第 7 部

名前構造

第 1 章

序論

1.1 はじめに

現在、UNIX[1]上ではユーザやファイル、サービス等が利用され、取り扱われている。そしてそれらは、一つのマシンの中でユニークな名前を持っている。UNIX 上で、ユーザにメイルを出そうとかファイルをコピーしようという場合、この名前を用いて処理している。これまでは、マシン内で重ならなければ、ネーミング方法にとくに問題はなく、なんとなくうまくやってきていた。しかし、ネットワークという概念が取り入れられ複数のマシンがつながっていくにつれて、マシン名・ログイン名などの重なりや階層的なアドレス空間などの問題が起こってきた。そしてもう一度名前について、考え直す必要が出てきた。今までもコンピュータサイエンスの分野において、重要な問題として取り上げられ改善されてきている。しかし、十分とはいえない。そこで本論文では、広域分散環境上での名前をもう一度考え直す。さらに名前構造を定義し、名前空間を作ってみる。

1.2 現状

ここで、koch にはログインネームを yuko、potato にはログインネーム Ishikawa でアカウントを持つ人を想定する。(Figure 1.1) もし、koch の上のファイルを potato 上にコピーしようとするなら、

```
% rcp yuko@koch:file .
```

```
% rcp koch.yuko:file .
```

で、できてしまう。それは、koch の中で$ HOME/.rhosts や/etc/hosts.equiv などによって、ログイン名 yuko=Ishikawa であると認識されるからである。各ホストで、これらのマッピングを管理している。(Figure 1.2)

[1] UNIX は AT&T の登録商標である

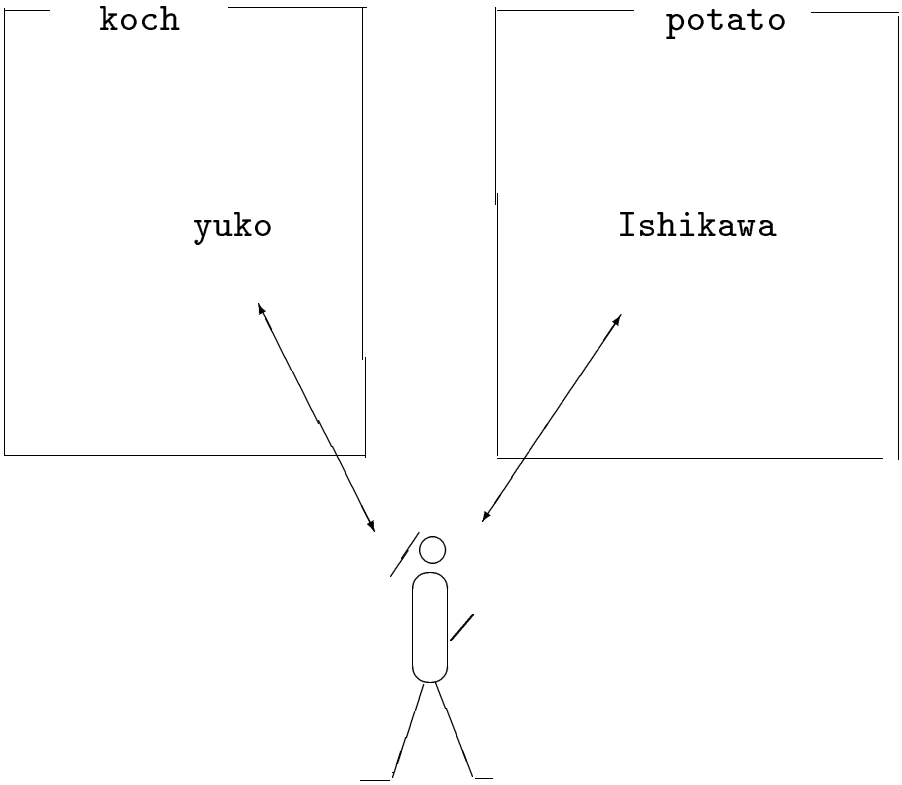


図 1.1: ユーザとログインネームとの対応

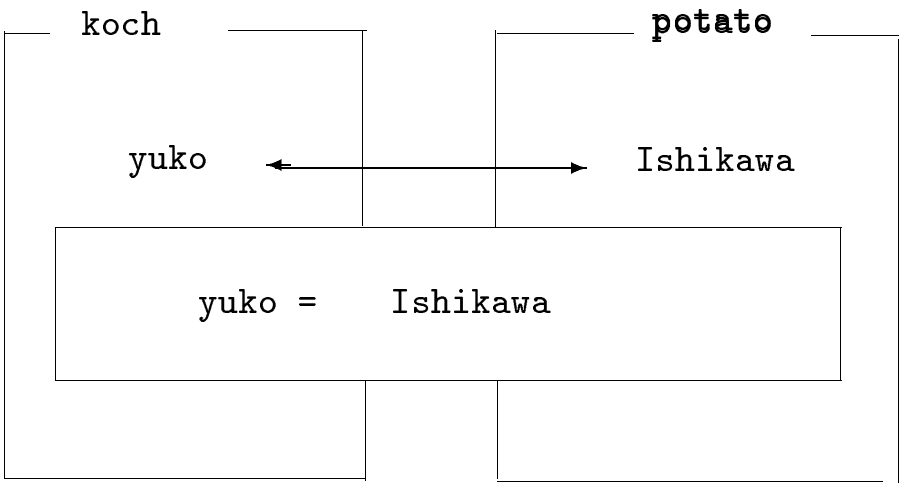


図 1.2: ホスト間の、同一ユーザに対するログインネームの対応

しかし、ホストが N 個つながっているとすれば各ホスト同士が１対１でマッピングするばあい、$N(N-1)/2$ 通りのマッピングの仕方が考えられる。これは、 N^2 に比例することになり破綻する。名前構造を考え直していかなくてはならない。このように、ログインネームを実体との対応で考えるようにすれば (Figure 1.3)、ログインネームは名前の一部と考えることができる。そうすれば、たとえば複数の人が同じログインネームを持つことも可能になってくるはずである。そして、そのようなことはネットワークがつながってくるこれからの環境として、大いに考えられることである。

UNIX では、この他にホスト名を管理するネームサーバの機能が提供されている。これは、ホスト内の資源の名前とホスト名との組合せによって、全体での一意な名前付けとなっている。また、 NFS による分散ファイルシステムが構成する名前空間や YP という分散データベースによる管理によって、発展した名前管理が実現されている。

1.3 本研究の目的

次のような点に気をつけて、名前空間を作っていく。

- なるべくユーザに対し、名前空間内のドメイン間の物理的関係を意識させないもの
- ネットワーク上の資源が物理的に移動しても、名前空間やドメイン構造の論理的な構造は変える必要がないもの
- 名前（ドメイン名も含む）は、実体と密接な関係を持つ

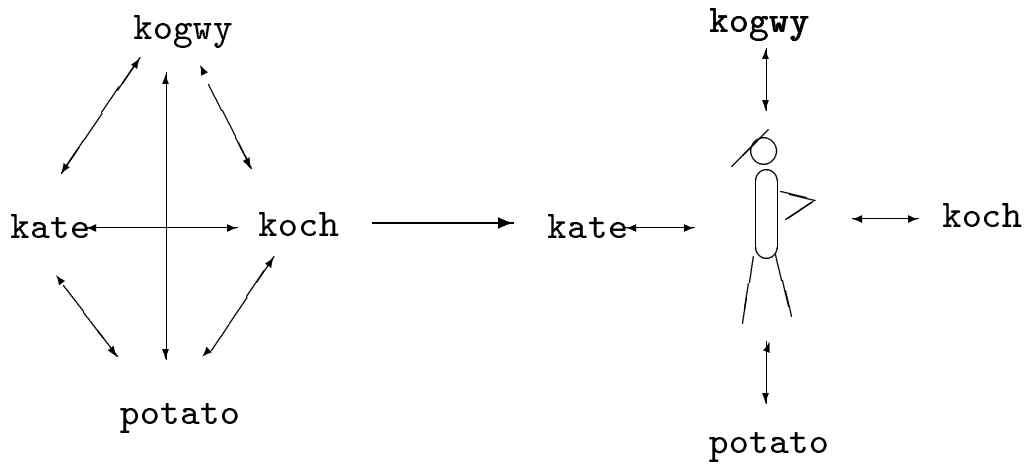


図 1.3: ログインネームとユーザとの対応

第 2 章

名前空間

2.1　名前

2.1.1　名前の概念

名前がなかったら、大変である。もしもこの世の中に名前がなかったら、いろいろな物や人、事物がその辺に転がっているだけで、誰もそれを話題にしたり、示したり、伝えたりできなくなってしまう。たとえば、小学校のお昼休みの運動場の様なところを考えてみる。もし名前がなかったら、先生が　“さいとう君” を呼びだそうとしても、“ほらそこのあなた” とか、“赤いセーターを着た、背の高い少年” などということになりとても不便である。何百人もいる小学生の中から、その少年を一人だけ呼び出そうとするためには、まず彼を他の小学生と区別しなければならない。また、先生はそのことを誰かに頼むかもしれないので、頼まれた人がその少年は誰のことかを理解する必要がある。名前があれば、先生は “さいとう君” という名前を使って、彼を呼び出すことができる。私たちの日常生活において名前とは、人や物、事物を表すラベルであると考えられる。ここでは名前を “一意に実体を示すことのできる、識別子” と定義することにする。名前は、実用的なものでなくてはならない。名前を使って、自分だけでなく他の人もその実体を識別できなければならない。そのために、紛らわしかったり間違いやすいものであってはいけない。もしそうであると、“一意に識別する” という名前の定義に反することになってしまう。いい名前であるとか悪い名前であるとかいう判断は、その名前が示す実体の目的に、一致しているかどうかによる。もう一度 “さいとう君” の例で、考えてみる。“さいとう” という名字をもつ人はとても多いので、“さいとう君” とだけでは何人もの少年が集まってしまう。そうならないために、下の名前がある。名字は、そのうちの家系、出身地、時代に強く影響している。そのため、“鈴木”、“斎藤” 等の名字をもつ人が多く、名字だけでは他人との区別がつきにいので、下の名前も使うことになる。そこで、下の名前はなるべく個性的で使いやすく通じやすい名前をつける方が、名前の目的に一致してい

る。名前は、識別性と伝達性を満たすことが大切である。社会的には名前は、二つの側面を持つ。一つは、“ 先生” などのように、人それぞれの立場、環境、関係によってさす実体が、異なるような名前。つまり、一つの名前がたくさんの意味を持っている。これに対して “斎藤信男教授” という名前は、誰がどの様に使おうともただ一人をさすことになる。そして、人の頭の中ではこの二つの対応が表となって存在し、友達と話すときは “先生” を使い、何か特別な用紙に書く場合は、“斎藤信男教授” となって表現される。名前が、単一的な実体をさすとは限らない。“斎藤” という名前は、時には “斎藤信男研究室全体” の集合をさす場合もある。このようなときには、この複数の意味を持った名前が、一体どちらをさすものなのかを区別する決まりが、必要となってくる。

2.1.2　コンピュータの世界における名前

コンピュータの世界においても同様の事がいえる。名前は各システムにおいて、意味を持つことができる。システム上の、ユーザ、マシン、機能、ファイルなどがそれぞれ固有の名前を持っている。ユーザは、マシンに対してログインネームを持ち、これによっていろいろなサービスを受けることが可能になる。またこのサービスも名前を持ち、ユーザはコマンドとしてそれを指定することによって、必要なサービスを得ることができる。MS-DOS 上で `type aaa.txt` と入力すれば、ユーザの考えた通り `aaa.txt` というファイルの内容が画面に出力される。誰も `type` がユーザ名であるとか、マシン名であるなどとは考えない。このように、暗黙のうちにユーザの期待したサービスと実体とのマッピングを構成し管理しているのが、コンピュータの世界における名前構造であるということができる。

2.1.3　ネットワークの世界における名前

上の場合は、定義域を一つのマシン上に想定していた。つぎに、各定義域を結んだネットワークド メインという環境に広げてみる。(Figure 2.1) すると、各定義域でそれぞれなされていた管理だけでは、不十分となってしまうのは明らかである。つまり、ネットワークというのは、ただ回線によって複数の定義域を結んだだけではない。そこで、ネットワーク全体をまとめるためには、今までのローカルな管理方法から外のネットワークワイドにおける管理へと、変えていかなくてはならない。さらに今あるメイルのアドレス構造を生かして、階層的な名前空間を考える。

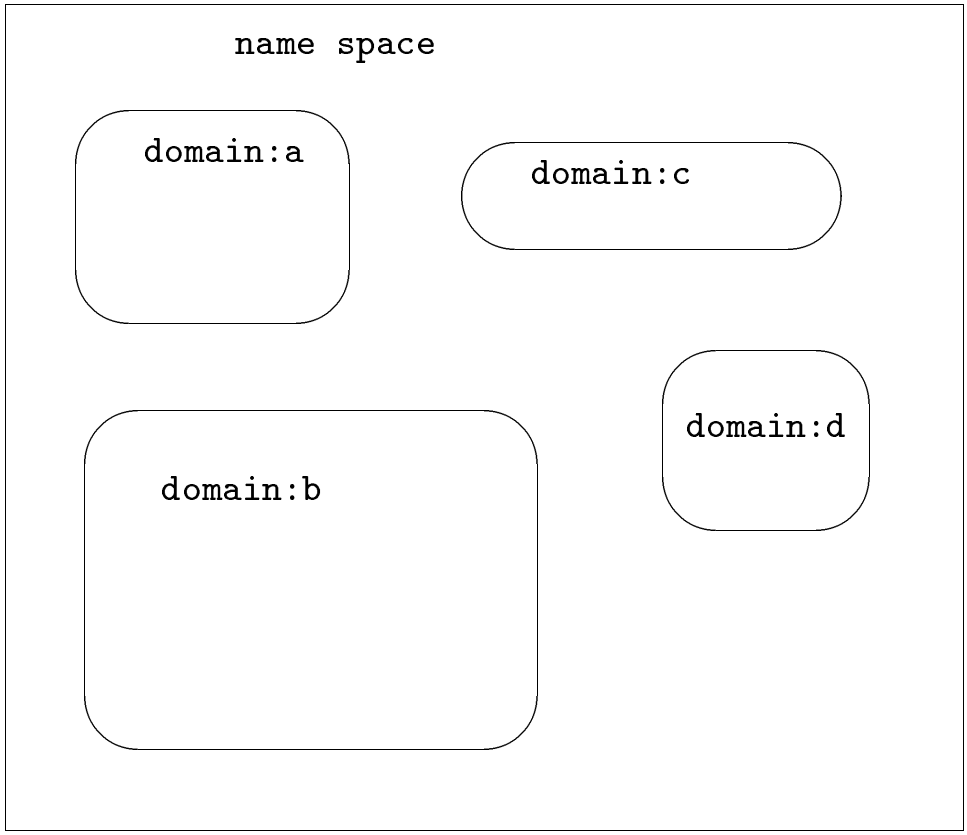


図 2.1: ネームスペースとドメイン

2.2　名前空間と ID 体系

人間は、何らかの形でファイルにアクセスしようとする時、ファイル名を使う。これは UNIX においては、８文字以内の文字列であると決められている。しかしマシンの側では、このファイルを i-node という数字の列で認識する。また、NFS おいては v-node、プロセスはプロセス番号を、ユーザやグループはそれぞれユーザ ID・グループ ID をそれぞれ持っている。これらをここではまとめて、ID 体系と呼ぶ。一つのオブジェクトは、憶えやすい名前を提供する人間側の名前と、ユニークなアドレスとしてのマシン側の ID の二つを持つ。そこで、分散環境においてどのように ID のバインディングを取り扱うかどのように名前付けを行なうか、どのようにユニークな ID を決定していくかを考える。

2.2.1　ID システムの一般的特徴

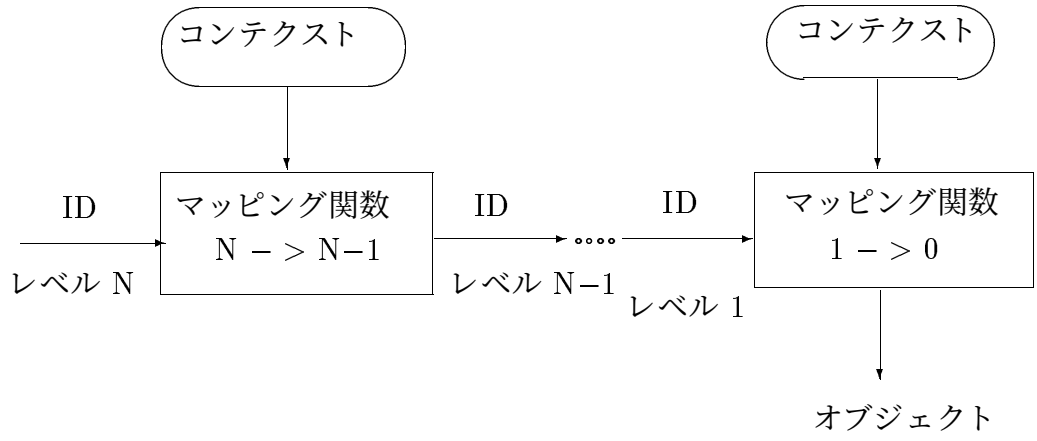


図 2.2: ID とマッピング関数

- ID はいろいろな目的（例えば、参照・配置・スケジューリング・エラー処理・並列処理・資源の共有など）に使われる。ID を使うことで、より複雑な要素を構成することができる。

- ID は、システム内のすべてのレベルで異なった形をしている。
- インタープロセス通信においては、ポートの物理アドレスに対し、時には適当なマッピング関数やコンテクストを使って、他の ID へとマップされる。一般的には、名前ー＞アドレスー＞ルートのようにマップされる。(Figure 2.2)
- マッピングのメカニズムは、あるレベルの ID を次のレベルの ID へ結びつける働きをする。もしオブジェクトが共有されたり、作られたり、壊されたりした場合には、コンテクストの更新のメカニズムが必要となってくる。
- ID・プロテクション・エラー・資源管理のメカニズムは、互いに影響しあっている。

ここで重要なことは、動的なバインディングである。必要な時にアドレスやルートにバインドされることにより、オブジェクトの動的な再配置・拡張・共有が可能になる。このオブジェクトの動的な再配置とは、実行時つまり必要な時にオブジェクトの再配置を行なうことにより、システムの効率を良くする。そのために、もう一つのネーミングのレベルを設け、途中の名前一時的にある論理アドレスにバインドするようにした。そしてこの論理アドレスは、実行時に物理アドレスにマップされる。(Figure 2.3)

ID システムを分散的に構築していくことにおける問題点は、要素となるシステムが混在していることである。そしてそれぞれが、独自のメモリアドレッシング・ファイル・プロセス・ID システムを持っていることである。さらに、メッセージ転送時における遅れ・システムのクラッシュなども考えられる。そこで次のような点を課題とした。

- モデルとなるシステムの各レベルに、どのような型の ID が必要か。
- 何処でコンテクストが保持され、レベル間の ID マッピングが行なわれるべきか。
- どのようにシステムの混在を取り扱うべきか。

2.2.2 ID 体系の目的

- 二つのレベルの ID をサポートする。一つは人間にとってわかりやすく、もう一つはマシンにとってわかりやすいもの。これらは独立のメカニズムである。マシン側の ID はビットパターンで、マシンによって容易く扱われ保持される。そして、保護や資源管理などにも便利である。一方人間側の ID は、人間の読める文字列となっている。

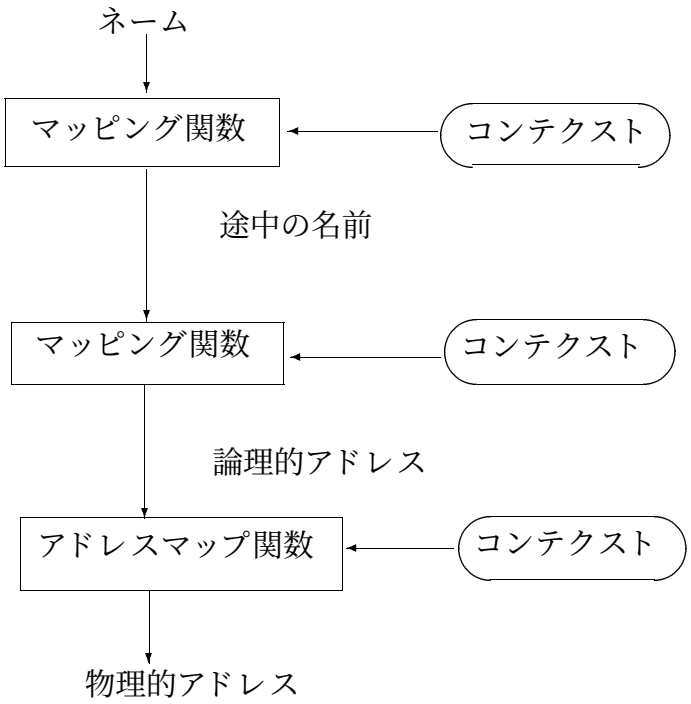


図 2.3: ネームからアドレスへの変換

- 分散環境において、ユニークな ID を生成する。それにより効率や信頼性の問題を解決する。

- ID 空間をホストがローカルに持つ空間とは考えずに、グローバルな空間としてみる。また ID のメカニズムは、システムの物理的な連結とは独立であるべきである。

- 名前とアドレスという二つのレベルがあり、その間は動的にバインドされる。またオブジェクトが移動すれば、コンテクストも更新される。

- オブジェクトの複数のコピーを許す。それが書き込み・変更可能ならば、制限が厳しくなる。この時複数のレベルでの ID と、動的なバインディングが要求される。

- 複数のローカルユーザが同じオブジェクトに対し、それぞれの ID を定義する。そこで、グローバルな ID とローカルな ID とをバインドするメカニズムが必要とされる。

- ID が重なることなく資源の共有、コピーができること。そのためには複数の ID ドメインを作り、メカニズムを切替えるコンテクストが必要である。そして、どのコンテクストを用いたかというコンテクスト ID と、オブジェクト ID の二つの ID 空間を考えなくてはならない。

- ブロードキャストやグループ ID などのように、複数のオブジェクトが同じ ID を共有することもある。そのような時には、複数レベルの ID と 1 対多のマッピングが必要となってくる。

2.2.3 ユニークなマシン側の ID

グローバルな ID 空間を構成するために、オブジェクトは各 ID 空間に対して ID を持ち、それを互いに連結させる。これは、(コンテクストの ID、オブジェクトの ID) の組になる。しかしこの欠点は、ID の型や長さが長く効率が悪い。また、コンテクストの定義域の境界を明らかにしなくてはならない。そこで、統一的な ID 空間・グローバル空間内のローカルドメインを区切る方法を発展させる。ローカルな ID とグローバルな ID とのマッピングは、永久的な時もあれば一時的な時もある。ローカルな ID は、ローカルな環境で最適のものを選択することができる。

次に、ここでの資源やオブジェクトは、

- サーバプロセスにより管理される。

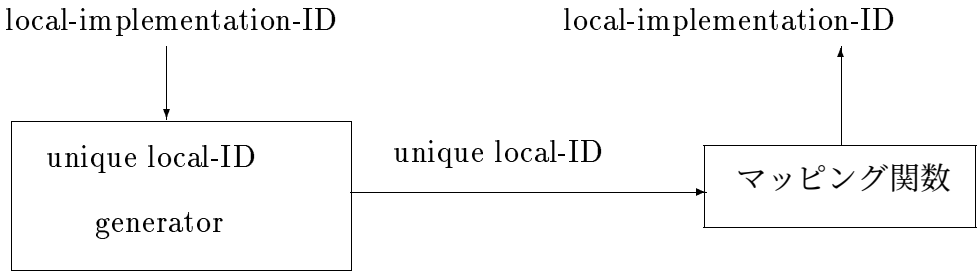


図 2.4: local-implementation-ID と unique-local-ID

- プロセスがサーバプロセスに、要求メッセージを送ったり答えを受け取ることにより、管理される。

などが想定されている。そこでこれらにつく object-ID は、(server-ID,unique-local-ID) の形で構成される。さらにこの unique-local-ID は、サーバによってサーバのローカルドメイン内で一意な local-implementation-ID にバインドされる。(Figure 2.4)(Figure 2.5)

もしオブジェクトが複数のコピーを持っていたら、それぞれが local-implementation-ID を持つ。そしてオブジェクトが移動すると、local-implementation-ID も更新される。いずれも object-ID は同じものを持つ。

IPC レイヤーでは、すべてのプロセスが IP（アドレス）を持つ。これは、ポート・ソケット・メイルボックスなどにバインドされる。もし一つの ID が複数のプロセスに対応していると、サービスの型によってプロセスの一つに決める場合に便利である。(Figure 2.6)

プロセスは二つの型の ID を持つ。一つはプロセスサーバのアドレスが一部を構成する object-ID。もう一つは、サービスに関するメッセージを送るアドレス。

プロセスのアクセスメカニズムは、アクセス権やアクセスリストの管理を行なう。アクセスリストは、プロテクション情報やパスワードを含む。

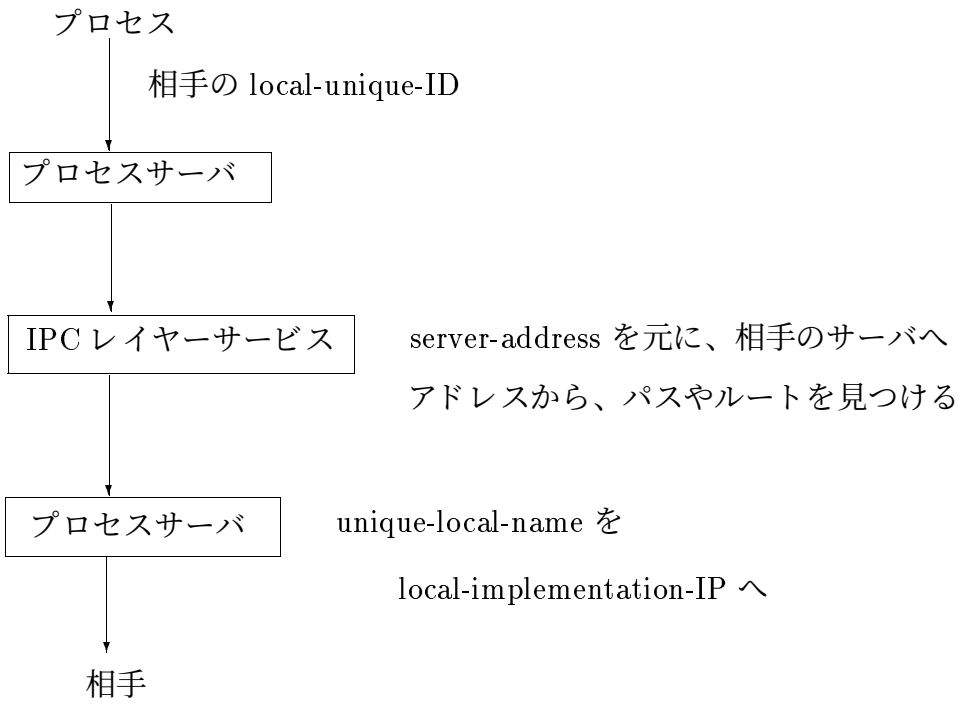


図 2.5: local-unique-ID によって相手を見つけるまで

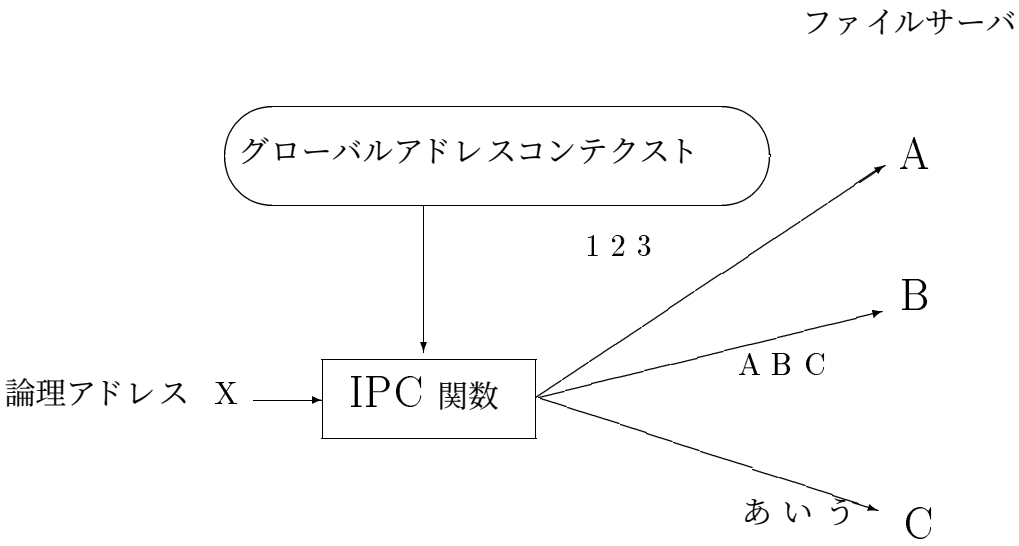


図 2.6: 一つの ID が複数のプロセスに対応している場合

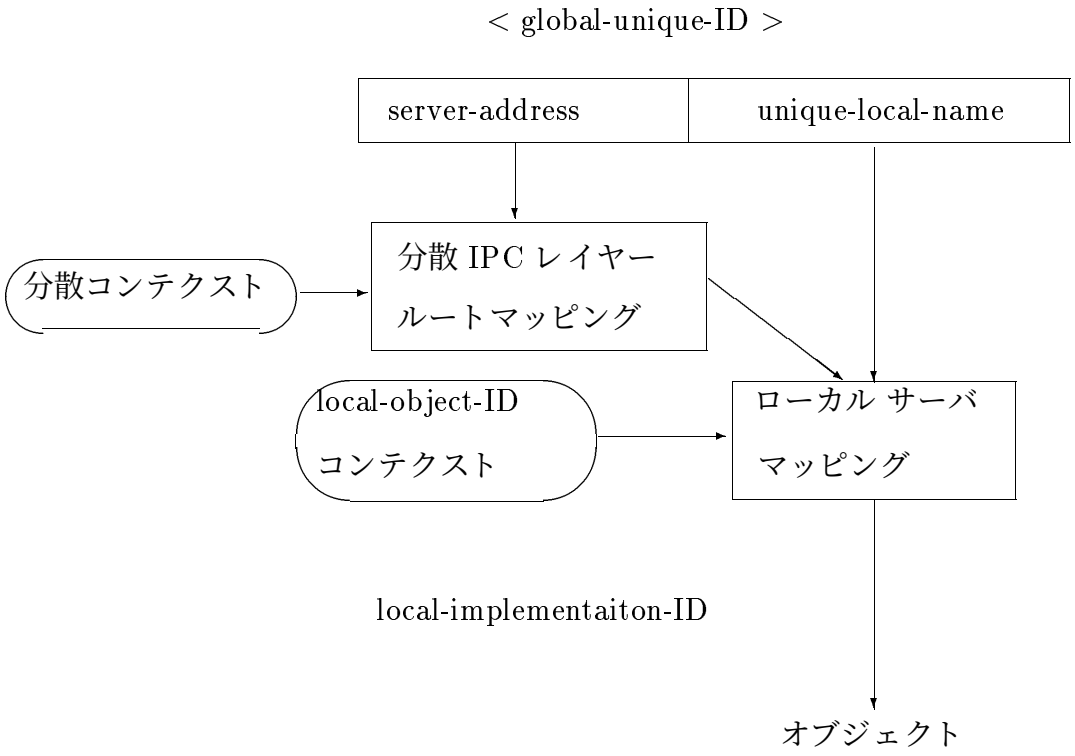


図 2.7: global-unique-ID のフィールド

< object-ID >

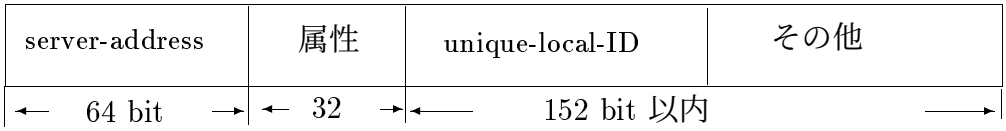


図 2.8: object-ID のフィールド

object-ID のフィールドは、Figure 2.8のようになる。

2.2.4　人間側の名前

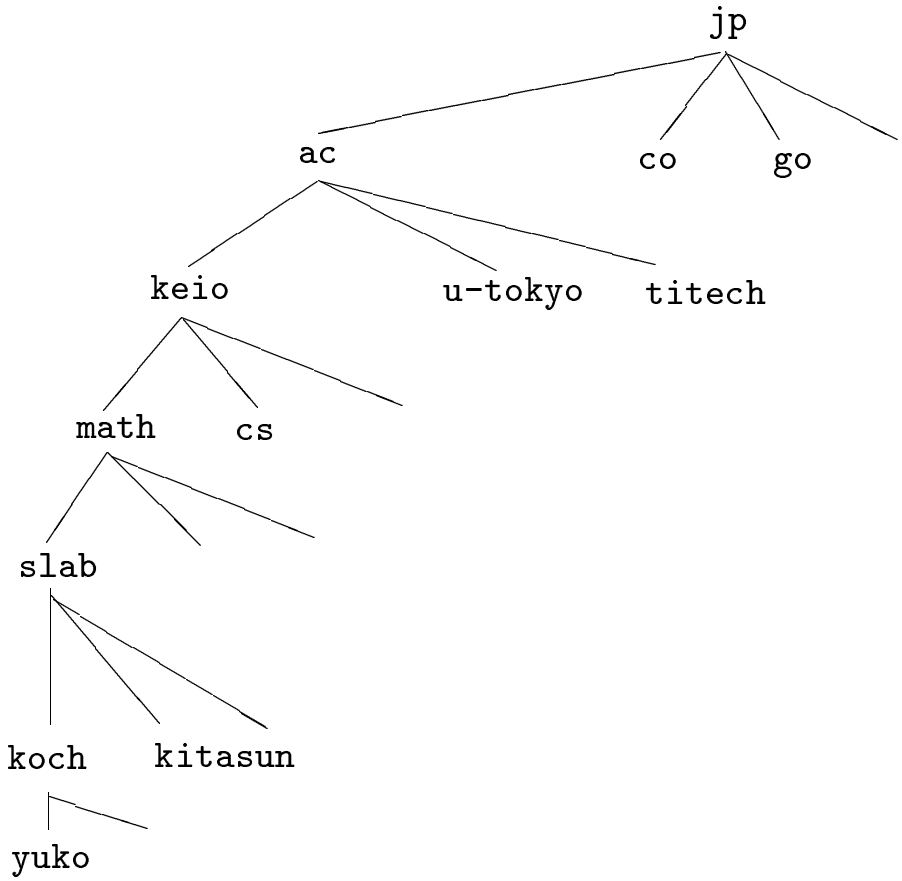
高レベルにおけるオブジェクトの名前付けは、ユーザの要求に合わせて構成され互いに関連し、オブジェクトを共有させる。また、複数のコピーやオブジェクトの移動もサポートする。このコンテクストは、ネームサーバによって管理される。ネームサーバは、(name, マシン側の name) のコンテクストを用いて、人間側の名前をマシン側の名前にマップする。一つの例として、マッピングが何段階も行なわれるメールサービスがあげられる。例えば、

```
% mail yuko@koch
```

とした場合、“koch” はこのホストのメイルサーバのアドレスへ、“yuko” はメイルボックスのパス名へとマップされる。

2.3 アドレス空間と名前空間

名前だけでは、その実体を一つに決定することはできても、いざその実体に触れようとするとやはり困難であることに気が付く。つまり、“斎藤信男教授” という名前によって、一人の人を想定することはできた。しかし、話をしようとか手紙を出そうとか会ったりしようとするときには、これだけでは何もできない。そこで、今度は “斎藤信男教授” という名前が指す人が住んでいる住所だとか、電話番号といった情報が必要になってくる。または、慶應大学の理工学部数理学科斎藤研究室といったものでもよい。そこで、住所録や電話帳で調べたりして、なんとか実体に接触しようとする。　このように、名前と実体とを結び付けるための手段が必要になってくる。それがアドレスである。名前に意味をもたせ、実用的なものにするには、名前とアドレスを対応させるメカニズムもなくてはならない。名前とアドレスは、同じものとして考えられがちである。なぜなら、これらは実体と１対１に対応し、密接に関係しているものであるからである。しかし、これらはまったく違うものである。アドレスとは、実体の居場所を物理的に示す。そして、実体の内容を示すような意味的な要素は含んでいないと考えられる。ただ単に情報を伝達していく上で認識されるような、形式をとっている。そしてこの形式は、手紙だったら住所、電話だったら電話番号というように、各システムにおいて異なっている。システムそれぞれが、物理的要素やドメイン構造に依存して、利用しやすいような形をとっている。そして全体において有効な意味を持ち、統一的な空間を形成することが必須である。ここで、電話番号システムの例があげられる。たとえば、斎藤研究室の電話番号は 04**-**-**23 であるが、これは、階層構造的にアドレス空間を作っている。（市外局番-市内局番-回線局番）の３階層をなし、各階層において最終的にたどり着く場所を定義していく。コンピュータの世界でも、同じ様なことが行われている。電話番号システムと同じようにネットワーク内の資源のアドレスも、階層的に定義されている。つまり、ネットワーク内で全資源が構成している名前空間を、そのままアドレス空間としてみてしまう。そして、空間を木構造に分割し、各ノードの層をトップレベル、セカンドレベル … というように分ける。そして、そのままアドレス表現として、並べていく。(Figure 2.9)




```
% mail yuko@koch.slab.math.keio.ac.jp
```

図 2.9: アドレス空間

2.4 アドレス空間の階層構造

2.4.1 アドレッシングとルーティング

アドレス付けと経路制御は、互いに関係している。ここで設計上の問題点として、アドレス空間の規模と構造があげられる。特にアドレス空間の構造は、次のような点を中心に考える。

- どのようにアドレスが経路に影響するか。
- 各ノードに要求されるマッピングの規模。
- マッピングの分散化。
- ネットワークの拡張と再構成のしやすさ。

アドレスの形式は、フラットと階層構造がある。

＜フラットな場合＞

- 各ノードが、ルーティング可能なアドレスへのマップのコンテクストやテーブルを持つ。
- プロセスが制約無く移動できる。ネットワークサーバが、環境のどこでも移動できるようなプロセスを要素としている場合適する。
- キャッシングやブロードキャストなどのメカニズムにより、コンテクストの大きさを削れる。
- これによりプロセスの移動が楽になれば、より高いレベルのシステム設計が単純化される。

＜階層アドレスの場合＞

- 各アドレスが持つアドレス対ルートのコンテクストが、小さい。よって、ネットワーク向きである。
- ノードのルーティングアルゴリズムは、アドレスフィールドの値を外へのリンクの一つにマップし、次のフィールドはそのまた次へとマップされる。(Figure 2.10) そのため、すべてのノードがすべての相手のルーティング情報を持つ必要がない。

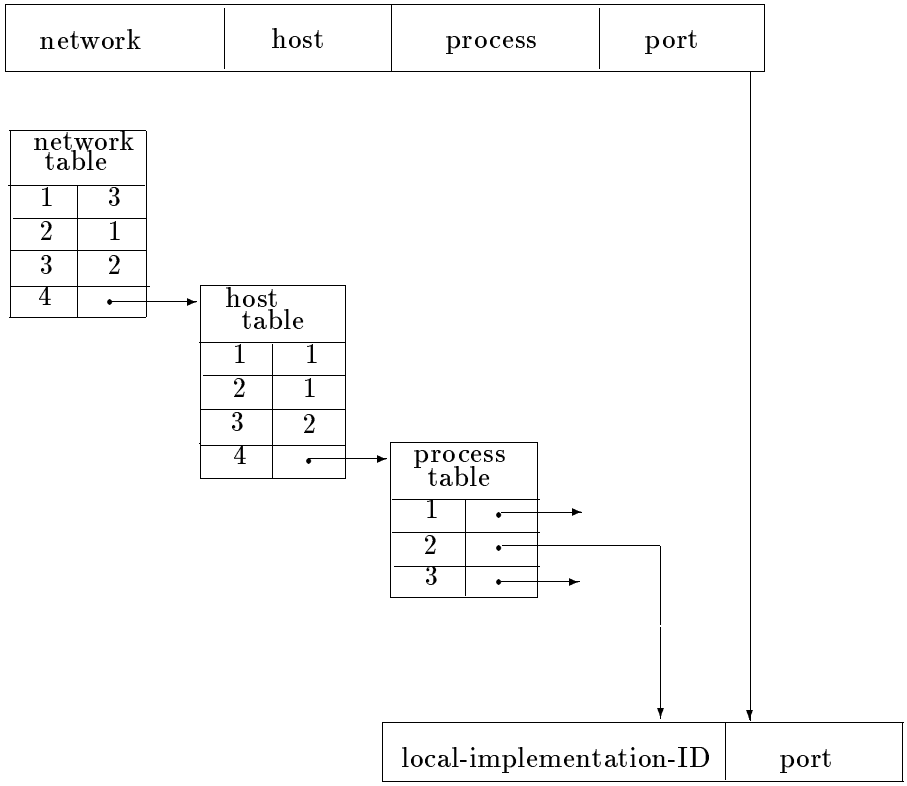


図 2.10: 階層構造のアドレスフィールド

＜IPC プロトコルのアドレス空間＞

IPC プロトコルの各レベルでは、異なるアドレス空間を用いている。そのために、

- 能率を良くするために高レベルでの複チャネルを、低いレベルで一本化できる。
- 信頼性やパフォーマンスを上げるために、高レベルで一本のチャネルを低いレベルで多重化できる。
- 低いレベルではプロトコルが同じでも、上のレベルでは異なるプロトコルが使える。

第 3 章

ドメインについて

3.1 なぜドメインが必要か

ここでは、ドメインを ”管理的・地域的境界で区切られるホスト・人の集まりと定義した。このように、ホスト・人をいろいろな側面（特に管理・組織面）で区切ることの必要性について、以下のように考えた。

1. 管理の分散化・階層化が行なえる。

   集権的な管理を、各ドメインごとの責任に分散できる。これは、集中管理から分散処理への動きの中で、当然考えるべき問題である。つまり、あるドメインは自分の下にあるドメインやサブドメイン・ホストの管理のみを行なう。しかも、自分の一段下までは責任を持つが、それ以上は下のドメインに任せることにする。またあるオペレーションに対し、自分が分担された仕事のみを果たす。例えばメイルの転送を行なう時には、一箇所で集中的に管理するのは、不可能である。またそれをいくつかに分け、どのドメインがどのようなゲートウェイを持ち、どのように経路制御を行なうかというグローバルな情報を、一層のドメインを代表するゲートウェイが持ち適当に更新していくとする。このような時にも、たとえこの更新を自動化した所で、各組織の管理者の負担は残る。また、ゲートウェイ間の一貫性の問題もある。しかしその情報をさらに下のドメインへ分散していくことによって、トラフィックが小さくなる。ある一点の故障が、すべてに影響するのを防ぐことができる。

2. アドレスの管理を階層化・再構成できる。

   これはアドレスに関することで、メイル・経路制御などを、ドメインにしたがっておこない処理するということである。ドメイン構造を基準としたアドレス体系を作る。これは、今までのメイルアドレスの概念である。

```
% mail yuko@koch.keio.ac.jp
```

とした時の keio.ac.jp は、ユーザが属するドメインを小さい方からつなげて書いたものである。そこで、アドレスに関したオペレーションが行なわれやすく、またユーザにとってもドメイン構造さえわかれば、それに従ってアドレスを指定するのでわかりやすい。名前の管理が楽に行なえる。また、もしネットワークとネットワークとを結んだインターネットを考えても、容易にアドレスを拡張できる。例えば、今までは日本のネットワークは実際一つのドメインを形成していたが、はっきりと決定しているわけではなかった。メイルアドレスのみ junet となり、他は便宜上この形式を用いているにすぎなかった。しかし今回の jp 化の提案により、国際的にメイルアドレスとして割り当てられた ‘‘jp’’ というアドレスを、日本国外ともにトップドメイン名として、明示したことになる。そしてドメインを、アドレス管理の基盤とすることができた。

3. 全体的な名前管理の統一

   ドメイン名は、それが代表する組織を意味的に表すものでなくてはならない。さらに、その下に何段もの階層構造が考えられるので、できるだけ短くする。そして広く使われる（他の組織からも）ことを想定し、上層にあるドメイン名とは重ならないようにする。混乱を招きにくく、ユーザの使いやすい名前をつける。

4. 経路制御などのメカニズムが改善できる

   膨大な数のホストがつながれた現在、経路制御は特に重要な問題として取り上げられている。例えば、物理的には近道があってもそれを公共的に通ることは許されていないとか、ある場所でリンクが切れてしまったら、別のルートを通るとか。また、エラーとなったパケットの処理なども含まれる。それらの問題も、各ドメイン内にゲートウェイを一つずつおき、それらが自分の責任となる情報を保持し、互いに横と連絡を取り合いながら削除・更新していくことによって解決される。

5. メールなどのフォーマット化が可能になる

   これからは、メイルだけでなくコマンドの引数にもファイル名・ユーザ名・ホストやドメインなどの指定が必要となっている。その時にドメインをきちんと定義し区切っておけば、それらのフォーマットかもドメインに沿って行なえる。

3.2　どのようにドメインの境界を区切っていけば良いか

1. ドメインは、物理的なネットワークの接続の仕方には依存しない。

   今までも、そしてこれからもホストが次々に接続され、ネットワークが大きくなってくると、一つのホストまたは一つの組織が相手に到着するのに、いろいろな経路が考えられることになる。例えば、慶応内のホスト kitasun と東大の ccut の間には、kogwy（慶応）-nirvana（東工大）-ccut（東大）間で UUCP が張られていた。しかし今度、koch（慶応）-ccut（東大）間に専用回線がつながれたおかげで、二つの異なった経路ができた。このような時、もし接続によってドメインが決定されていたら、またここでドメインを再構成しなくてはならない。さらにホスト間の接続、さらにはネットワーク間の接続は、これからもどんどん行なわれていくであろう。また、同じ会社でも、東京支店と大阪支店とは、物理的にはなれてしまっている。しかし管理的には一つの組織であるので、同じドメインの方が便利でわかりやすい。

2. 静的なドメイン

   （１）とかなり重なっている。ネットワークの構造は、たびたび変化する流動的なものである。しかし、この変化に動じないドメイン化を行なう。そうすれば、アドレスはドメインに依存しているわけだから、アドレスを変えることもなく、ユーザもその影響を受けることがない。

3. ドメインは、ネットワークの管理的組織である。

   ドメインは、その下のサブドメイン・ホストを管理する母体・ポリシーを持つ。つまり、メイルや経路制御のためのゲートウェイ・ルーティング情報などを保持している。ネットワークの構造は、一つの組織の中でいくつかの固まりになり、他の組織との間にはいろいろな意味で隔たりがある。それをとりしきるゲートウェイなどが、ドメインには必ず必要となってくる。

3.3　ドメインとは

最後に、ドメインを次のように定義した

1. ネットワークまたはそれに準ずるものの管理組織である。

2. 内部を管理していく実務・ポリシーがある。

3. 管理する母体（ネームサーバ・アドレッシング・ゲートウェイ・ルーティングを管理する機能）との密接な関係

4. 実社会の組織と結び付けて考える

3.4 ドメインをつなげる

現在 **WIDE** ネットワークでは、jp 化にともなって、jp の下に ac, co, go, ... など、各組織を企業・学校などの教育関係・政府とそれぞれまとめて、もう一つの階層を作ることにした。それは次のような理由によるものである。

一つ一つのドメインをつなげて新しいドメインを作るような時にも、多くの問題が出てくる。なぜなら、各ドメインが独立した組織や、異なった機能をもって活動しているからである。つまり、それらの異なった種類のドメインをつなぐのが、**WIDE** ネットワークということになる。当然それらを総括する時には、いろいろな制約を受けることになるであろう。しかし、その中の限られた範囲で通信するのが **WIDE** ネットワークの目標の一つではないだろうか。ここでは、

- 様々な特質のドメインが、それぞれ異なった社会的役割を果たす
- これらの役割が様々な形で、密接に相互に関連しあっている

この時、それぞれに独立性を持たせながら、どのようにまとめていけば良いかを考える。

3.4.1 全体構造と各ドメインの役割

まず第一に全体集合というものは、おそらくそれ自体の存在を続けていくこと以外には、明らかな目標を持っていないことが多い。かつまた全体というのは、様々な組織によって構成されているため、それぞれの役割分担が、全体的な目標の下に直接的に把握できる形で確立されることは、難しい。そこで、**WIDE** ネットワークを考えた時には、どのようなドメインがどのように機能し、どのような役割を持って **WIDE** に参加したら良いかを決める基準がなくてはならない。そしてその基準は、実際に適用されなくてはならない。

次に **WIDE** ネットワークの構造と、各ドメインの機能・働きはどのような関係があるかを見極めなくてはならない。それは、相異なる役割を持つドメインは互いにどのように結び付いているかによって、大きな影響を受ける。それは、各ドメインがお互いに対面することなく直接的には依

存しあっていないように見えても、実は間接的に関係しているからである。例えば、各ドメインが自分の目標を達成しようとした時に、他のドメインとの間で資源の交換をしなくてはならない。ここでもその間の媒体として、**WIDE** のような組織や決まりが必要となってくる。この組織や決まりは、各ドメインをまとめあげかつそれらが独自の目標を達成する手助けともなる。つまり各ドメインは、自らの個人の目標を追求しながらも、全体の目標達成の力の一部となり得るのである。

第三に、各ドメインの構成は変化しやすい。そしてそれらの働きも、不断に変動していく。そこで、上で述べた決まり・基準は安定した形でなくてはならない。どんなドメインも、ただ単に相互連結しあっているわけではなく、いくつかの系列の下に分けられる。それらはドメインが持つ役割によって分類され、さらにいくつかの段階

や階層にも分けられる。そして各層が、何らかの形で他の層の上か下かに位置付けられる。これが階層構造をなしているといえる。

3.4.2　全体の目標と個々のドメインの目標

相異なるドメインが、それぞれ異なる目標を持っている。ここでその目標とは何か、また異なるドメインがどの程度まで同じ目標を持つかが問題となってくる。

＜各ドメインの目標＞

- その外の **WIDE** 全体的な環境を自らを適用できる。
- 独自の一つ以上の目標を持つ。
- そのいくつか下のドメインの相互依存関係を保持し、一つのドメインへのまとまりを管理・維持する。
- 全体集合内での、自分の立場や役割を認識する。

ここのドメインは **WIDE** 全体集合の中で、自律性を発展させることができる。つまり全体や他のドメインから、独立した目標を発展させ、しかもそれらを効果的に追求することができる。それらの自律性が、全体の階層構造を決定する。

＜ **WIDE** 全体集合に関する問題点＞

- 各ドメインの目標が全体の中で、どのような位置を占めているか。

- **WIDE** 全体の目標や基準は、どのように作られそのようにそれらを維持していくのか。
- 次にそれらの目標をどのように制度化し、個々のドメインに受け入れさせることができるのか。
- 全体目標を達成するために必要な、社会的メカニズムをどのように作るか。
- いくら小さな役に立たないドメインであっても、全体構成を見直す時には、一つの成員として見い出さなくてはならない。

3.4.3 きまりについて

＜きまりの定義＞

個々のドメインの活動を全体集合の内部に組織化し、その目標に密接に結び付いている存在。

＜きまりの目的＞

- 個々のドメインの目標に対応する。また、一般的な目標にそれらを結び付ける。
- 全体的な目標を達成するための、組織的な枠組を作り上げる。
- 個々のドメインの目標を達成する上で、必要な資源を他のドメインからもらったり、全体的に配置付けしたりする。
- 他人が他のドメインへアクセスする時に、従うべき物。

3.4.4 最後に

個々のドメインの役割が、全体においてどのように展開するかを理解するには、まず第一にドメインの諸活動をある程度分析しなくてはならない。ここで一方では、個々の自律性、他方では全体的秩序という厄介な問題に出会うことになる。しかしある程度まで、個人が全体の目標を自分自身の目標の一部として考えれば、解決の方向に向かうことができるのではないか。

第 4 章

既存の名前空間

ここでは、ドメイン型の名前・ARPA Internet のメイルシステムにおける利用・ホストアドレスのサポート・その機能のためのプロトコルサーバを紹介する。

4.1 はじめに

4.1.1 ドメイン名の必要性

- アプリケーションが、複数のホスト・ネットワーク・インターネットワークに及ぶにつれて、複数の管理境界やオペレーションの方法も、拡張されなくてはならない。

- ARPA Internet 内のシステムが大きくなるにつれ、ホスト名と ARPA Inter net アドレスとのマッピングが必要になってきた。ここで、NIC(Network Infor mation Center) 内のホストを一つのテーブル内で集中管理するのは不可能なので、分散データベースが必要になってきた。

4.1.2 メイルの場合

- まず、資源を参照する時に使われる、不変の名前空間が必要である。そして暗号化などの問題から、名前はアドレス・ルートなどの情報を含んではいけない。

- データベースのたて方向の大きさや変更の回数に加えて、キャッシュを用いた分散化が必要である。また名前の追加や削除も、分散化されるべきだ。さらに名前を用いて、ホストアドレスやメイルボックスなどの情報も得られなくてはならない。

4.1.3 ソルーションの要素

DOMAIN 名空間は、ツリー構造の名前空間である。各ノードやリーフは一連の情報を持ち、キュエリーオペレーションは特定の情報セットから、ある型の情報を引き出す。例えば ARPA Internet では、ホストを認めるのにドメイン名を使い、資源のアドレスをたずねるキュエリーには、ARPA Internet host address を返す。しかし、ドメインメカニズムの一般性を保つには、ホスト名やアドレスなどの情報と１対１に対応しているだけではいけない。

ネームサーバは、ドメインのツリー構造について情報を用いている。一般にあるネームサーバは、ドメイン空間のあるサブネットについては完全な情報をもち、他のネームサーバへのポインターとなっている。ネームサーバが情報を持つドメインの一部を ZONE と呼び、ネームサーバはこれらの AUTHORITY である。

resolver はユーザの要求に対し、ネームサーバから情報を引き出すプログラムである。ネームサーバにアクセスし、その情報を用いて直接キュエリーに答えるか、他のネームサーバにそのキュエリーを送る。ユーザプログラムに直接アクセスできるシステムルーチンなので、プロトコルは必要でない。

ユーザから見れば、ドメインシステムは resolver へのプロシージャーを通してアクセスされる。resolver からは、ドメインシステムは多数のネームサーバで構成される。ネームサーバからは、ドメインシステムは ZONE と呼ばれるローカルな情報の集まりで、構成されている。ネームサーバは、いくつかの ZONE のローカルなコピーを持ち、定期的に更新される。また、ネームサーバは resolver からのキュエリーを処理する。ネームサーバとして同じマシン上の resolver は、データベースを共有し、またそれとは別にあとのキュエリー用に外部の情報もキャッシュしている。

4.1.4 データベースモデル

ドメインシステムの体系は、一般的なデータベースシステム内における問題点を解決し、ユーザの次のような要求を想定して設計されている。

- データベースの大きさは、ホストやユーザ数に比例して増え、他の情報もドメインシステムに加えられる。
- システムは、サブセットが動的に変化しても、それに対応できる。
- データベースの管理的境界は組織に付随し、また各組織はネームサーバを提供する。

- クライアントは、あるネームサーバへの参照を受け取る前に、その途中のネームサーバを確認できなくてはならない。

コピーは更新用のタイムアウトの値とともに、配られる。分配者 (distributor) がタイムアウトの値を設定し、受け取り手が更新を実行する。もしネットワークやホストがフェイルしたら、更新するまでは古い情報が使われる。

データをアクセスするのに、データグラムとバーチャルサーキットの二つのやり方がある。更新のオペレーションには、バーチャルサーキットの信頼性が必要ではあるが、キュエリ・リスポンスは両方とも使える。なぜなら、両方とも同じメッセージフォーマットが使われるからである。

ドメインシステムでは、各ホストにばらまかれているマスターファイル内に、全データの元がある。これらのマスターファイルは、ローカルなシステム管理者に変更され、ローカルなネームサーバによって読まれる。これらのファイルは、FTP や mail などのメカニズムでホスト間を移動する。これらの機能にはネームサーバは必要なく、組織がマスターファイルをローカルに保持し、外のホストへ転送する。

ドメインシステムは、データのアクセス・他のネームサーバの参照・データのキャッシュやデータの更新のためのプロシージャーを定義する。

＜システム管理者の提供＞

- ZONE の境界の定義
- データのマスターファイル
- マスターファイルの更新
- 更新の方針

＜ドメインシステムの提供＞

- データ資源のフォーマット
- データベースキュエリーの方法
- 外のネームサーバからの、ローカルデータを更新する方法

4.2　名前空間の仕様

ドメインの名前空間はツリー構造をなし (Figure 4.1)、各ノードやリーフはラベルを持つ。ノードやリーフのドメイン名は、ルートからリーフへ

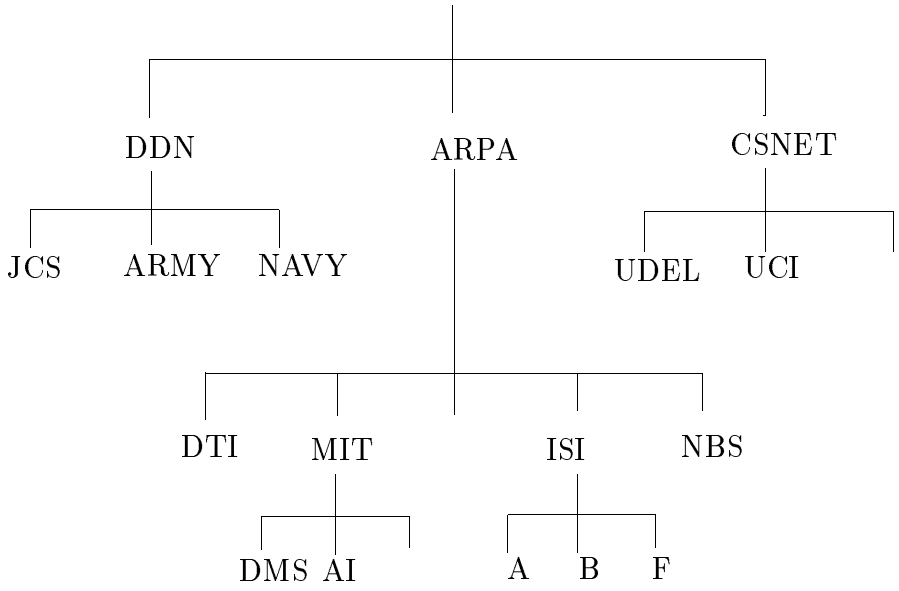


図 4.1: ドメインの名前空間

の各ラベルのパスとなっている。各ラベルは “.” で区切られ、ラベルオクテットとラベルの長さを表すオクテットの合計を２５５までとしているので、ドメイン名は２５４文字までとなっている。“*” は一つのラベルに対応する。これはプロトコル間の境界に、デフォルト的なレコードを存在させるために使われる。

このツリー構造 (Figure 4.1) は、管理組織の並列的な活動を、意図している。各ノードやリーフは、新しいサブドメインを作ることができる。もし我々が、このようなドメイン空間の AI の下に ANPAN と PURIN を加えようとするなら、AIがそれらの管理的エンティティとなる。

4.2.1 資源群の情報

ある名前に関係する資源情報の集まりは、それぞれの資源のレコード (Figure 4.2) で構成されている。

各資源レコードは、次のような要素を持つ。

- 資源群にはドメイン名があり、それが情報の “owner” となっている。ネームサーバはこの資源レコードをデータベースとして保持し、メッセージ内で転送される。

F.ISI.ARPA	A	IN	10.2.0.52
F.ISI.ARPA	A	CS	213-822-2112
F.ISI.ARPA	MD	IN	F.ISI.ARPA
F.ISI.ARPA	MF	IN	B.ISI.ARPA

図 4.2: 資源のレコード

- 資源型のフィールドには、ホストアドレスやメイル管理ホスト等、資源の型を指定する。
- クラスフィールドは、資源データのフォーマットを規定する。例えば、ARPA In ternet format(IN),Computer Science Network format(CSNET), アドレスデータ等がある。

＜資源タイプのフィールド＞

- A - ドメイン名に関連したホストアドレス
- MF - ドメインのメイルフォワーダー
- MD - ドメインのメイルディストリビュータ
- NS - ドメインの管理的ネームサーバ
- SOA - 管理的 ZONE の始まり
- CNAME - エイリアスされたものの正式名

＜クラスのフィールド＞

- IN - ARPA Internet system
- CS - CSNET system

SOA の “owner” フィールドのドメイン名は ZONE の始まりを意味し、NS は管理者が責任を持つ名前空間のポイントを決める。(Figure 4.3)

<owner>	SOA	<class>	<domain name>

<owner>	NS	<class>	<domain name>

図 4.3: 資源のレコード（SOA と NS）

4.2.2 キュエリー

ネームサーバへのキュエリーは、目的のドメイン名 (QNAME)・キュエリータイプ (QTYPE)・キュエリークラス (QCLASS) を含んでいる。

これらを受け取ったネームサーバは、一致するレコードを探す。もし一致する情報はないが、それを知っているネームサーバを知っている場合、ネームサーバは一致するレコードのドメイン名と、それをアドレスにバインドするレコードを返す。

4.2.3 ネームサーバ

一般的にネームサーバはドメインの一部を管理し、その区域を ZONE という (Figure 4.4)。ネームサーバは、ZONE を持たない場合もあれば複数の ZONE を持つ場合もある。もし ZONE が複数のネームサーバ内で重なっていたら、それはデータベース内の欠陥としてみなされる。もしキュエリーが他のネームサーバのものである時、求められる情報を返す時もあれば、他のネームサーバへのポインタを返す時もある。ネームサーバが、同じドメインにあるホスト内に存在しなくてもよい。

より複雑なネームサーバは、他のドメインからのキャッシュを用いる。たいてい re solver が他のネームサーバを尋ねた時、得られた情報をキャッシュする。より複雑なホストでは、resolver とネームサーバが協力して、データベースを共有する。キャッシュのために、time-to-live(TTL) フィールドが必要になる。また機能の単純化のために、自分が管理しているものと長期間一定のレコードだけ持つ。また、必要がなくなったら捨てるようにする。

4.2.4 ドメインの権威と管理

- ドメインを構成する管理的エンティティは、ネームサーバを提供し親ネームサーバにその情報を持たせる。管理者としてのネームサーバや

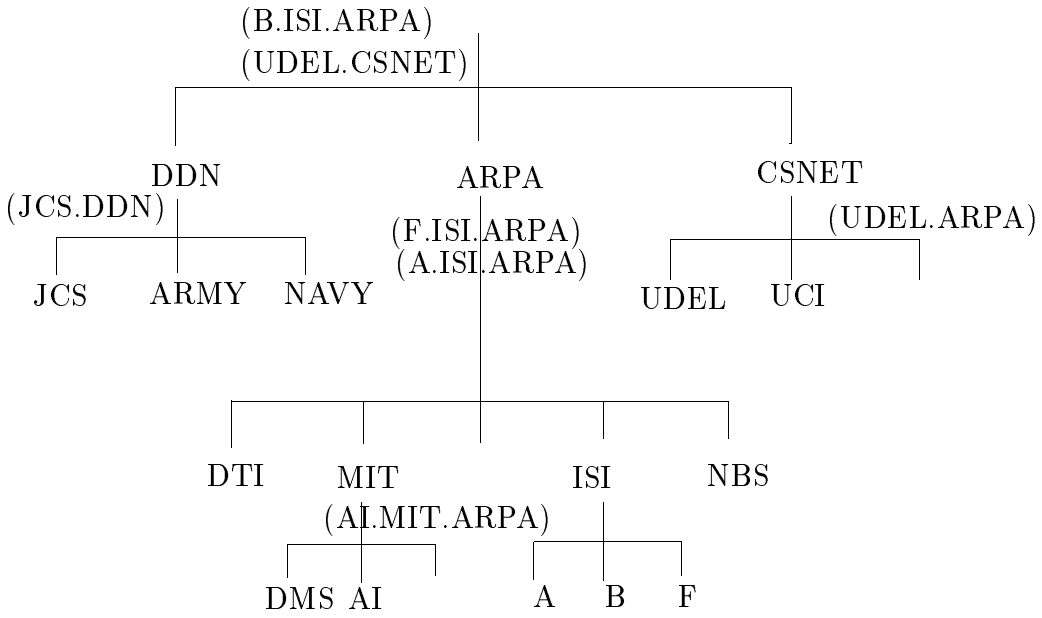


図 4.4: ドメインの名前空間とネームサーバ

ルートには、次のような基準が使われる。

1. 親ドメインの管理者に登録する。
2. 責任者の確認
3. 余分なネームサーバの提供

- ドメイン名は重ならないように管理者に登録され、他のドメインからその存在を認めてもらう。
- 各ドメインには責任者がいて、ドメインを確認したり変更したり、ホストの問題を解決したりする。
- 一度認められると、他のドメインの管理者と通信して、新しいネームサーバの NS レコードをそれらのデータベースに加えてもらう。

4.2.5 付加的情報

resolver がユーザプログラムに情報を返す時、それが次のキュエリーを生む。このような二重の通信を避けるために、ネームサーバのリスポンスには付加的情報をつけ、ネームサーバが知っている限りの情報を返す。

一つのメイルボックスが、複数の名前を持つこともある。その時にはドメインシステムは、CNAME レコードを使う。ネームサーバがいくつかの資源群内に望みのレコードを見つけられない時には、CNAME レコードを含んでいないか探す。もしもあったら、ネームサーバはリスポンス内に CNAME レコードを含め、そのデータフィールドのドメイン名にキュエリーをつづける。

4.2.6 ワイルドカード

管理者は、ドメインの一部または全部の資源情報のデフォルトを、取り扱うような場合もある。例えば CSNET ドメインの管理者は、ドメイン内のすべての IN class のホストにメイルを送りたいとする。このとき “*” が提供される。

“*” の意味は、QNAME とデータベース内のレコードとで異なっている。QNAME 内の “*” は、資源のレコード内の “*” にのみ一致する。その他の場合は、そのラベルにおけるデフォルトを意味する。

4.2.7 ネームサーバへの要求

1. その親ドメインのネームサーバに認識されている。

図 4.5: resolver の機能

2. 管理しているドメインについては、完全な情報を持つ。
3. マスタファイルやそれを持つネームサーバによって、情報を定期的に更新する。
4. もし情報をキャッシュするなら、TTL の管理も行なう。
5. 簡単なキュエリーにも答える。

4.2.8　完成サービス

ドメインシステムにおいては、ネームサーバは次のような完成のサービスを提供している。

resolver は目的のネームサーバに、QNAME は文字の一部を、QTYPE と QCLASS は望みのものを設定したキュエリーを送る。このリクエストは、目的のドメインのレコードも含んでいる。そしてネームサーバは、このレコードによって資源のありかを認識する。例えば、QTYPE=A,QNAME=B,QCLASS=IN と ISI.ARPA のレコードを含むリクエストは、ISI.ARPA のドメイン内の B で始まる資源を意味する。まず初めに、このようなリクエストをネームサーバが受け取ると、QTYPE が A で QCLASS が IN、QNAME が B の全レコードを探してくる。もし複数見つかった場合には、ドメイン名が目的のものともっとも一致していて、しかも短いものを選ぶ。それでも複数あれば、それらすべてを返す。

もし適当なレコードがなかったら、ネームサーバは QNAME 内の右側に文字を追加する。この場合、BB.ISI.ARPA にも一致するようになる。

4.3　resolver の機能

なぜ resolver とネームサーバを分けたかというと、resolver は基本的に機能している時間が可変であるのに対し、ネームサーバはデータグラムを

使うこともあって、ネットワーク内の遅滞とサービス時間とを合わせた一定のリスポンスタイムを持つようにできるからである。

resolver の目的 (Figure 4.5)

- 要求時に複数のネームサーバとつながる。
- あとのキュエリー用に、レコードをキャッシュしておく。

4.3.1 resolver の機能

- クライアントの管理的ネームサーバはどれか、そしてどのようにアクセスするかを知っていなくてはならない。
- クライアントのプロトコルに合わせて、キュエリーの QCLASS フィールドを埋める。複数のプロトコルを持つクライアントに対しては、クライアント側に選択させる。
- resolver は、どのキュエリーをもサポートできるよう、できるだけ多くのネームサーバと連絡できる。この時ループしないように情報が不足しているものは捨て、また同じネームサーバへの二度のキュエリーは避ける。
- もしネームサーバが答えなかったら、代理となるようなネームサーバに尋ねる。
- もしキュエリーがデータグラムを用いていたら、損失からの回復や複製ができる。

4.3.2 キャッシュ管理機能

キャッシュ機能は、ネームサーバの効率を上げることはできるが、結果が正しくない場合も多い。これらは、管理しているネームサーバによって更新されるか、またはタイムアウトする。

ネームサーバがレコードを返す場合、TTL のフィールドを持っている。そして res olver も TTL フィールドを一緒にキャッシュし、一定時間過ぎるとそのレコードは捨てられる。またネームサーバとデータベースを共有している場合は、時間が過ぎるとただ単に削除されるのではなく、TTL フィールドの値が定期的に減らされる。

第 5 章

WIDE の名前空間

5.1 名前をつけるべき、資源

ここでは、名前をつける対象一つ一つについて、細かく見ていく。とくに、**WIDE** のネットワークに、定義域を限定して、考えることにする。

もし机の上に、いろいろなものがてんでにばらばらに散らばっていたら、どのように見えるであろうか。たぶん、どのような物が、どこにあるのかは、ちょっと見ただけではわからないであろう。そしていざ、そこにある物のうちの一つを使おうとした時には、苦労して探さなくてはならないであろう。そこで、それらを整理整噸し、一目でありかがわかるようにならべておけば、そのうちのどれかが必要になったとき、すぐに手に入れることができる。

では、散らばったものを、どのように分類し秩序立てて、並べていけば良いのだろうか。まず第一に、分類・整理する時には、利用者にとっての便利さ・必要さが大切である。物は、それによって管理・支配される。

ものを分類する時に、どのようなことに気をつけたらよいであろうか。

1. 似た性質の物は、近くに集める。
2. 利用者が、すぐにその物のある場所を知って、利用できる。
3. 分類した後は、整った構成になっている。
4. 何か新しいものが入ってきた時に、同じ性質の仲間のところに入れられる。

5.2 WIDE の資源

既存の OS である、MS-DOS,VMS,UNIX において、それぞれ結果は同じものである場合でも、それぞれ資源の指定の仕方が違う。今までの各 OS がそれぞれ閉じた環境を形成していた時代においては、別に問題はなかった。

しかし **WIDE** の状況を考えた時、それらの環境は一転して机の上の散らばった品々になってしまうのである。ここではそのような状況で、各個人が効率的かつ容易に **WIDE** ネットワーク情の資源を利用するためには、どのようなことを決定し、どのような体系を考えていくべきかについて述べていく。

1. まず **WIDE** の資源とは何かをはっきりさせ、その上で各資源をどのような OS でもその概念が使えるよう、一般的かつ普遍的に定義し分類していく。

2. ユーザインターフェイスとしてのオペレーションの仕方をきちんと定義し、ユーザの要求を適切に処置できるようにする。そのためには、理想となるシェルコマンドのメカニズムを考える。

3. 名前ずけされる実体の性質を考慮して、“**WIDE** 資源の名前体系” を作り、名前をどのようにつけるかを決める。

5.3 名前を付けたい資源

・人

人は送られてきたメイルを受け取り、talk や phone をかけたりかけられたりする。ユーザ情報を提供する。このユーザ情報とは、

```
% finger
```

で得られるような情報で、住所・電話番号・本名等である。コンピュータの世界で計算機上に静的に人を表そうとするならば、このような (属性、属性値) のセットの表であると考えられる。例えば計算機上で斎藤先生の像を作るとすれば、それは passwd の ns の一エントリとなる。この一行内の項目 (属性と属性値のペア) をもっと増やそうとするならば、計算機上の斎藤先生は平面から立体的にもなりうる。しかし、各個人個人がユーザ情報を増やせば増やすほど、メイルがきちんと届くようになる、詳しい情報を得られるなどの利点はあるが、反対にデータベースが大きくなってしまい、コストが掛かりパフォーマンスが悪くなるなどの問題も出てくる。そのようなことを避けるために “人” は計算機上で時と場合に応じて、いろいろな形となる。例えばメイルを送るような場合、引数はログイン名であるが、実際はメールボックスにメイルがしまわれる。また phone,talk の場合、その人が現在ログインしている端末の出力装置、スクリーンとなる。そこで人は、要求に応じた必要な側面を見せる。

・ホスト

ランニング OS である。今までの世界つまり non-network な世界においては、ホストに関連するコマンドとしては wall,uptime,shutdown,who,ps,kill,sh,login 等がある。特に kill などはホスト上で走っている複数のプロセスに対するものである。さらに network-wide な世界では、telnet,ftp,rsh,rcp,rlogin 等既存のコマンドを利用し、さらに発展させた形で今の世界と連続している。

5.4　ネームド オブジェクトの定義

次に、これらの **WIDE** の資源に関して、一つ一つ定義していく。

5.4.1 各資源の定義

人 ユーザ。コンピュータシステムのサービスを、受ける人々の呼び名。
ネットワーク上の資源（人を除く）、または、その集合（計算機システム）を、利用し、それを使って仕事をする人。

host ネットワーク上の１ノードで、CPU を持ち、名前を持ち、ユーザが login できる計算機。

server ネットワーク上で、client の要求に対し、何らかのサービス
を行なうプロセス。

printer ネットワーク上で、データ（文字や図形）を紙上に印刷する出力装置。

file レコードの組織的な集まり。
定められた規則にしたがって、論理的な意味を持つデータの集合。
頭から連続的に、順番にしかデータを読み書きできないものや、データが並んでいる順番に関係なく、どこからでも読み書きできるものがある。普通、 ディスク上やテープ上に設計され、作られる。

directory 辞書。キーがあり、それに元ずいて情報を検索する。互いに無関係に思われる散らばった実体を、ある共通の側面を元に一つにまとめてバインディングする。

archiver file を長期保存するもの。３次記憶。
バックアップに用いる。
ユーザは。必要とするデータファイルをすべて、コンピュータとオンラインで接続された外部記憶装置に保存しておくと、必要な時にそれが利用できて、便利である。しかし、外部装置には制限があり、すべてをオンラインにしておくことはできないので、データファイルの一部を磁気テープや CD などに移しておく。そして、要求があった時にオンラインに移動する。

device host の周辺機器。

database 情報検索に用いられる file。
たくさんのデータを、合理的に無駄のないようにまとめ、データを構造化して各データの検索や更新などを、効率よく行なえるようにしてある。

5.4.2　資源の集まり

domain　管理的、地域的境界で区切られる、host、人の集まり。

組織　管理的境界で区切られる、host、人の集まり。

service　server の集まり、又は、server の行なった結果。

network　電話回線、ethernet など、何らかの形で接続された通信網のこと。
その通信網が、コンピュータを利用する、またはコンピュータ同志が、互いに通信することを目的としている場合、それをネットワークという。
host や他の計算機の集まりで、また、その集まりも含む。
地域的（国？）境界で区切られた集まりの単位で名前を持つ。

group　資源の集まり、又は、その集まり。例えばユーザの集まり (メイルやアクセス権に利用される。) ホストの機種による集合、ファイルやデバイス、ドメインをいくつかまとめて group とも呼べる。実体は、単純に名前のリストであるといえる。

5.4.3　資源の型の定義

各資源の **attribute** は次の様に定義できる。

人 ネットワーク上の計算機システムを利用する実体。
ユーザ。
あるマシン上に、メイルボックスやアカウントを持っている。

service ユーザやファイルに関して何か仕事をし、その結果それらにより良い環境を与える。

host CPU を持つ。ユーザが直接ネットワークにかかわり合いを持つ玄関口。
ネットワークにつながっていないと host ではない。

domain 管理的・地域的境界で区切られる、ホストや人の集まり。
普通は、階層構造をなしている。そしてアドレスの各ノードとなっている。

組織 外から見て、管理母体が一つ。
社会的に認められている集まり。

network host,printer など、CPU を持つ計算機を物理的に接続した範囲全体。
ドメインの管理的な集まり。

group 各資源型の集合。

5.4.4 名前を付けたい資源の型分け

1. サービス

主にコマンドによって、サービスを提供する。サーバに対し要求を出してくる側をクライアントという。サーバはクライアントの要求に対し、必要な情報を提供する (ネームサーバ:アクセスしたいホストの情報やルーティング)。または、要求に直接答える (`jserver` やデーモンなど)。この時に、サービスを行なうプロセスはホスト上で動いているが、ホストを意識させないサービスを提供することが望まれる。つまり、クライアントは自分の望む実体やサービスのある場所を考慮することなく、アクセス・利用できることが必要である。またクライアントに対し、サーバはいくつかの候補をあげ、その中から選ぶようにしても良い。そこで、サーバはクライアントの要求に答えるという意味で、サービス型とした。次にファイルを紙面上に出力する場合を考える。ユーザは、

```
% lpr -Pimagen .cshrc
```

のように端末にコマンドを入力すると、lpr コマンドによりファイルはプリンタ待ち行列に入る。そして自分の番が来たら、スプールファイルから取り出される。これは、lpd というプリンタデーモンが管理している。次に printcap で指定されているフィルターコマンド (if,of,tf) を通る。troff 等のようにフォントイメージを必要とするものは、この時読み込まれる。そして実際には、ホストにつながれたプリンタのうち imagen にファイルが出力される。ここでわかるように、

```
% lpr -Pimagen
```

と入力することによって、ユーザは “プリンタからファイルを出力する” というサービスを受けることができる。そしてこのサービスは、lpd デーモン、課金やビットイメージ変換などのフィルタ、/dev/lp のプリンタデバイス等、いろいろなものが関係している。そこでこれらを一つのプリンタサービスと見ることができるので、途中のサーバ、プリンタ、デバイス、データファイル等全てをサービス型とした。また、ディレクトリはそれ以下のファイルの情報をユーザに提供する、アーカイバはユーザの要求におおじて、ファイルを保存したりバックアップ、ダンプを行なうなどの点から、サービス型とした。

2. ドメイン

ドメイン、組織、ネットワークも互いに密接にかかわっている。ただ単位ホストを通信回線でつないだ、物理的なホストの集合を想定する。ここでは、ホスト間の何の関係も管理も性質も要求されていない。しかし実社会においては、コンピュータを利用するユーザは組織を形成している。そしてホストは、その組織にもとずいて、連結されている。例えば、慶応大学・NTT・電総研などがあげられ、実社会の管理・統括された人の集まりであるといえる。そこで、コンピュータネットワークの世界に反映させるため、“ドメイン” という概念が導入された。このドメインとは、ネットワークにおいて閉じた各組織とその関係の表現であり、今のところは物理的なつながりにも影響を受けている。そこで **WIDE** 上の資源を考えた時に、ドメインを型の一つとしてあげた。ホストはこのドメインの一つ一つとして考えられる。例えば、慶応ないのマシン koch を考えた時、“/” で始まるその下の UNIX ファイル構造全てが階層構造をなし、一つのドメインを形成している。また組織も、keio,ntt,etl などのようにドメインの階層構

造の一部を形成している。ネットワークも同様にドメイン型とした。なぜなら、ネットワークとはあいまいなものであるからである。例えば、BITNET のように ARPA ネットや電話回線を使って、ただつなげただけの物もある。また、もし慶應大学のあるマシンが BITNET ともつながっていたら、koch はネットワークとして、BITNET と J UNET の両方に属することになる。しかし、完全に無くしてしまうわけではない。各資源が、どのネットワークに属しているかを示す側面も、不可欠である。そこでこれからの名前体系に、ネットワークを表現しようとするなら、次のようにネットワーク管理を意味するドメインをつくって、その下にネットワーク管理組織をおけばいい。

3. ユーザ

4. グループ

5.5 ネームドオブジェクトの表記方法

以上で、机の上に散らばった品々を、それぞれが持つ特徴によって分類し、整理整頓することができた。今度は、それらに対して" 必要なものをとってくる" という作業をしなくてはならない。たとえば、取ろうとするものが遠くにあったら、誰かに探して取ってきてもらわなくてはならないし、近くだったらすぐ手を伸ばせば良い。また、誰か他の人のものであれば、その持ち主にことわらなくてはならない。そして、物によってはそーっと、箱に入れて運ぶかもしれないし、転がしながら持ってくる物もある。

5.5.1 既存の表記方法

研究室にはいって端末の前に座り、まず login する。

```
% potato login:yuko

passwd:
```

はじめに、用事のある人たちにメイルを出す。

```
% mail tomo@kogwy.keio.junet

% mail jun@ccut.cc.u-tokyo.junet
```

つぎに、koch に login する。

```
% rlogin koch

% cd report
```

そして、potato からファイルをコピーする。

```
% rsh potato more ~yuko/report/sotsu

% rcp potato:~yuko/report/sotsu .
```

ここまでは、誰もが毎日何度となくやっている動作なのであるが、よく見てみるとコマンドの書き方が様々である。mail に関してだったらユーザを引数にしているし、more はファイル名、cd はディレクトリ名、そして rlogin はマシン名をとっている。また同一のコマンドでありながら、引数の形式をいろいろとれる場合がある。例えば、

```
% rcp yuko@kogwy:remote local

% rcp kogwy.yuko:remote local

% rlogin kogwy -l yuko

% talk yuko@kogwy

% talk yuko.kogwy

% talk kogwy!yuko

% mail yuko@kogwy.keio.junet

% mail yuko@keio.junet

% mail yuko
```

このように、コマンドによって引数の形式が異なり、そしてさらに同じコマンドでも、複数の種類の引数が可能となっている。そこで、

1. コマンドにおける名前構造が、統一されていない。
2. ログインネームでしか、ユーザを識別できない。

が、問題点としてあげられる。

5.5.2 既存の名前構造

現在名前構造は、一般的に次のようになっている。

＜名前空間＞::=＜トップドメイン＞ “.” ＜サブドメイン＞
＜トップドメイン＞::=“jp”
＜サブドメイン＞::=＜ドメイン＞ “.” ＜サブオブジェクト＞ | ＜ドメイン＞ “.” ＜サブドメイン＞
＜ドメイン＞::= “ac”|“co”|“keio”|“u-tokyo”|“cc”|“cs”|“scitech”|“math”|“junet”|“bitnet”
＜サブオブジェクト＞::=＜オブジェクト＞ | ＜オブジェクト＞ “.” ＜サブオブジェクト＞
＜オブジェクト＞::=＜ user ＞ | ＜ file ＞ | ＜ database ＞ | ＜ host ＞ | ＜ printer ＞
| ＜ archiver ＞ | ＜ server ＞ | ＜ device ＞

{jp,domain}
{ac,domain}
{keio,domain}
{cc,domain}
{junet,domain}
{yuko,user}
{koch,domain}
{egg.c,service}
{imagen,service}
{jserver,service}

5.5.3 実装案

（例１）

＜名前空間＞::=＜トップドメイン＞ “.” ＜サブドメイン＞
＜トップドメイン＞::=“jp”
＜サブドメイン＞::=＜ドメイン＞ “.” ＜型別オブジェクト＞ | ＜サブドメイン＞
“.” ＜ドメイン＞ “.” ＜型別オブジェクト＞
＜ドメイン＞::= “ac”|“co”|“keio”|“u-tokyo”|“ cc”|“cs”|“scitech”|“math”|“junet”|“bitnet”
＜型別オブジェクト＞::=“service”“.” ＜サービス型オブジェクト＞ |“user”“.” ＜ユーザ型オブジェクト＞
|“domain”“.” ＜ドメイン型オブジェクト＞
＜サービス型オブジェクト＞::=＜ server ＞ | ＜ printer ＞ | ＜ file ＞ | ＜ device ＞ | ＜ archiver ＞
| ＜ directory ＞
＜ユーザ型オブジェクト＞::=＜ user ＞
＜ドメイン型オブジェクト＞::=＜ host ＞ | ＜ network ＞

（例２）

＜名前空間＞::=＜トップドメイン＞ “.” ＜サブドメイン＞
＜トップドメイン＞::=“jp”
＜サブドメイン＞::=＜ドメイン＞ “.” ＜サブオブジェクト＞ | ＜ドメイン＞ “.” ＜サブドメイン＞
＜ドメイン＞::= “ac”|“co”|“keio”|“u-tokyo”|“ cc”|“cs”|“scitech”|“math”|“junet”|“bitnet”
＜サブオブジェクト＞::=＜オブジェクト＞ | ＜オブジェクト＞ “.” ＜サブオブジェクト＞
＜オブジェクト＞::=＜ user ＞ | ＜ file ＞ | ＜ database ＞ | ＜ host ＞ | ＜ printer ＞
| ＜ archiver ＞ | ＜ server ＞ | ＜ device ＞

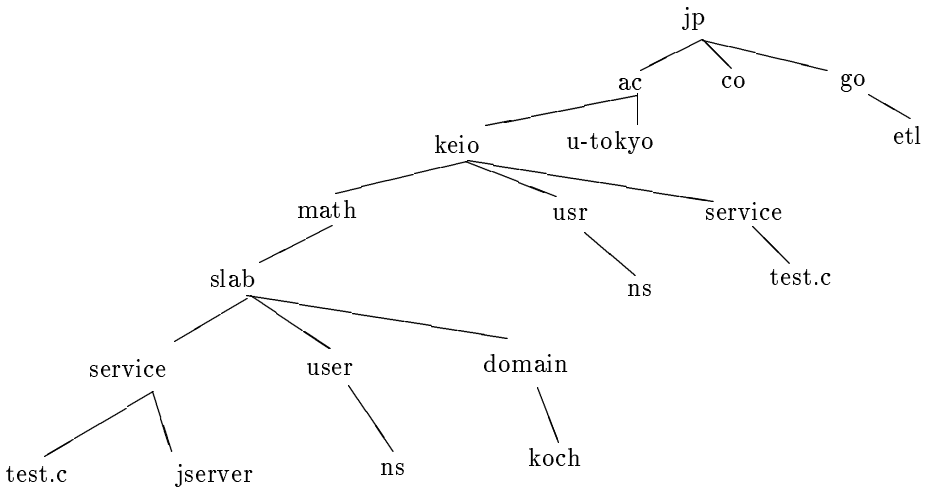

jp
ac
co
go
keio
u-tokyo
etl
math
usr
service
slab
ns
test.c
service
user
domain
test.c
jserver
ns
koch

図 5.1: 例 1

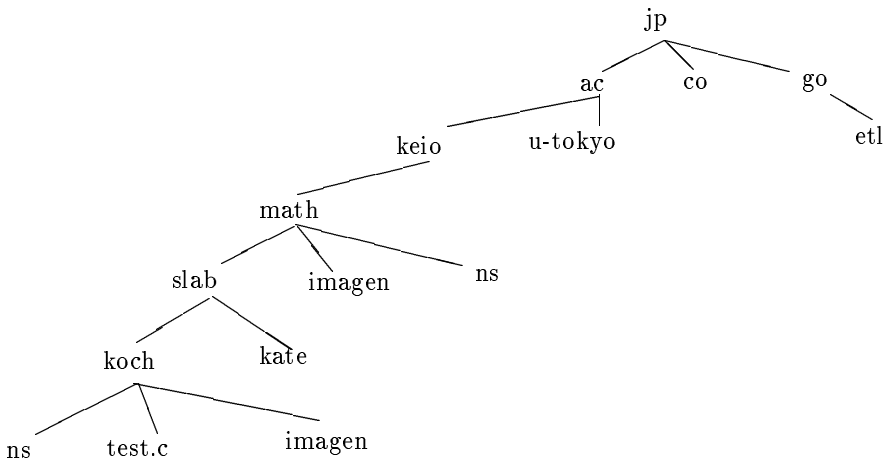

jp
ac
co
go
keio
u-tokyo
etl
math
slab
imagen
ns
koch
kate
ns
test.c
imagen

図 5.2: 例 2

5.5.4　実装例の比較

次に、名前空間を決める時に、考えること

1. オブジェクトが移動しても、構造は変わらない
2. ユーザのいる場所を意識させない
3. セマンティックスな意味のステップ、属性が多数行出てこない。
4. オブジェクトの変更、削除、追加が簡単
5. コマンドとの関わり
6. 名前の重なり

	例 1	例 2
1	ホストが移動しても、同じドメイン内であればそれほど名前構造に変化がない。(ただし一番下のドメインに付いているもののみ)	ホストが移動すると、その下のオブジェクトも一緒に移動する。
2	場所はそれほど意識しない。(ただし一番下のドメインに付いているもののみ)	オブジェクトがどのホストの下にあるのか、知る必要がある。
3	型別のディレクトリが増え、BIND サーバの構造が複雑になった。	数としては、階層数は同じであるが、UNIX ファイル構造に近いので、楽に抵抗なく取り入れられる。
5	コマンドもこの三つに分類できるので、その意味では都合がよい。	現在使っている引数の指定の仕方とあまり変わらないので、ユーザにとっては使いやすい。
6	ユーザ、サービス、ホストで同じ名前を持つことができる。	ユーザ、サービス、ホストで、同じアトムドメイン内では、重なれない。

ここで重要となるポイントは、UNIX 上で実装するにあたって UNIX のよい点を生かすということである。それは、シンプルさをモットーにすることである。名前空間を、上のように BNF を使って書き表した場合、複雑さは次のような点で決められる。

1. 行が多い。
2. セマンティックスなステップが多く入ってくる。
3. 末端以外の場所にも属性が入ってくる。

となる。そういう点から見て、“service”“user”“domain” などの属性別の階層構造を分けるよりも、既存の UNIX の階層構造に依存した（例２）の形式を、今回は採用した。そこで、本当にこちらの名前体系を実際に UNIX 上にインプリメントし、コマンドを用いて研究室やその他のグループの方々に使って頂くことによって、便利さ・効率の良さを実証することにした。

5.6 実装

5.6.1 引数の指定方法

まずコマンドの引数が、何を対象としているかによって、主な既存のコマンドを分類してみる。（ここでは、サービスは jserver,lpr など、コマンド自体となっているので除外される。）

引数	コマンド
ユーザ	finger,talk,phone,mail
ファイル	rcp,cp,mv,cat,more,file,chmod
ディレクトリ	ls,chmod,mkdir
ホスト	rlogin,rsh

ここでもう一度、前章の “資源の定義” に戻って考えてみる。そこでは、ユーザを “ネットワーク上の計算機システムを利用する人”、ホストを “ユーザが直接ネットワークと関わりをもつ玄関口” と定義した。また、ファイルやディレクトリに関しても、同じことが言える。 **WIDE** というネットワークにおける個々のユーザやファイルを考えた時に、ホストを境界としてローカルな世界と、ネットワークワイドの世界に分けることができる。ローカルな世界は、各ホストごとで /(root) を基点として UNIX のファイルシステムを形成している。現状としては、

~yuko:yuko のホームディレクトリ

.:カレントワーキングディレクトリ

..:カレントワーキングディレクトリの親ディレクトリ

などの相対的な表現以外は、/user/yuko/report/sotsuのように、/(root)から “
” でつなげていく絶対表現を用いている。これは、UNIX の機能上このまま用いることにする。次にネットワークワイドの世界では、メイルを例にとって考える。メイルに関しては、各ホストの mailer がメイルを sendmail に渡し、sendmail はメイルの送り先を見て、ローカルなもの・SMTP で送れるもの・UUCP でしかおくれないものとに分ける。基本的には送り先をみんなが知っていれば自分で処理し、もし知らなければ、そのドメインを代表するホストへ送るという動作をする。そのために、keio 内のマシンである koch からメイルを出す場合、同じ koch であれば、

```
% mail user
```

もし、同じドメインの他のマシン上のユーザであれば、

```
% mail user@machine
```

また、外のドメイン内のホスト上のユーザへであれば、

```
% mail user@machine.domain1.domain2.domain3.junet
```

そこでホストに関しては、この形式を用いて相手のホストと自分のホストとの、共通ドメインの一つ前まで書くようにする。 以上の二つを “:” で区切ってつなぎ合わせて、コマンドの引数とした。例えば、

```
% rcp ccut.cc.u-tokyo:/usr/xroads/yuko/report/sotsu .

% rcp ccut.cc.u-tokyo:~yuko/report/sotsu .

% finger yuko@kate

% rlogin kogwy.cs
```

のように指定する。このようにすれば、既存のコマンド引数の形式がそのまま使え、しかもそれをネットワークの世界に応用することができた。またどんなに多くのドメインやホストがそのままネットワークに加わっても、同じ形式を使うことができる。

5.7　wphone について

名前サーバを使用する応用システムとして **WIDE** 版 phone である wphone を UNIX 4.3BSD 上で試作した。

phone は UNIX システムで使用されている、ユーザ間で会話をするためのアプリケーションであり、次のように使用される。

`phone` *user*@*host* [*tty*]

`phone` を使用すると、ネットワークを介して *host* に `login` している *user* と会話をすることができる.

現在の計算機システムではユーザの名前空間はホスト毎に独立している。そのために、ネットワーク環境でユーザを特定するためにはホストも指定する必要がある。しかし、**WIDE** 環境ではネットワーク全体を定義域とした名前空間を提供するので、ユーザの名前だけでネットワーク上の特定のユーザを示すことができる。

そこで `wphone` は次のように使用される。

`wphone` *username* [*hostname* [*tty*]]

ただし、実際に会話を行なうためには特定のホストの特定の端末を知る必要がある。

従来の `phone` の場合には *tty* が省略された場合はユーザが使用している端末を `phone` が見つける[1]。同様に `wphone` にもユーザが使用しているホスト/端末を見つける機能が必要になる。

5.7.1 wphone の動作

引数の username として、前章で定義した

- /top-domain/second-domain/.../domain/user

というフォーマットを適用する。wphone は、wgetobjbyname という関数を呼び出して、つぎのような目的のオブジェクトの

```
struct wobjent {
        struct sockaddr address;
        char            *object-type;
        char            *object-name;
};
```

という情報を得る。この address は、そのユーザのユーザサーバが存在するホストのアドレス・ホスト名・そして IPC の口（この場合は、そのユーザサーバと通信するためのポート番号）へのポインターとなる。object-type は、**WIDE** object を 3 つの型に分類した時の、“user”,“service”,“domain” にあたる。今回は、“phone” を考えているため、この object-type は “user” となる。object-name は wphone の引数をそのまま渡す。

[1] utmp を参照するようになっている。

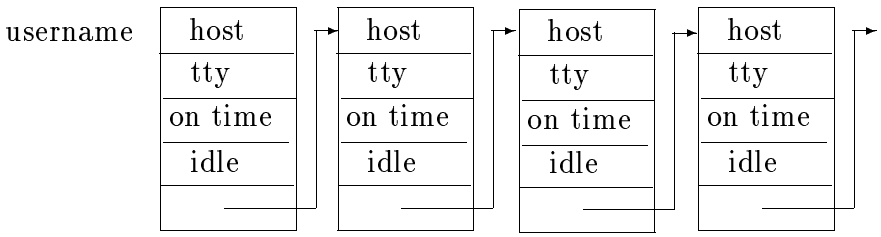


図 5.3: 各ユーザのログイン情報

次に、wconnect を呼んでこのユーザサーバにコネクションを張る。そして、そのユーザサーバからユーザの実名・ユーザがログインしている中で wphone をかける端末を一つ得る。そして、そこに phone をかける。

5.7.2　wuser の動作

wuser.infod の動作

“/usr/spool/rwho” には、各ホストごとにファイルがあり、そこに各ホストにログインしているユーザ・ユーザのログインしている端末・ログインした時刻の情報がはいっている。そこでその情報を利用して、各ユーザごとに次のような構造体を作る。

それを、wuser.phoned に渡す。

wuserd の動作

sdomain（ネームサーバ,7001）と sclient（7002）の両方に対して connection を張る。もし sdomain から要求が来たら、fork して fromdomain を実行する。もし sclient から要求が来たら、fork して fromclient を実行する。各ユーザごとにファイルがあり、そこには wuser.infod から受け取った情報が入っている。そこで fromdomain は、要求をネームサーバから受け取ると、そのユーザのユーザサーバのいるホストのアドレス・ユーザサーバの口（この場合はポート番号）・オブジェクトのタイプ・オブジェクト名を返す。fromclient の方は、まずユーザ名を受け取り、次にそのユーザに対して得たい情報のタイプを聞く。このタイプとして、name,realname,tty,logout, wphone がある。そうすると、そのユーザのログイン情報が格納されているファイルである user.tty や user をオープンして、それぞれのデータを返す。wphone から要求が来た場合には pipe をつくって fork し、親はその

ユーザの構造体を wuser.phoned に渡す。そして wuser.phoned から得た答をそのまま wphone に返す。

wuser.phoned

各ユーザのログイン情報を受け取ったら、それを構造体に入れていく。そして、各 tty のうち一番 idle time が小さい tty の情報を返す。

5.7.3 nameserver の動作

ネームサーバは、例えば “/jp/ac/keio/yuko” のような要求を受け取る。そうすると、自分のドメイン名とこの “/jp/ac/keio” とを比べて、同じだったら自分の所持しているデータベースファイルを探す。このデータベースファイルは、次のような形式を採る。

```
----------------------
#
# KEIO MATH SLAB NAME SERVER DATABASE
#

# My domain name

/jp/ac/keio domain 131.113.1.1

# Parent domain name

/jp/ac domain 192.41.197.4

# user name

yuko user wuser
tomo user wuser
uno user wuser
jun user wuser
onoe user wuser
saga user wuser
----------------------
```

そこでこの “yuko” というもののタイプが user であることを知る。このユーザの情報を管理しているのは、wuser というサーバなのでそことコネ

クションを張る。そしてユーザ名を渡す。そして、wuserd からそのユーザの情報 (そのユーザサーバの口など) を受け取り、要求してきた wphone に返す。

もし受け取った資源が自分の管理しているものでなかったら、親ドメインのサーバにその資源名と要求者のアドレスを渡す。そして親ドメインのネームサーバは、同様な動作を繰り返す。

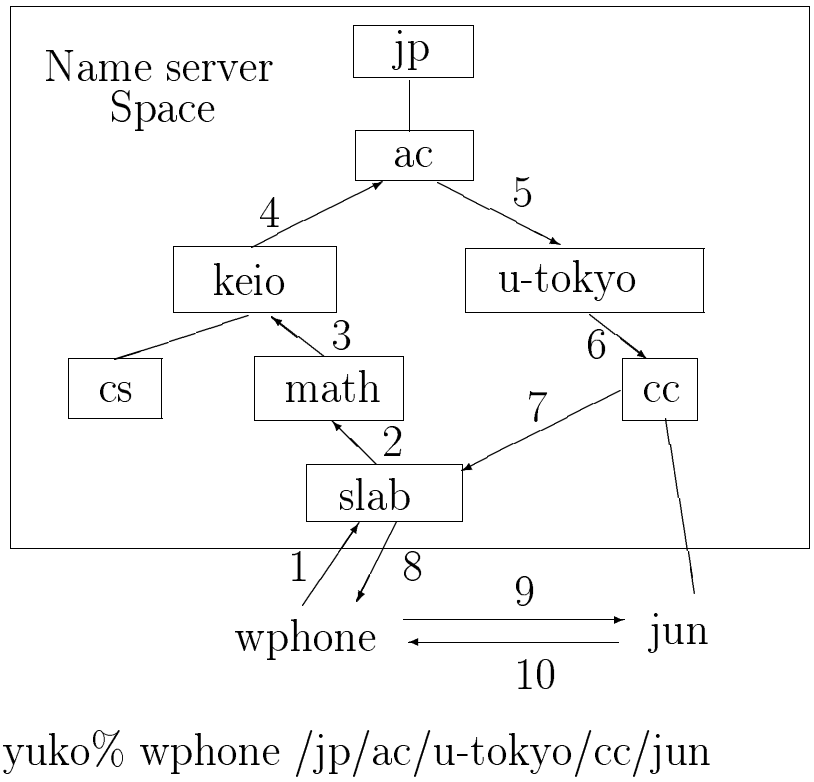


図 5.4: wphone の構造

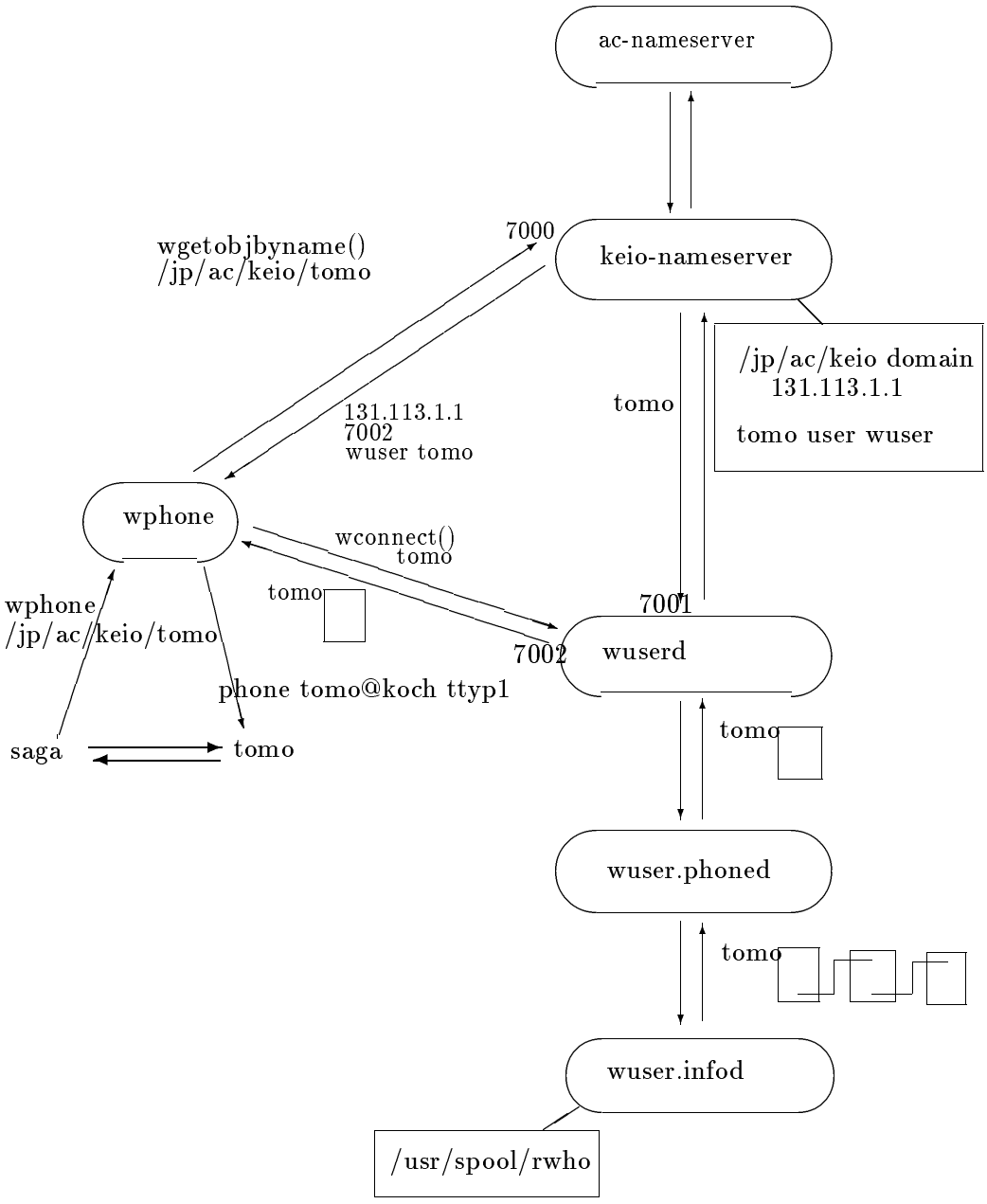


図 5.5: wphone の構造・その 2

第 6 章

結論

この論文では、今まで各ドメインで独立に構成していた名前空間を、**WIDE** 分散環境におけるグローバルな名前空間に統一するために、もう一度名前について考え直した。

まず名前とは何か、そして名前が付けられる資源とは、どのようなものかについて考えた。はじめに一般社会における名前を考え、コンピュータの世界、そしてネットワークへと定義域を広げていった。名前とは、ある実体を一意に識別するための” 識別子” と定義した。それらは、マシン側にわかりやすい識別子と、人間にとって扱いやすい識別子とに分けられる。そして名前には、実用的である、伝達性・識別性がある等の特徴をもつ。次に、資源とは名前が付けられる実体とした。とくに **WIDE** の資源とは、サービスをへの要求を受けるためにある ”口 ”を持っているもの定義した。そしてその口を通じて、ユーザはサービスを提供される。さらに、それらの資源をいくつかのタイプに分類した。タイプとして、特にユーザ・サービス・ドメインがあげられる。次に、そのオブジェクトを包含する、名前空間について考えた。これは、どのような組織や資源でも取り込むことができるような、グローバルな名前空間でなくてはならない。その構造を、タイプの概念を採り入れながら階層的に構成した。それは、次の理由による。

- 情報を各ドメインに分散して、管理できる。
- 各ドメインで一意性を保証すれば、自動的に全体の名前空間内で一意となる。
- ネットワークにおいて、拡張・再構成しやすい。

名前だけがわかっても、いざその実体にアクセスしようとすると、困難であることがわかる。そこで今度は、名前構造を考える必要が出てくる。ここでは、この名前構造を階層的に構想した。これは、上で定義した名前空間にならった。最終ドメインには、ドメイン・サービス・ユーザの３つ

の型のオブジェクトが現れるが、それ以外には、ドメイン型のオブジェクトが現れる。これは、UNIX ファイルシステムと同じ構造となる。この時、名前と実体とを結び付けるための手段が必要となってくる。それがアドレスである。名前に意味を持たせ実用的なものとするためには、名前とアドレスとを対応させるメカニズムがなくてはならない。アドレスと名前とは、同じものとして考えられがちである。しかし、これらは全く違うものである。アドレスとは、実体の居場所を物理的に示す。ただ単に情報を伝達していく上で認識されるような形式をとっている。そこでここでの名前表現は、名前構造をそのままトップドメインから表現した、それ自体は意味の持たない形式を採用した。UNIX のよい点を生かすといくことも重要なポイントとなる。

最後に、コマンドの引数としてこの名前表現を適用した。まず既存のコマンドの引数形式が不統一であることを、問題点としてあげた。そこで引数となる資源のタイプ別に、コマンドを分類した。そしてそれらに共通の名前表現の形式を設定した。その結果、今までばらばらだった引数の表現形式を、統一することができた。しかも、アプリケーションに依存しない名前表現となった。しかもそれをネットワークの世界に応用することができる。どんなに多くのドメインやホストがそのままネットワークに加わっても、同じ形式を用いることができる。そのため、分散環境上でユーザが効率良く資源にアクセスすることができるようになった。

第 7 章

今後の課題

今後の課題として、次のような点があげられる。

- 分散環境において資源にアクセスする場合の、資源の保護の問題。これから各組織がどんどんつながってくる時に、これは重要な問題となっている。アクセス権・ケーパビリティ等が考えられる。
- これらを **WIDE** の分散環境において、実際のアプリケーションとして実装していかなくてはならない。具体的には、ユーザ型のオブジェクトに対するオペレーションとして、ユーザの居場所を意識させないコマンドの指定があげられる。
- ユーザ型のオブジェクトの名前と、実体との対応を考えていく。複数のドメインがつながった時に、いくつかのドメインに包含されるユーザの問題を解決していかなくてはならない。