

透過的かつ適応的にトランスポート層プロトコルを変更する フレームワークの開発

川 島 龍 太[†] 計 宇 生^{††} 丸 山 勝 巳^{††}

インターネットを始めとする情報ネットワークでは、TCP/IP プロトコル群が長らく中核的な役割を果たしている。そのため、ほとんどのネットワーク型アプリケーションは TCP/IP プロトコル群の利用を前提として開発されており、さらに QoS やセキュリティなどのサービス機能もアプリケーションレベルで実装されている。結果として、ネットワーク技術の高度化に伴い、ネットワーク型アプリケーションの開発もより複雑化・固定化してしまっている。そこで、筆者らは、マルチプラットフォーム上でサービス機能およびトランスポート層プロトコルに依存しない透過的なアプリケーション実行環境を実現するためのフレームワークを提案する。提案フレームワークは、システムコールフック技術を応用し、既存アプリケーションへの透過的なサービス機能追加、任意トランスポート層プロトコルの追加および通信相手のプロトコルに応じて適応的にプロトコルを選択する機能（プロトコルフリー環境）を実現する。

Design and Implementation of a Transparent Framework Providing Transport-Layer Protocol-Free Environment

RYOTA KAWASHIMA,[†] YUSHENG JI^{††} and KATSUMI MARUYAMA^{††}

TCP/IP protocol stack has long been core functions of information networks including the Internet. Therefore, many applications are bound to work with current TCP/IP protocols and service functions like QoS or security also implemented as application-level libraries. As a result, network applications become increasingly complex and lose their flexibility.

In this paper, we propose a framework for transparent protocol-free environment of any transport-layer protocols which enables users to add service functions or arbitrary transport-layer protocols into existing applications transparently. Moreover, the framework utilizes the system-call hooking methods and enables server-type applications to support protocols used by client-type applications adaptively.

1. はじめに

インターネットを始めとする情報ネットワークでは、TCP/IP プロトコル群が長らく中核的な役割を果たしている。そのため、多くのネットワーク型アプリケーションは TCP/IP プロトコル群の利用を前提として開発されており、さらに QoS やセキュリティなどのサービス機能もアプリケーションレベルで実装されている。つまり、アプリケーションが利用するプロトコルやサービス機能は、基本的にアプリケーションプログラム内で指定しなければならない。

この TCP/IP の固定化に起因して、ネットワーク

内には、新式のプロトコルやサービス機能に対応していない旧式システムも未だ数多く存在している。したがって、新規システム開発の際には既存システムとの互換性にも気を配らなければならない。

そこで、筆者らは、マルチプラットフォーム上でサービス機能およびトランスポート層プロトコル (L4 プロトコル) に依存しない透過的なアプリケーション実行環境を実現するためのフレームワークを提案する。筆者らは、これまでに Windows や Linux などで作動作する既存のネットワーク型アプリケーションに対して、サービス機能を透過的に挿入するためのフレームワークである FreeNA¹⁾を開発してきた。本論文では、この FreeNA を拡張し、任意の L4 プロトコルの追加およびプロトコルフリー化を実現する。本システムを用いることにより、新規/既存のアプリケーションに対して透過的にサービス機能を追加できるだけでなく、TCP/UDP 以外の任意 L4 プロトコルを透過的に利用

[†] 総合研究大学院大学 複合科学研究科
School of Multidisciplinary Sciences, The Graduate
University for Advanced Studies (SOKENDAI)

^{††} 国立情報学研究所
National Institute of Informatics (NII)

できるようになる。さらに、通信相手が利用するプロトコルに応じて、プロトコルを適応的に選択するプロトコルフリー環境を実現する。

本論文では、Windows および Linux 上で提案フレームワークを実装し、性能評価を行った。評価には、本システムによって追加された（カーネル内のものではない）UDP プロトコルを用いている。その結果、新たに追加された UDP プロトコルは、十分なパケットサイズでは既存 UDP プロトコルに近い性能（最大 800Mbps 超）を得ることができた。

2. 提案フレームワークの概要

本章では、提案フレームワークについて、その概要を一通り述べる。

● サービス機能の追加

新規／既存のネットワーク型アプリケーションに対して、暗号化、認証、誤り制御などのサービス機能を透過的に追加する。サービス機能は共有ライブラリとして実装されており、アプリケーションが利用するフロー単位（IP アドレス、ポート番号）で追加することが可能である。

● トランスポート層プロトコルの追加

本来ならば、アプリケーションは、あらかじめ OS がサポートしているプロトコル（TCP や UDP）しか利用できない。そのため、研究や特殊目的でサポート外のプロトコルを利用する際には、アプリケーション側でソケットの RAW IP 機能や UDP トンネリングなどを実装する必要がある。あるいは、Linux などのカーネル内に直接プロトコルを実装することも可能であるが、運用性に乏しく、セキュリティ上も問題がある。提案フレームワークでは、L4 プロトコルをユーザライブラリとして実装することによって、既存アプリケーション・OS をそのまま活用している。

● トランスポート層プロトコルフリー環境

L4 プロトコルは、エンド間で信頼性、フロー制御、多重化などの機能を実現するために利用されている。したがって、アプリケーションの性質、利用形態に応じて利用するプロトコルを使い分けことができると便利である。そのためには、アプリケーション側が利用形態や通信相手に応じてプロトコルを適応的に変更可能にする仕組みを持つ必要がある。本システムは、あたかもリバースプロキシのように動作することでアプリケーションの代わりにこの仕組みを実現している。

● 設定ファイルをベースとした完全透過的な機能拡張
提案フレームワークでは、アプリケーション本体からネットワークの詳細を分離させるために、設定ファイ

ルをベースとした完全透過的な機能拡張を行う。このため、アプリケーションは予め機能拡張可能な設計になっている必要はなく、修正や再コンパイルなしに必要な機能を追加することができる。

● マルチプラットフォーム対応

提案フレームワークは、Windows や Linux などの様々なプラットフォーム上で動作するように設計されている。API や ABI(Application Binary Interface) などのプラットフォーム固有の詳細は本システムによって隠蔽され、利用者には抽象化されたインタフェースが提供される。

3. アーキテクチャ

提案フレームワークは、前述した FreeNA の基本構造を踏襲しており、エンドアプリケーション間の通信パスに必要な機能を挿入することによって透過的な機能拡張を実現している。図 1 に、フレームワークのシステム概要図を示す。図のように、提案フレームワークは FreeNA クライアント／サーバ、設定ファイル、サービス機能ライブラリ、L4 プロトコルライブラリなどのコンポーネントによって構成されている。

3.1 FreeNA クライアント／サーバ

FreeNA サーバは、利用者の指示に基づいてアプリケーションに対して指定されたサービス機能やプロトコルを挿入する役割を果たす。具体的には、アプリケーションが呼び出す特定のシステムコールをフックし、代わりに指定された処理を実行するように設定する。FreeNA クライアントは、利用者が FreeNA サーバに命令を出すためのコマンド群などのインタフェースを提供している。

3.2 サービス機能ライブラリ

QoS やセキュリティなどのサービス機能は、それぞれ単一の共有ライブラリとして実現される。したがって、サービス機能自身は FreeNA サーバには直接組み込まれておらず、第三者による開発・配布が可能である。ライブラリが提供する関数群は、フックされたシステムコールに代わってアプリケーションから直接実行される。

3.3 トランスポート層プロトコルライブラリ

L4 プロトコルも、各サービス機能と同様に OS のカーネル内ではなくユーザ空間で実装されている。通常のネットワーク型アプリケーションとは異なり、カーネル内の TCP/UDP プロトコルスタックを介さずに直接 IP プロトコルスタックを利用することによって任意のプロトコルをアプリケーションに追加することが可能になる。

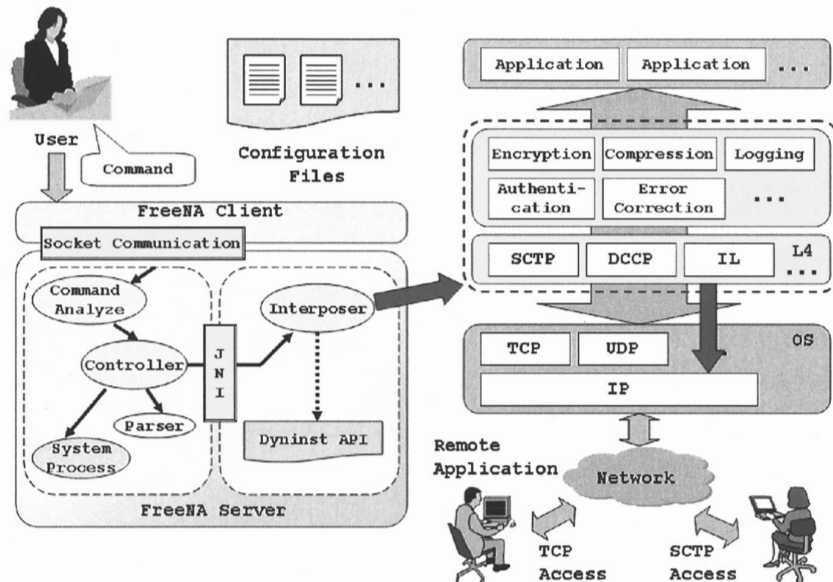


図 1 FreeNA をベースとした提案フレームワークのシステム概要図
Fig.1 System architecture of the framework based on FreeNA.

3.4 設定ファイル

アプリケーションに適用されるサービス機能やプロトコルは、全て XML 形式の設定ファイルによって制御することができる。設定ファイルはアプリケーションごとに用意するため、アプリケーションの特性に合わせた設定が可能である。図 2 に設定ファイルのフォーマットを示す。

```
<?xml version="1.0" encoding="Shift_JIS"?>
<configuration application="ftp_server">
  <services>
    <service name="crypto" lib="libcrypto.so">
      <parameter name="algorithm" value="AES"/>
      <parameter name="key_size" value="128"/>
      <parameter name="mode" value="CBC"/>
      <rule use="true" service="ftp-ctl" port="21"
        transport="TCP,SCTP" type="client"/>
    </service>
    <service name="compression" lib="libcomp.so">
      <rule use="true" service="ftp-data" port="20"
        transport="TCP,SCTP" type="client"/>
      <rule use="false" service="*" port="*"
        transport="*" type="*" />
    </service>
  </services>
  <using-rules>
    <rule use="true" service="ftp-data" port="20"
      transport="TCP,SCTP" type="client"/>
    <rule use="false" service="*" port="*"
      transport="*" type="*" />
  </using-rules>
  <protocols>
    <default name="TCP" lib="kernel"/>
    <option name="SCTP" lib="libsctp.so">
      <parameter name="path-mtu" value="true"/>
    </option>
  </protocols>
</configuration>
```

図 2 設定ファイルのフォーマット例
Fig.2 An example of configuration file

services タグには、利用するサービス機能、ライ

ブラリ名、パラメタなどを指定する。rule タグには、指定したサービスを適用する通信フロー条件を記述する。このサービス挿入ルールには、グローバルルールとローカルルールの二種類があり、ローカルルールは単一のサービスに対してのみ適用され、グローバルルールよりも優先される。一方、protocols タグにも同様に利用する L4 プロトコル、ライブラリ名、パラメタを記述する。default タグで指定したプロトコルは、自身が接続を開始する際に使用される。option タグで指定した場合は、接続相手が使用した際に自身に対応可能なプロトコルを指定する。

4. 実 装

本章では、提案フレームワークのコア概念である透過的なサービス機能の追加、L4 プロトコルの追加およびプロトコルフリー環境の実現方法について詳しく述べる。

図 3 は、サービス機能および L4 プロトコルの挿入構造を示している。FreeNA は、各サービスライブラリ、プロトコルライブラリを階層的に挿入する。各ライブラリにはそれぞれソケット関数と同等のインタフェースを持つ関数群があり、アプリケーションを起点として連鎖的に呼び出される。その他、制御ライブラリ、インタフェースライブラリと呼ばれるライブラリも挿入される。

制御ライブラリは、設定ファイルを基にしてライブラリ群の関数呼び出しグラフを構築し、実行時に Dyninst API²⁾ によってメモリ上のアプリケーションプロセスイメージにマッピングされる。この際、アプリケーションがソケット関数を呼び出す `call` 命令の呼び出し先は、ライブラリの対応する関数へと変更される。

インタフェースライブラリは、サービスライブラリとプラットフォーム固有のソケット関数との橋渡しを行う。内部に受信データ用のバッファを持ち、メッセージ境界の同期を取っている。また、追加した L4 プロトコルを使用する際には、ソケット関数の代わりに、追加した L4 ライブラリの関数を呼び出す。

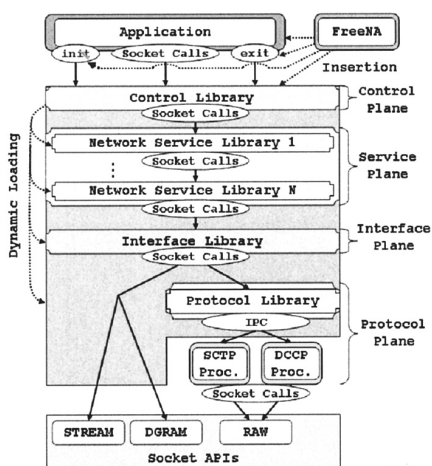


図3 サービス機能およびトランスポート層プロトコルの挿入構造
Fig.3 An insertion structure of service functions and transport-layer protocols

4.1 サービス機能の挿入

サービス機能の挿入については既に FreeNA によって実現されており、本システムでもその仕組みをそのまま利用している。実行時、制御ライブラリは設定ファイル内のグローバルルールおよびローカルルールに基づいて、ソケットごとにどのサービス機能を利用するかを判断を行う。サービス機能を利用すると判断した場合、制御ライブラリは下位層のサービスライブラリ関数を直接実行する。そうでなければ、インタフェースライブラリの関数群を実行する。これらの詳細については、文献¹⁾に詳しく述べられている。

4.2 トランスポート層プロトコルの挿入

ユーザ空間内で L4 プロトコルを実装するには、RAW ソケットと呼ばれる機能を利用する必要がある。

RAW ソケットは、通常の STREAM ソケットや DGRAM ソケットとは異なり、直接 IP パケットを作成することができる。通常、RAW ソケットを作成するためには、管理者権限あるいは Linux における特殊なカーバリティ（CAP_NET_RAW）が必要になる。

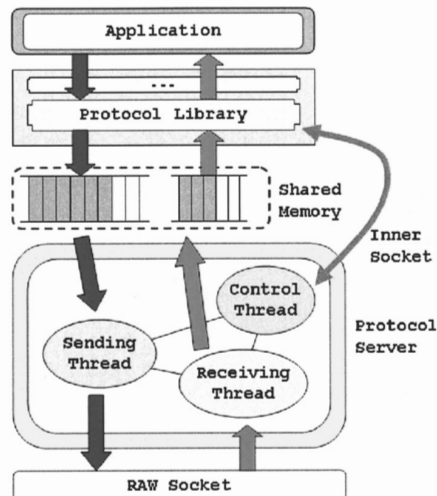


図4 トランスポート層プロトコルの実装構造
Fig.4 An implementation structure of transport-layer protocols

提案フレームワークでもこの RAW ソケットを利用して L4 プロトコルを実現している。図4にその実装構造を示す。L4 プロトコルはプロトコルライブラリとプロトコルサーバから構成されている。プロトコルサーバはパケットの送受信やコネクション管理などの中核処理を行う独立したプロセスであり、プロトコルライブラリは、インタフェースライブラリとプロトコルサーバの橋渡しを行う。ライブラリ-サーバ間では、データパケットをやり取りする際には共有メモリを利用し、制御情報をやり取りする際には内部用のソケット通信を行っている。

このような独立したプロトコルサーバが必要なのは、RAW ソケットがポート番号という概念を持たないからである。プロトコルサーバ内でポート番号を管理し、受信したパケットを適切なプロセスに転送させることで、メモリ使用量を抑えつつ複数のアプリケーションに対応することが可能になる。また、プロトコルサーバを必要な権限で実行すれば、アプリケーションは通常の権限のまま動作させることができる。

4.3 プロトコルフリーアクセス

ソケット関数には、コネクション型のトランスポート層プロトコル用の関数群があり、概念上は TCP 以

外のプロトコルにも適用できる。そこで、この関数群を新たに追加したプロトコルにも適用させる。実際には、**connect** や **accept** などのソケット関数を用いてプロトコルサーバとやり取りし、サーバ内部では RAW ソケットを用いて該当する処理を行う。この仕組みを応用し、相手アプリケーションが利用するプロトコルに自アプリケーションが適応的に合わせるプロトコルフリー環境を実現する。

サーバ型のアプリケーションでは、インタフェースライブラリ内で利用するプロトコル用のソケットを全て作成し、**select** システムコールによってクライアントからのアクセスを監視している。そしてクライアントが使用したプロトコルに応じてサーバが使用するプロトコルを決定する。

また、UDP などのコネクションレスプロトコルの場合は、最初に呼び出される受信関数に対して **select** を適用することによって同様の仕組みを実現することが可能であるが、変更後プロトコルも同じくコネクションレスである必要がある。

5. 評価

本章では、3, 4 章で示した提案フレームワークのアーキテクチャを元にして行った性能評価実験について述べる。

以下の実験は、ギガビットイーサネットで接続された二台のマシン上にそれぞれデュアルインストールした Linux および Windows 上で行った。また、両 OS において、実験用アプリケーションは同一のソースファイルからコンパイルされている。表 1 に実験で使ったマシン環境を示す。

表 1 実験で使ったマシン環境
Table 1 Machine specifications

Machine1	
OS	WindowsXP/Linux(2.6.18)
CPU	Intel PentiumM 1.73GHz
Memory	512 MB
Network	Ethernet (1000BASE-T)
Machine2	
OS	WindowsXP/Linux(2.6.18)
CPU	Intel PentiumD 2.8GHz
Memory	4 GB
Network	Ethernet (1000BASE-T)

5.1 サービス機能追加時のオーバーヘッド

まず、提案フレームワークによって暗号化などのサービス機能を追加した際の実効スループットを評価する。比較対象として、アプリケーションコード内に直接

サービス機能を追加したものを用いる。評価結果については文献¹⁾に詳しいが、まとめると、処理量の少ない軽量サービスを追加した場合は、対象アプリケーションに対して最大で 2% 程度のスループット低下がみられた。また、重量サービスを追加した場合の性能低下は比較対象に対して最大で 1% 程度だった。

5.2 トランスポート層プロトコル追加時のオーバーヘッド

筆者らは、プロトコルの追加によって生じるオーバーヘッドを評価するために、UDP プロトコルを模倣する UDP プロトコルライブラリ／サーバを実装した。これらを用いて、元々カーネル内に実装されている UDP プロトコルスタックとの比較を行う。評価アプリケーションとして、10,000 個のパケットをバースト的に送信するものを使用した。

図 5 は Linux 上、図 6 は Windows 上で測定した受信側アプリケーションの実効スループットである。結果を見ると、まず Windows では Linux に比べて軒並みスループットが低くなっており、さらに *TCP(Kernel)* や *UDP(Kernel)* の性能が大きく低下するポイントがあることが分かるが、このような現象は Linux では見られないため、Windows の内部実装あるいは NIC 用ドライバの実装に起因するものと考えられる。次に、図 5 では、*UDP(Framework)* は *UDP(Kernel)* に対して、パケットサイズが 1024 バイトの時に最大で 27% の性能低下が見られた。しかしながら、パケットサイズが大きくなると *UDP(Framework)* の実効スループットは 800Mbps 以上にまで向上するため、実用上問題になるほど性能が低下するわけではない。一方で、Windows 上では、1024 バイト以下の時は *UDP(Kernel)* より大幅に性能が劣っているものの、1024 バイト超では性能差は小さい（最大で 8.5%）。

5.3 考察

評価実験の結果、提案フレームワークを用いた場合、L4 プロトコルを追加した場合には明らかにオーバーヘッドが存在すると言える。この原因として、プロトコルの中核処理を独立したプロセスで行っているために頻繁にプロセス切り替えが発生したことが挙げられる。アプリケーション／サーバプロセスの実行順序などのタイミング制御が最適でないとき頻繁にスリープやプロセス切り替えが発生し、性能が低下する。

そのため、効率性を最重要視する場合、アプリケーションの通信特性に応じたタイミング制御を行うプロトコルライブラリ／サーバを新たに用意するか、あるいは汎用性を犠牲にしてプロトコルサーバの処理をライブラリ内で行う方法が考えられる。

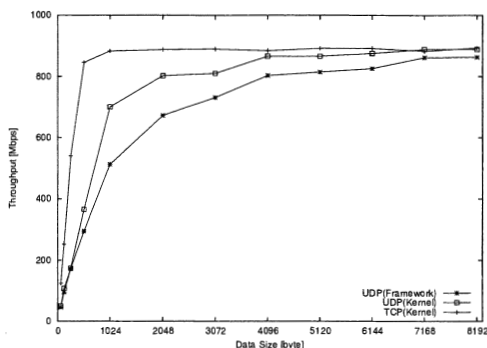


図 5 Linux 上の受信側アプリケーションにおける
実効スループット

Fig. 5 Throughput of the receiver application on Linux

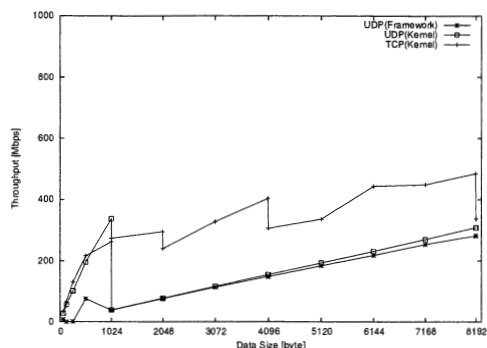


図 6 Windows 上の受信側アプリケーションにおける
実効スループット

Fig. 6 Throughput of the receiver application on Windows

6. 関連研究

MetaSocket³⁾ は Adaptive Java と呼ばれる拡張言語を用いることで、既存プログラムに対して透過的かつ動的な機能追加を実現している。MetaSocket は JVM 上で動作するアプリケーションにしか対応していないが、本システムではネイティブアプリケーションに対してでも透過的な機能追加が可能である。

TESLA⁴⁾ はリンクやローダを修正することで、ネイティブアプリケーションに対して透過的にセッション層機能の追加を実現している。しかし、他プラットフォームへの移植性が低く、またトランスポート層機能の追加には対応していない。

VTL⁵⁾ は、仮想マシン上のアプリケーションに対して透過的に機能追加を行うためのフレームワークである。L4 プロトコルの変換機能も備えているが、実際に送受信されたパケットをキャプチャして変更プロトコルのパケットに直接変換しているため、提案フレームワークとは実現方法が異なっている。

STP⁶⁾ は、モバイルコードとして実装された L4 プロトコルを、カーネル内に動的に追加させることができる。プロトコルは必要に応じて自律的にダウンロードされ、カーネル内の専用サンドボックス環境内で実行される。しかしながら、これらの機能は主にカーネル内で実装されているため、提案フレームワークに比べて移植性の点で劣っている。

7. まとめ

本論文では、QoS やセキュリティなどのサービス機能や任意の L4 プロトコルを既存のアプリケーションに対して完全透過的に追加するためのフレームワークを提案した。さらに、通信相手が利用するプロトコル

に応じて、アプリケーション自身が利用するプロトコルを適応的に変更するプロトコルフリー環境を実現した。本システムはマルチプラットフォームシステムとして設計されており、現在のところ Windows および Linux 上で利用可能である。

実装したシステムを用いて性能評価を行ったところ、本システムによって追加された UDP プロトコルは、十分なパケットサイズではカーネル内実装の UDP プロトコルに近い性能（最大 800Mbps 超）を得た。

参考文献

- 1) 川島 龍太, 計 宇生, 丸山 勝巳, マルチプラットフォームにおけるネットワーク機能の透過的拡張のためのフレームワークの開発, *FIT2008 情報科学技術フォーラム*, Sep (2008)
- 2) B.Buck and J.K.Hollingsworth, An API for Runtime Code Patching, *International Journal of High Performance Computing Applications*, Vol.14, No.4, pp.317-329 (2000)
- 3) S.M.Sadjadi, et al., MetaSockets: design and operation of runtime reconfigurable communication services, *Software-Practice & Experience*, Vol.36, No.11-12, pp.1157-1178 (2006)
- 4) J.Salz, A.Snoeren, and H.Balakrishnan, TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services, *Proc. of USITS'03*, Seattle, WA, USA, Mar (2003)
- 5) J.R.Lange and P.A.Dinda, Transparent Network Services via a Virtual Traffic Layer for Virtual Machines, *IEEE International Symposium on High Performance Distributed Computing*, Monterey, CA, USA, Jun 25-29 (2007)
- 6) P.Patel, et al., Upgrading Transport Protocols using Untrusted Mobile Code, *ACM Proc. of SOSPP'03*, pp.1-14, Oct (2003)