CORBAを用いたNetwork Semaphoreの実装の試み

高エネルギー加速器研究機構
加速器研究部
山本　昇
Noboru Yamamoto
High Energy Accelerato Research Organization
Accelerator Laboratory
1-1 Oho, Tsukuba
Ibaraki, Japan

DRAFT 平成13年7月6日

概 要

加速器制御システムはネットワーク分散型の計算機システムで構築されている。さまざまな制御プログラムがネットワークでお互いに接続された計算機の上で稼働している。このような分散環境下のプロセス間での同期をとるための一方法として、ネットワーク上にセマファを実現するためのプログラム (Network Semaphore) を開発した。実装にはCORBAを利用することにより、異機種間であっても容易にこのセマファを利用できる事ができた。また、CORBAのオブジェクト指向技術の採用によって、効率良くプログラムが記述できた。CORBAの実装としては、プログラム言語Python向けに開発されたfnorbを採用した。

1　始めに

近年の加速器制御システムはいわゆる“標準モデル”に従った、ネットワーク接続による分散型の制御システムが主流である。特にLANL/ANLが中心に開発されたEPICSが公開されたことによって、この枠組みを採用することで、制御システム開発の主力を個々の加速器に特有なアプリケーション開発に注ぐことが可能となった。このようなネットワーク分散型のシステムでもプロセス間の同期や、制御資源と排他的な利用のために機構が必要な場合がある。

例えば、KEKB制御システムでは加速器全体にわたる機器の同期のためにeventコード分配システムを使っている。全周のビーム起動測定の同期などが主な使用法である。全周の軌道測定のためには、分散配置されたビーム軌道測定装置を測定可能な状態にした後、Eventコードジェネレータに指令をだし、エベントコードを送出する。この一連の作業の間、ビーム軌道測定装置およびエベントコードをこのアプリケーションが占有する必要がある。特に、加速器の他のパラメータとの相関をとるなどの目的には、このような同期測定の仕組みが必要不可欠である。

残念ながら、KEKBで採用しているEPICSの枠組みではこのような機能は、現在の所、汎用の機能としては用意されていない。

一般に実時間制御を行う計算機上ではこのような目的のために各種のセマファが利用される[1]。　実時間処理OSでは用途に応じて各種のセマファ(バイナリ・セマファ、相互排除セマファ、計数セマファ等) が用意

[1]セマファ(semaphore) の元々の意味は鉄道などで使われた腕木信号器である。

されている。これらのセマファの内、バイナリ・セマファはすべてのセマファの基本である。通常セマファは OS のシステム機能として実装されており、同一計算機上のプロセス間の同期等には利用できるが、複数の計算機にまたがるプロセスの同期には直接応用することができない。

Unix ではセマファにかわり (時間の遅れが問題にならない場合には)、ファイルロックが資源の排他制御によく使われる。ファイルロックは一応 NFS 等のネットワークでファイルシステムを共有している計算機の間では分散プロセスの同期・排他制御の目的で利用することが可能である。反対にファイルシステムを共有しない計算機間では、これをりようすることはできない。またファイルロックのためのプログラム・インタフェースは機種・OS 毎に若干の違いがあり、異機種間での移植の問題もある。

さまざまな計算機が共存するネットワーク分散環境下で稼働するプロセス間の同期をはかるためのセマファ(ここでは便宜上ネットワークセマファと呼ぶ）を用意することでこのような問題に対処することができる。このような環境下でのアプリケーション開発には、通信規約からアプリケーションとのインタフェース規約まですべてを自主開発することも可能であるが、開発の効率化のためには既に存在する分散アプリケーション開発環境を利用することが有利である。CORBA[1] はネットワーク分散処理環境化における分散オブジェクトアプリケーション開発のための枠組みであって、これに準拠したアプリケーション間では、ネットワークを越えたオブジェクトの相互運用が、機種を問わず可能となる。CORBA は非営利の団体である OMG(Open Management Group) の定めた仕様であり、これに基づいてさまざまな CORBA 処理系が実装されている。商用の CORBA 処理系だけでなく、最近ではフリーな (一部は非商用の限定がつく）CORBA 処理系も入手可能 [2, 4] であり、これらの処理系を利用することで、分散オブジェクト環境を作りあげることが可能である。

Fnorb はこのような処理系の一つであり、プログラミング言語 Python への CORBA マッピングをもった CORBA 処理系である。汎用で強力かつクリーンなオブジェクト指向言語 Python を利用できることで、分散オブジェクト環境を短時間で構築することができる。

この小文では、Python/Fnorb を用いて、Network 分散環境下で利用可能なセマファ機構の作成について報告する。

2　使用するツールについて

今回の試作では、CORBA として fnorb, サーバを実装するプログラム言語として Python を採用した。 この組み合わせはプログラム言語 Python の開発の容易さに助けられ、アプリケーションの rapid prototyping に最適なものであった。

ここでは、簡単に fnorb および Python の紹介を行う。

2.1　Fnorb

Fnorb はオーストラリアの Queensland 大学と CRC for Distributed System Technology が共同で開発し、非営利での利用に対しては無料で配布している CORBA2.0 の処理系である。Fnorb はほとんどが Python で記述されており、OMG-IDL(Open Management Group Interface Definitiln Language) から Python へのマッピングを実装している。CORBA により他システムと連携するクライアントアプリケーションの開発はもちろんのこと、CORBA のサーバも Fnorb だけを用いて開発が可能である。

CORBA の規定するサービス (COS: Common Object Services?) のうち Interfece Repository サービスと Naming service の実装も合わせて配付されている。

Fnorb は CORBA 2.0 に準拠し、IIOP(GIOP), DSI, DII, BOA, 等を実装している。OMG ID L からプロ

グラム言語 Python へのマッピングをおこなう IDL compiler , fnidl, を備えている。さらに CORBA Services の内 Naming server (fnaming) および Interface repository (fnifr) も実装している。

2.2 Python

Python[3] は CNRI(Corporation for National Research Initiative, 非営利の計算機およびネットワークに関する研究および研究支援を行う団体) が処理系を配付しているオブジェクト指向プログラムプログラミング言語である。現在のところC言語で実装された Python(CPython ともよばれる）と Java で実装された JPython の二つの実装系が配布されている。

Python の実装は Python を単独の Interpreter として使うことだけではなく、その他のプログラムと組み合わせる事を考慮して行われている。これにより Python に独自の機能を組み込むことや、既存のプログラムに Python を組み込むことでそのプログラムに制御機能を付け加えることが簡単に行えるようになっている。

Python は数多くの OS で稼働実績があり、Linux を含むほとんどの Unix, Windows(95/98/NT), DOS, Mac OS, BeOS 等で動作する。　また Tk ウイジェットを使う Tkinter などのモジュールを使うことでポータブルな GUI をもつアプリケーション開発が可能となっている。

X, Windows, Mac それぞれで動作する GUI インタフェースも Tkinter 以外に用意されている。

3 Network Semaphore の設計

3.1 セマファ

セマファは、実時間処理用の計算機システムでよく使われるプロセス間同期などのための仕組みである。通常 OS の提供する基本的な機能として提供され、OS によってさまざまな型が用意されている。基本となるのは、0 あるいは 1 の値をとるバイナリセマファである。実時間 OS のひとつである VxWorks ではこのほかに、計数セマファ(counting semaphore)、相互排除セマファ(mutulal-exlusion semaphore) が用意されている。

(バイナリ）セマファの基本的な操作は, セマファを最初に (?) 導入した Djikstra の用語に従うと、P(Passere, 通過許可）　及び V(Verhoog) である。これら VxWorks のセマファ操作関数 semTake() および semGive() に対応している。元来のセマファでは、P 操作を行った時、既に他プロセスがセマファを確保している場合には P 操作を行ったプロセスはセマファが解放されるまでブロックされる。　VxWorks の semTake() 関数ではオプションとしてタイムアウト時間が指定できる。このタイムアウト時間の間にセマファが解放されないばあいにはタイムアウトエラーを示すステータスコードを返す。

3.2 ネットワークセマファの設計と実装

POSIX の定義するセマファには名前付きと名前なしの二種類のセマファがある。ネットワークセマファでは互いに独立なプロセス、それらは別計算機上のプロセスのであることもある、の間でセマファを共有することになるので、名前付きセマファとするのが適当である。

セマファの利用に際しては、まずセマファを生成することが必要である。これはセマファを管理するサーバに依頼することになる。

生成されたセマファオブジェクトはバイナリセマファとする。したがって P 操作および V 操作に対応す

るメソッドが定義される。実際の応用では Blocking および Non Blocking の P 操作が必要である。これらを考慮して、OMG-IDL によるインタフェースの設計をおこなった [付録 C.1]。この IDL による記述を fnorb で処理することで、サーバを実装するさいに必要となるスケルトンファイルと、クライアントでこのサービスを使う際に必要となるスタブファイルが作成される。

```
abco1.23: /proj/GNU/Fnorb/script/fnidl NetSemaphore.idl
abco1.24: ls NetSemaphore_skel/
__init__.py    __init__.pyc
abco1.25: ls  NetSemaphore/
__init__.py    __init__.pyc
abco1.26:
```

fnidl の使用例:NetSemaphore.idl をコンパイルすると、NetSemaphore_skel/ と NetSemaphore/ が作成される。それぞれのディレクトリの__init__.py がスケルトン定義ファイルとスタブ定義ファイルになっている。

Network Semaphore の実装を Python/Fnorb を用いて行った。現在のバージョンでは (手元の OS で Thread が使えないために）セマファの Take() 時に呼び出し側のプロセスをブロックすることができない。その意味ではこの実装はセマファでは無く、単なるネットワーク共有のフラグである。

ともあれ、実際のコードを付録 C.2 および C.3 に示す。IDL(付録 C.1) では二つの Interface(これは Python でサーバ/クライアントを実装するさいには class にマップされる) を定義している。Server Interface は Semaphore の生成、検索を管理するオブジェクトである。サーバプログラムではこの Server オブジェクトの実体 (instance) を生成し、Fnorb のエベントループを起動している。

クライアント側では、Server オブジェクトから CreateSemaphore あるいは FindSemaphorebyName で取得した Semaphore オブジェクトの実体に Take(),Relase() を呼び出すことで、セマファの状態を変更する。サーバ側では、client でのメソッド呼出を受けて、セマファの管理を行う。現在の実装では、Take() を呼び出すとその返り値としてセマファを解放するためのユニークなキー値 (文字列）を返す。Release() にこのキー値をしていする。キー値がセマファをロックするキー値と一致するときにのみセマファが解放される。

既にセマファがロックされているときに Take() を呼び出すと空文字列 ("") を値として返す。Release() も解放に失敗すると空の文字列 ("") を値として返す。

4 Network Semaphore の実行

4.1 サーバの起動

サーバは server.py を通常の Python プログラムと同様に, 次のコマンド入力で行う。

python server.py

このコマンドによる実行例を以下に示す。

```
abco1.27: python server.py
Initialising the ORB...
Initialising the BOA...
Creating object reference...
```

```
Creating implementation...
Activating the implementation...
Server created and accepting requests...
```

サーバの実行

この例では、サーバは稼働中のプロセスの IOR を server.ref ファイルに書き出す。クライアントはこのファイルを読み出して接続先サーバの IOR を知る。実際の運用ではこの部分は NameServer へ登録するようにすべきである。

4.2 クライアントの実行例

サーバが起動した後、クライアントを起動する。この実行例では、client.main() でサーバへのリファレンスを作成後、CreateSemaphore() メソッドで”test”というセマファを作成している。

```
acsad3.kek.jp.12: python
Python 1.5.2 (#10, May 19 1999, 17:26:05)  [GCC 2.8.1] on osf1V3
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import client,sys
>>> server=client.main(sys.argv)
Initialising the ORB...
>>> sem=server.CreateSemaphore("test")
>>> key=sem.Take()
>>> print key
01DD7D7C-E845-FE75-DC9A-552AFCEE
>>> sem.Release(key)
'01DD7D7C-E845-FE75-DC9A-552AFCEE'
>>>
```

クライアント実行例

上記のクライアント側プログラムを実行するとサーバ側では次の様なメッセージを出力する。

```
*** creating semophore
*** linked obj with boa
locked 01DD7D7C-E845-FE75-DC9A-552AFCEE
released 01DD7D7C-E845-FE75-DC9A-552AFCEE <__main__.Semaphore instance at 4011c7f0>
```

サーバの実行記録

この Network セマファの Take(), Release() に要する時間を簡単に測定してみた。Digital Unix3.2g の動作している Alpha 8200 サーバ上では、

```
>>> t1=time.time();key=sem2.Take();t2=time.time()
>>> print t2-t1
0.0448499917984
>>> t1=time.time();key=sem2.Release(key);t2=time.time()
```

```
>>> print t2-t1
0.0331499576569
```

ネットセマファサーバの実行時間の測定。

となった。このとき、Server は HP-UX10.2 の動作する Melcom RK-460 で稼働している。

Take(), Release() いずれの場合にも、サーバで実際にメソッドの処理にかかっている時間は 1msec 以下であるので、これらの実行時間のほとんどが fnorb によるオーバーヘッドだと考えられる。

5 Naming Server

Fnorb には CORBA で規定される Naming Service を実装した Naming server も含まれている。Naming Server を利用することで、server-client 間で IOR のやり取りを自動的に行うことができる。ただしこの場合には、Naming Server の IOR を server/client がどうやって知るかという問題は残ってしまう。

Fnorb の naming server, fnaming, の起動は次のコマンドを使用する。

```
./fnaming --ior >/tmp/fnaming.ior&
```

“ior” は fnaming に自らの IOR を標準出力に出力することを指令するためのオプションである。

fnorb のプログラムに Naming server の使用とその IOR を伝えるには三つの方法がある。

1. fnorb の configuration ファイルに Naming service エントリを付け加える。
2. 環境変数 FNORB_NAMING_SERVICE に Naming Server の IOR を割当てる。
3. 実行時の引数に Naming server の IOR を与える。

環境変数をつかった方法では、例えば:

```
setenv FNORB_NAMING_SERVICE ‘cat /tmp/fnaming.ior ‘
```

などとすることができる。名前付きセマファの場合、名前の管理は Naming Server で行うことも可能となる。

6 結論

Python/fnorb を用いて簡単な CORBA アプリケーション ”ネットワークセマファ”を作成した。ネットワークプロトコル等の詳細を議論することなく、目的とするアプリケーションにオブジェクト指向のプログラミング手法を適用することができ、プロトタイプの開発としては極めて短い時間で終えることができた。(この例の実装では 1 人・日)

現在のバージョンは通常の意味のセマファのように Calling process をロックすることができない。このプロセスロックを実現するためには、Netsemaphore サーバで各要求を thread に割当て、サーバ上のセマファによってこれらの thread の実行を制御すればよい [9]。

この実験を行うまで、筆者は CORBA はともすれば大掛かりな仕組みが必要なものとの印象を持っていたが、この実験を通じて CORBA をつかった分散環境の作成が身近なものに感じられた。分散オブジェクト環境がより一般的になることで、これまでにない応用がうまれでるのでは無いかと期待している。

謝辞

この Network semaphore のアイデアについて議論に付き合って下さった小田切淳一氏、中村達郎氏に感謝いたします。また、CORBA について協力をいただいた上窪田氏にも合わせて感謝いたします。

A Python と Fnorb の入手とインストール

Python と Fnorb はいずれも非商用の利用であればフリーに利用できるソフトウェアである (Python はいかなる応用であっても無料で利用可能である)。Python、Fnorb はいずれもインターネット上のサーバから入手可能である。

Python http://www.phthon.org/

Fnorb http://www.dstc.edu.au/Fnorb/

これらの URL から tar.gz 形式のファイルをダウンロードする。

A.1 Python のインストール

norb は configure スクリプトの中で python がインストールされているディレクトリを調べるので、fnorb の configure を実行する前に、必ず python をインストールしておく必要があることに注意しておく。Python では展開されたディレクトリの直下で

```
./configure
```

を実行する。これによって必要な Makefile 等が作成される。筆者のこれまでの経験ではあとは、

```
make
```

を行うだけでコンパイルは問題なく行われる。インストールは、

```
make install
```

で行う。

Python では make のターゲットとして、test が定義されているので、コンパイルされた python プログラムのテストを、

```
make test
```

で行うのがよいだろう。

複数のプラットフォームをサポートする必要がある場合 (KEK の SAD 計算機のように) には、通常 configure スクリプトを実行するディレクトリで、プラットフォーム毎に、

```
mkdir ‘uname -s‘
```

を実行しプラットフォーム毎のディレクトリを作る。それぞれのディレクトリに移動した後、

```
env srcdir=.. VPATH=.. ../configure
```

を実行する。

A.2 fnorb のインストール

fnorb の配付ファイルを展開した後、fnorb/src ディレクトリに移動する。ここで、

```
make -f Makefile.pre.in boot
```

を実行する。これで Makefile ができ上がるのであとは

```
make
make install
```

としてコンパイルおよびインストールを実行する。複数のプラットフォームをサポートするためには、Python と同じくプラットフォーム毎のディレクトリで、

```
make -f ../Makefile.pre.in boot srcdir=.. VPATH=..
```

を実行しプラットフォーム毎の Makefiel を作成する。fnorb にはいくつかの例が用意されているので、これらの例を稼働させることで、fnorb の動作を確認するできる。

B その他の Free な CORBA 処理系

B.1 ILU

ILU(Inter-Language Unification system) は Xerox PARC(Palo Alto Research Center) で開発された分散オブジェクトによるアプリケーション開発のためのシステムである。名前が示すように複数のオブジェクト指向言語をつかった分散オブジェクトアプリケーションをサポートしている。元々は CORBA とは独立に開発され、独自の ISL(ILU's Interface Specification Language) をそなえていた。CORBA との連携を可能とするように、OMG-IDL C++, ANSI C, Python, Java, および Common Lisp をサポートしている。PARC 以外で開発された Modula-3, Guile Scheme, および Perl 5 のサポートも存在する。ILU の稼働する OS は種々の Unix (SunOS, Solaris, HP-UX, AIX, OSF, IRIX, FreeBSD, Linux, LynxOS, SCO Unix, etc.) and MS-Windows (3.1, 95, NT) と広範囲にわたる。また、ILU は Thread と event loop の両方の運用形態をサポートしている。ILU は OMG-IDL をサポートし、CORBA-ORB システムとしてとらえることも可能ではあるが、CORBA を越えた機能の拡張がなされている。ILU ではまた、すでに存在する RPC のサービスを ILU のオブジェクトとして利用可能である。ILU は HTTP の実装をその中に含んでおり、オブジェクト指向のウェブブラウザー・サーバを実装するために利用することができる。

現在は ILU2.0 Alpha が配付されている。この版では、CORBA IIOP, the World Wide Web HTTP protocol, the World Wide Web Consortium's HTTP-NG protocols, Sun RPC (ONC RPC), a CORBA-compliant C mapping, a CORBA-compliant C++ mapping, a CORBA-compliant Java mapping (both JDK 1.2 and JDK 1.1 support), Python (through Python 1.5.x), Common Lisp , OS kernel thread , security via GSS-based context negotiation and on-the-wire message encryption, multiple languages in the same address space communicating via ILU calls, OMG IDL as well as ILU's ISL, GNU autoconf for configuration, an ANSI C XML parser, an implementation of the IETF Common Authentication Technology working group's GSS, URLs, new transport semantics to allow filtering などがサポートされている。ドキュメントも改善された。現在のアルファ版は C, Java, Common Lisp, and Python をサポートしている。C++ and Guile Scheme のサポートはテスト的な形でのみ用意されている。Modula-3 と Perl を ILU と使うためのソフトウェアは別のグループから入手可能である。ILU 著作権はそれがフリーソフトウェアであるとしている。

B.2 OmniORB2

OmniORB2 は AT&T Cambridge 研究所で開発されたフリーな CORBA2 準拠の ORB である。GNU パブリックライセンスにしたがって配付されている。最新版は 2.7.1 である。OmniORB2 は C++言語へのマッピングをおこなう。マルチスレッドを差ポートする。IIOP を Native transport としてつかう。また COS Naming service が付属する。開発者の主張によれば、安定したまた最速の CORBA2 準拠の ORB である。

B.3 MICO

MICO は " **M**ICO **I**s **CO**RBA." の省略形である。このプロジェクトの目的はフリーで CORBA2.2 に完全に準拠した実装を作り出すことである。MICO は OpenSource のプロジェクトの一つとして名前がよく知られている。またさまざまな用途に広く使われている。このプロジェクトの最終目的は MICO を最新の CORBA 標準に準拠したものに保っていくことである。MICO のソースコードは GNU 著作権同意書の下に置かれている。

B.4 TAO

ACE ORB(TAO) は "オープンソース" で高性能な CORBA2.2 準拠の ORB である。元来実時間処理の目的に開発されたために、QoS(Quoality of Service) を提供する。TAO は、most flavors of UNIX, Linux and NT, as well as real time systems such as Lynx, PSOS and VxWorks, and embedded systems like Windows CE. などさまざまな OS 上で稼働する。TAO は 広く使われている Adaptive Communication Environment (ACE) ライブラリやパターンフ・レームワークを使っている。その結果、TAO は非常に柔軟性にとんだ設計となっており、異なるプラットフォームにも簡単に拡張できる。

さらに、ネットワークやシステムが QoS をサポートすれば、TAO はそれらの機能を素早く取り込んで行くことが可能である。

B.5 ORBacus

ORBacus (formerly known as OmniBroker) は頑丈で高機能の ORB である。ORBacus はすでに確立された実績と大企業級の機能を有している。

- Unsurpassed portability (Windows 95/98/NT, UNIX, Linux)
- Complete C++ and Java mappings
- Naming, Event and Property services included
- Pluggable protocols
- Fully multi-threaded
- Full support for DII, DSI, IR and DynAny
- HTML and RTF documentation generators
- SSL, Trading and other extensions available

ORBacus は CORBA 仕様に完全準拠であり、非商用使用にはすべてのソースコードを含め無償で提供されている。

C Network semaphore のソースコード

C.1 インターフェイス 定義

```
#pragma prefix "kekb.at.kek.jp"

module NetSemaphore {
        typedef string Name;
        typedef string Key;

        interface Semaphore{
                Key Take();
                Key Release(in Key key);
                long Wait();
        };

        interface Server {
                Semaphore CreateSemaphore(in Name name);
                Semaphore FindSemaphorebyName(in string name);
        };
};
```

C.2 サーバ実装

```
#!/usr/bin/env python
##############################################################################
""" Implementation of the NetSemaphoredIF interface. """
# Standard/built-in modules.
import sys

# Fnorb modules.
from Fnorb.orb import BOA, CORBA, uuid

# Stubs and skeletons generated by 'fnidl'.
import NetSemaphore, NetSemaphore_skel

_Semaphores={}

class Server(NetSemaphore_skel.Server_skel):
```

```
    """ Implementation of the 'Server' interface=class. """
    def __init__(self):
        apply(NetSemaphore_skel.Server_skel.__init__,(self,))

    def CreateSemaphore(self,name):
        global _Semaphores
        if _Semaphores.has_key(name):
            print "DuplicateSemaphore"
            return None

        print "*** creating semophore"
        sem=Semaphore(name)

        # create object reference
        boa=BOA.BOA_init()
        objref=boa.create(repr(sem), Semaphore._FNORB_ID)

        # bind objref to a semaphore object
        boa.obj_is_ready(objref, sem)
        print "*** linked obj with boa"

        return  sem

    def FindSemaphorebyName(self,name):
        global _Semaphores
        if _Semaphores.has_key(name):
            return _Semaphores[name]
        else:
            return None

class Semaphore(NetSemaphore_skel.Semaphore_skel):
    def __init__(self,name):
        global _Semaphores
        apply(NetSemaphore_skel.Semaphore_skel.__init__,(self,))
        _Semaphores[name]=self
        self.name=name
        self.locked=0
        self.key=0

    def  Take(self):
        print dir(self)
        print self.__dict__
```

```
        if self.locked:
            print "already locked",repr(self)
            return ""
        else:
            self.locked=1
            self.key=self.genKey()
            print "locked",self.key, repr(self)
            return self.key

    def genKey(self):
        return uuid.uuid()

    def  Release(self,key):
        if self.locked and self.key:
            if self.key == key:
                print "released",key,repr(self)
                self.locked=0
                self.key=0
                return key
            else:
                return ""
        else:
            return ""

    def  Wait():
        pass

    def  __del__():
        del _Semaphores[self.name]
```

C.3 クライアント例

```
# Standard/built-in modules.
import sys

# Fnorb modules.
from Fnorb.orb import CORBA

# Stubs generated by 'fnidl'.
import NetSemaphore
```

```
def mkServermkServer(argv):
    """ Do it! """

    print 'Initialising the ORB...'

    # Initialise the ORB.
    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)

    # Read the server's stringified IOR from a file (this is just a 'cheap and
    # cheerful' way of locating the server - in practise the client would use
    # the naming or trader services).
    f = open('server.ref', 'r')
    stringified_ior = f.read()
    f.close()

    # Convert the stringified IOR into an active object reference.
    server = orb.string_to_object(stringified_ior)

    # Make sure the object implements the expected interface!
    if not server._is_a(NetSemaphore.Server._FNORB_ID):
        raise 'This is not a "HelloWorldIF" server!'

    # Call the server!
    return server

################################################################################

if __name__ == '__main__':
    # Do it!
    server=mkServer(sys.argv)

################################################################################
```

D 用語

ORB:Object Request Broaker

CORBA:Common Object Request Broaker Architecture

GIOP:General Inter-ORB Protocol ORB 間の相互運用性確保のために OMG によって規定されたプロトコル規約。データ交換の際の形式と標準的なメッセージの形式のセットを規定している。データ交換に使われるネットワークプロトコルには依存しない。

IIOP:Internet Inter-ORB Protocol GIOP を TCP/IP 上で実現する際の方法について規定している。CORBA2.2 では GIOP と IIOP のサポートは必須の条件である。

*BOA:Basic Object Adapter*BOA は ORB と CORBA オブジェクトの実装との間で、CORBA の API を実装の API に変換する役割を担う。

*POA:Portable Object Adapter*BOA では、十分に規定されていないところがあるため、BOA の実装によっては相互運用ができない場合がある。この問題を解決し CORBA に準拠する ORB 間での相互運用性を高めるために規定された。fnorb では POA のサポートは予定されてはいるものの、BOA だけがサポートされている。

IOR: Interoperable Object Reference

UOL:Universal Object Locator Fnorb 独自の拡張機能である。OMG IOR を拡張し、容易に IOR を交換できるようにした。UOL は WWW における URL(Universal Resource Locator) のように IOR データを交換するためのプロトコルと IOR ソースを示すパスネームから構成される。現在サポートされるプロトコルは、'IOR'(stringfied object reference), 'file'(local file), 'http'(URL), 'name'(naming server) の四つである。

COSNaming:CORBAServices Naming service OMG が規定する標準的なサービス（CORBA Services）の内ネーミングサービスについての規約。

参考文献

[1] The Common Object Request Broker Architecture and Specification Revision 2.0, Object Management Group, July 1995, Updated 1996.; OMG Home Pag , http://www.omg.org/

[2] M. Chilvers, et al., "fnorb user's guide", April 1999, http://www.dstc.edu.au/Fnorb

[3] Python Home page, http://www.phthon.org/; A.Watters, G. von Rossum, J.C. Ahlstrom, "Internet Programing with Python", M&T Books, New York, 1996; Mark Lutz "Programming Python", O'Reilly & Associates, Inc., 1996 (邦訳「Python 入門」「Python プログラミング」, オライリージャパン、Tokyo, 1998; M. Lutz & D. Ascher , " Learning Python", Oreill & Associates, Inc, Sebastopol, USA, 1999

[4] omniORB, http://www.uk.research.att.com/omniORB/omniORB.html

[5] ILU (Inter Language Unification), ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

[6] ORBacus, http://www.ooc.com/

[7] MICO (Mico is COrba), http://www.mico.org/

[8] TAO and ACE, http://siesta.cs.wustl.edu/ schmidt/TAO.html

[9] 小田切　淳一 "ソケット・インタフェースを用いたネットワーク・セマフォアの実装の試み"、private communication