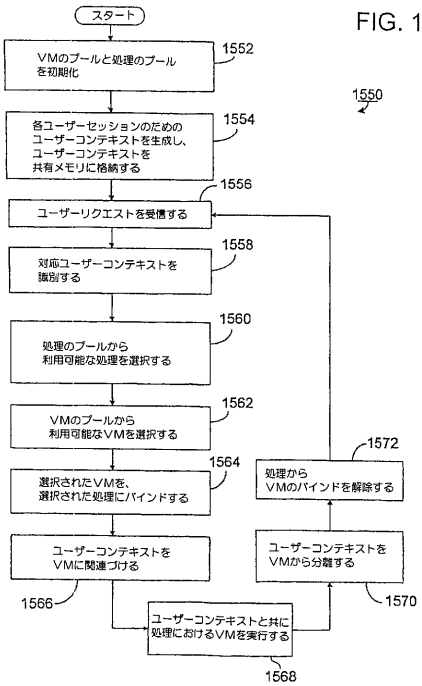


FIG. 15

【図１６】

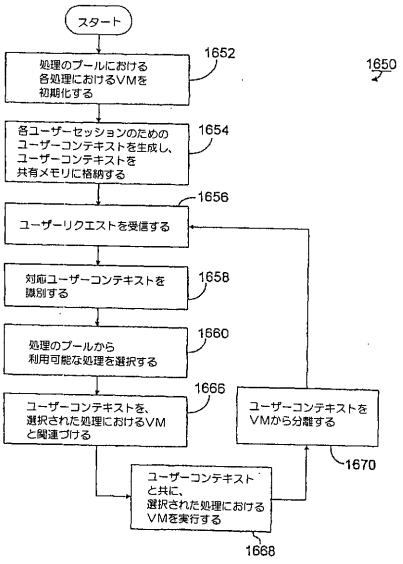


FIG. 16

【国際調査報告】

INTERNATIONAL SEARCH REPORT

International Application No
PCT/EP2005/005502

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/46 G06F9/445

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2003/135583 A1 (YARED PETER A ET AL) 17 July 2003 (2003-07-17) paragraph '0004! - paragraph '0008! paragraph '0019! - paragraph '0034!	1-20
Y	US 5 999 988 A (PELEGRI-LLOPART ET AL) 7 December 1999 (1999-12-07) column 4, line 16 - column 5, line 10 column 5, line 63 - column 10, line 40	1-20
A	US 2003/028865 A1 (SOKOLOV STEPAN ET AL) 6 February 2003 (2003-02-06) paragraph '0013! - paragraph '0036!	1-20
A	WO 03/040919 A (SAP AKTIENGESELLSCHAFT; KUCK, NORBERT; KUCK, HARALD; LOTT, EDGAR; ROHL) 15 May 2003 (2003-05-15) the whole document	1-20

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
26 October 2005	07/11/2005

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Kusnierczak, P

Form PCT/ISA/210 (second sheet) (January 2004)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/EP2005/005502

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2003135583 A1	17-07-2003	NONE	
US 5999988 A	07-12-1999	DE 69814611 D1	18-06-2003
		DE 69814611 T2	27-11-2003
		EP 0972241 A1	19-01-2000
		JP 2002516006 T	28-05-2002
		WO 9844414 A1	08-10-1998
US 2003028865 A1	06-02-2003	EP 1481320 A2	01-12-2004
		JP 2005521117 T	14-07-2005
		WO 03012637 A2	13-02-2003
WO 03040919 A	15-05-2003	EP 1442372 A2	04-08-2004

フロントページの続き

(81)指定国 AP(BW,GH,GM,KE,LS,MW,MZ,NA,SD,SL,SZ,TZ,UG,ZM,ZW),EA(AM,AZ,BY,KG,KZ,MD,RU,TJ,TM),EP(AT,BE,BG,CH,CY,CZ,DE,DK,EE,ES,FI,FR,GB,GR,HU,IE,IS,IT,LT,LU,MC,NL,PL,PT,RO,SE,SI,SK,TR),OA(BF,BJ,CF,CG,CI,CM,GA,GN,GQ,GW,ML,MR,NE,SN,TD,TG),AE,AG,AL,AM,AT,AU,AZ,BA,BB,BG,BR,BW,BY,BZ,CA,CH,CN,CO,CR,CU,CZ,DE,DK,DM,DZ,EC,EE,EG,ES,FI,GB,GD,GE,GH,GM,HR,HU,ID,IL,IN,IS,JP,KE,KG,KM,KP,KR,KZ,LC,LK,LR,LS,LT,LU,LV,MA,MD,MG,MK,MN,MW,MX,MZ,NA,NG,NI,NO,NZ,OM,PG,PH,PL,PT,RO,RU,SC,SD,SE,SG,SK,SL,SM,SY,TJ,TM,TN,TR,TT,TZ,UA,UG,US,UZ,VC,VN,YU,ZA,ZM,ZW

（特許庁注：以下のものは登録商標）
１．Ｌｉｎｕｘ

(74)代理人 100110364
弁理士 実広 信哉
(72)発明者 オリヴァー・シュミット
ドイツ・７６７４４・ヴォルト・アム・ライン・フィッシャーシュトラーセ・６
(72)発明者 ノーバート・クック
ドイツ・７６１３７・カールスルーヘ・カール－ホフマン－シュトラーセ・３
(72)発明者 エドガー・ロット
ドイツ・６９２２６・ヌスロッハ・ベンツ・シュトラーセ・２
(72)発明者 マーティン・シュトラースバーガー
ドイツ・７６６８４・オーストリンゲン・コンラディン－クロイツァー－シュトラーセ・１６
(72)発明者 アルノ・ヒルゲンバーク
ドイツ・６８１６１・マンハイム・Ｒ４・９
(72)発明者 ラルフ・シュメルター
ドイツ・６９１６８・ヴィースロッハ・ゾフィーンシュトラーセ・７
(72)発明者 ヤン・ドサート
ドイツ・６９２２６・ヌスロッハ・ブルクシュトラーセ・１９