Un MiddleWare Objet : ICE

- Qu'est ce que c'est ?
  - Middleware objet
  - Architecture client/serveur pour le développement d'applications distribuées
  - Support des l'hétérogénéité :
    - Des environnements de développement
    - Des environnements d'exécution

Un MiddleWare Objet : ICE

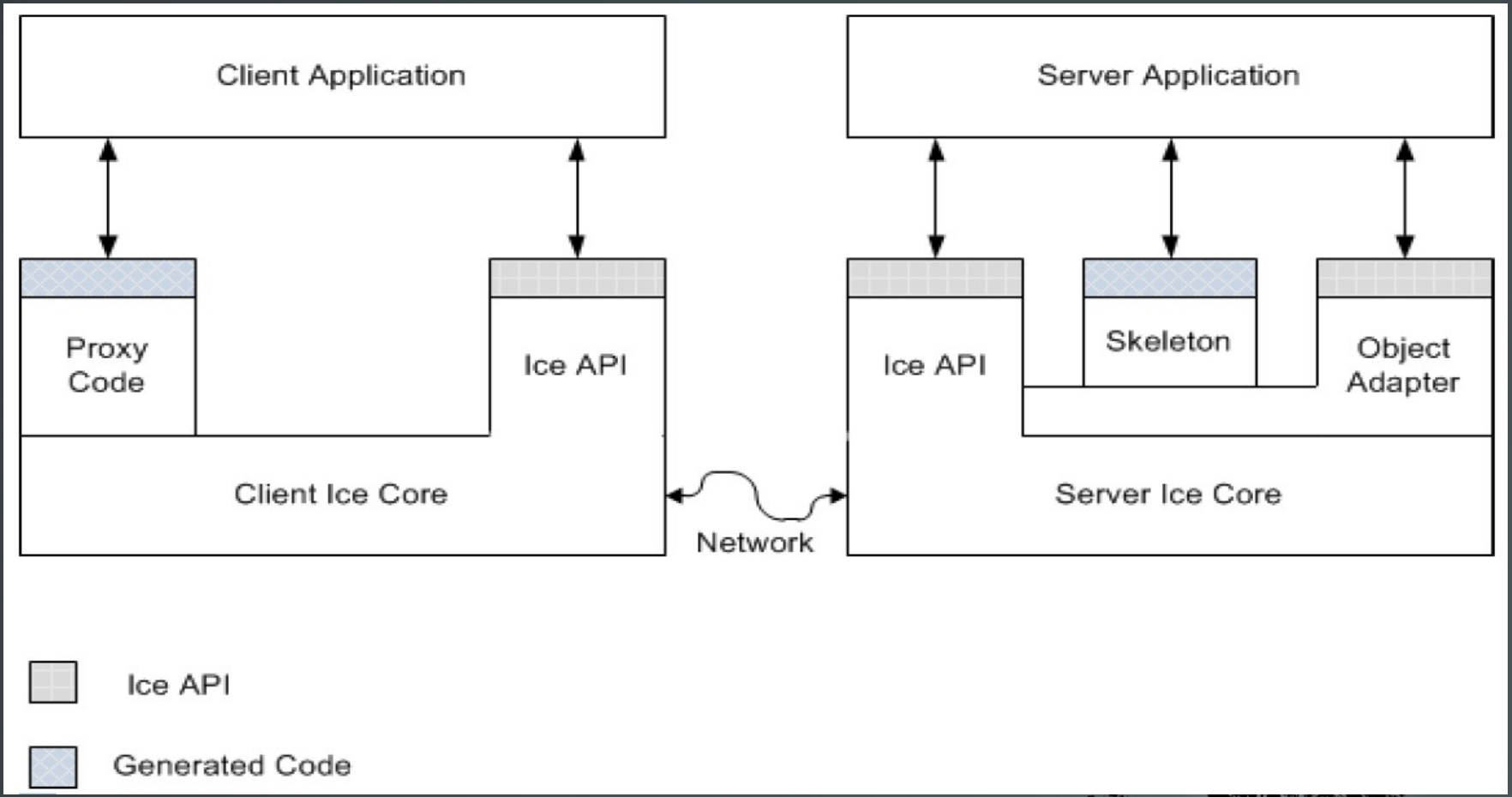
- D'ou ça sort ?
  - Développé par Zero C (industriel)
  - Distribué sous licence GPL
  - http://www.zeroc.com/
  - Support :
    - C++ , Java, C#, Objective C, Python, Ruby, PHP

ICE : plan du cours

- Architecture
- Le langage de spécification : slice
- Le Mapping
- Les Services

ICE : architecture

- Architecture client/serveur



ICE : architecture

- Proxy :
  - Avatar local du serveur produit à partir des définitions (en slice)
  - Intègre les fonctions de *marshalling : sérialization de structures de données complexes.*

ICE : architecture

- Skeleton :
  - produit à partir des définitions (en slice)
  - En charge de la transmission des informations envoyées par le proxy
  - Intègre aussi le code de marshalling/unmarshalling

ICE : architecture

- *Object adapter :*
  - Partie de l'API spécifique au coté serveur
  - Responsable de l'activation *ie* du lien entre une requête et l'objet qui va l'exécuter
  - Génère les références (les proxy)

ICE : Scénari de développement

- Développement à partir de zéro :
  - (1) Spécifications : interfaces Slices
  - (2) Mapping : projection des interfaces vers les langages de programmation cibles
  - (3) développement du code applicatif
- Intégration :
  - (1) Spécifications : interfaces Slices
  - (2) Mapping : projection des interfaces vers les langages de programmation cibles
  - (3) développement des coquilles : classes encapsulant les fonctionalités pré-existantes

ICE : exemple de serveur

```
#include <iostream> ;#include <Ice/Ice.h> ;#include <printer.h>
using namespace std;using namespace Demo;


class PrinterI : public Printer {
public: virtual void printString(const string& s, const Ice::Current&);
};
void     PrinterI::printString(const string& s, const Ice::Current&){
cout << s << endl;
}


int main(int argc, char* argv[]){
int status = 0;
Ice::CommunicatorPtr ic;


try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter =ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
        } catch (const Ice::Exception& e) {
        cerr << e << endl;status = 1;
} catch (const char* msg) {
cerr << msg << endl;status = 1;
}
if (ic) {
        try {ic->destroy();} catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
}}
return status;
}
```

ICE : exemple de client

```
#include <Ice/Ice.h>
#include <printer.h>

using namespace std;
using namespace Demo;
int main(int argc, char* argv[]){
int status = 0;
Ice::CommunicatorPtr ic;
try {
ic = Ice::initialize(argc, argv);
Ice::ObjectPrx base = ic->stringToProxy("SimplePrinter:default -p 10000");
PrinterPrx rinter = PrinterPrx::checkedCast(base);
if (!rinter)
throw "Invalid proxy";
rinter->printString("Hello World!");
} catch (const Ice::Exception& ex) {
cerr << ex << endl;
status = 1;
} catch (const char* msg) {
cerr << msg << endl;
status = 1;
}
if (ic)
ic->destroy();
return status;
}
```

ICE : SLICE

- Langage de spécification
- Indépendant de la plate-forme
- *Mappings* :

Language	Compiler
C++	`slice2cpp`
Java	`slice2java`
C#	`slice2cs`
Objective-C	`slice2objc`
Python	`slice2py`
Ruby	`slice2rb`
PHP	`slice2php`

ICE : SLICE

- Modules & interfaces similaires à IDL
- Types basiques :

Type	Range of Mapped Type	Size of Mapped Type
`bool`	`false` or `true`	? 1bit
`byte`	-128-127 or 0-255 [a]	? 8 bits
`short`	-2^{15} to $2^{15}-1$	? 16 bits
`int`	-2^{31} to $2^{31}-1$	? 32 bits
`long`	-2^{63} to $2^{63}-1$	? 64 bits
`float`	IEEE single-precision	? 32 bits
`double`	IEEE double-precision	? 64 bits
`string`	All Unicode characters, excluding the character with all bits zero.	Variable-length

ICE : SLICE

- Séquences :
  - sequence<fruit> FruitPlatter ;
- Structures :

Slice

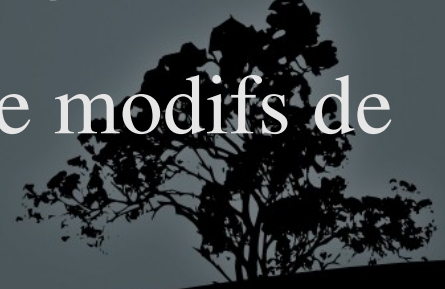
```
struct Part {
    string name;
    string description;
    // ...
    bool   serialIsValid;  // true if part has serial number
    long   serialNumber;
};
```

ICE : SLICE

- Ensembles
  - dictionary<string, string> NomAdresse ;
  - Couples <clefs,valeurs>
  - Réservés aux type de base :
    - Entiers, string, enum, structures contenant des entiers ou des chaînes
- Constantes :
  - *const int Max = 100 ;*

ICE : SLICE

- Opérations
  - Pas de spécification de passage entrant des paramètres
  - *void solde(float depot, out float solde) ;*
- Par défaut, mode *in : trnasport des clients vers le serveur,* mais mode *out* possible
- Pas de surcharge
- Idempotent : `idempotent TimeOfDay getTime();`
- Comme le const du C++ : pas de modifs de l'état de l'objet



ICE : SLICE

- Exceptions :

```
exception Error {};
exceptions are legal
exception RangeError {
long errorTime;
string reason= « out of range »;
TimeOfDay maxTime;
};
```

ICE : SLICE

- Exceptions :

```
exception RangeError extends LogicError {
TimeOfDay errorTime;
TimeOfDay minTime;
TimeOfDay maxTime;
};

Interface Calendar{
void addMeeting() throws errorTime ;
…
} ;
```

ICE : SLICE

- Exceptions :

```
exception Error {};
exceptions are legal
exception RangeError {
long errorTime;
string reason= « out of range »;
TimeOfDay maxTime;
};
```

ICE : SLICE

- Classes !
  - Permet le passage par référence ou par valeur
  - Héritage et polymorphisme
  - Utilisables comme les *union*s du C

Slice

```
class Location {
    string name;
    Point pt;
    bool display = true;
    string source = "GPS";
};
```

ICE : SLICE

- Classes
- Héritage simple seulement
  - interface Fichier extends Flux {…} ;
  - Classe fichier_cls extends flux_cls {…} ;

ICE : SLICE

- Metadata : informations passées au *backend*
  - *ie aux processus de traitement des définitions slices, typiquement le compilateur*
- Syntaxe :
- *["java:type:java.util.LinkedList"] sequence<int> IntSeq;*
- Les metadatas ne font pas vraiment partie du langage.

ICE : Mapping C++

- Modules : namespace
- enum → enum
- struct → struct

sequence<float> signal //slice →

typedef std::vector<ice ::float> signal ;//C++

signal s ;

s.push_back(3) ;

ICE : Mapping C++

```
[["cpp:include:list"]]
module image {
 enum forme { carre, cercle, losange };
["cpp:type:std::list< ::image::forme>"]
 sequence<formes> dessin;
};
```

→

```
#include <list>
namespace Food {
    typedef std::list< Food::Fruit> FruitPlatter;
}
```

ICE : Mapping C++

```
dictionary<tel,string> annuaire; //slice

typedef std::map<Ice::Long, string> annuaire;//C++

//exemple d'utilisation

annuaire em;

individu e;

e.number=04889989 ;

e.firstName = "Stan";

e.lastName = "Dupont";

em[e.number] = e;
```

ICE : Mapping C++

- Exceptions :
- Classes C++ qui dérivent de Ice::UserException ;

```
exception GenericError { string reason;};//slice
class GenericError: public Ice::UserException{//C++
public:
   std::string reason;
...} ;
```

ICE : Mapping C++

- Interfaces
- Coté client
  - Proxy :
    - Dans un espace de noms IceProxy
    - hérite de Ice::Object
  -

ICE : Mapping C++

- Interfaces

```
module Mod {        //slice
    interface Inter {
        Idempotent string op();
    };
};
```

ICE : Mapping C++

```
namespace IceProxy {                                    //C++
    namespace Mod {
        class Inter;
    }
}
namespace Mod {
    class Inter;
    typedef IceInternal::ProxyHandle< ::IceProxy::Mod::Inter>
InterPrx;
    typedef IceInternal::Handle< ::Mod::Inter> InterPtr;
}
namespace IceProxy {
    namespace Mod {
        class Inter : public virtual IceProxy::Ice::Object {
        public:
            typedef ::Mod::InterPrx ProxyType;
            typedef ::Mod::InterPtr PointerType;
            string op();
            string op(const Ice::Context&);
            // ...
        };
    };
}
```

ICE : Mapping C++

- Interfaces coté client :
- Pas d'instanciation directe du proxy
- Garbage collector sur la classe proxy
- Manipulation par l'intermédiaire d'une classe ProxyHandle : *<interface_id>Prx*

ICE : Mapping C++

- Interfaces coté client :

```
try {
    InterPrx s;        // Cration d'un proxy par défaut
    cout << s->op();           // invocation standard
    .....
} catch (const IceUtil::NullHandleException&) {
    cout << "Le proxy ne pointe sur aucun serveur ! » ;
}
```

ICE : Mapping C++

- Interfaces coté client :
- Les interfaces sont *stringifiables :*
  - *Mod::InterPrx m ;*
  - *....*
  - *cout << m ;*
  - *cout << m->ice_toString() ;*
  - *cout<<communicator->ProxyToString(m) ;*
  - *m=communicator->StringToProxy() ;*

ICE : Mapping C++

- Main serveur :

```
int main(int argc, char* argv[]){
   Ice::CommunicatorPtr ic = Ice::initialize(argc, argv);
   Ice::ObjectAdapterPtr adapter =
        ic->createObjectAdapterWithEndpoints("InterAdapter", "default -p
    10000");
      Ice::ObjectPtr object = new InterPrx;
      adapter->add(object, ic->stringToIdentity("Interface"));
      adapter->activate();
      ic->waitForShutdown();.
}
```

ICE : Mapping C++

- Skeleton :

```
namespace Mod {
   class Inter : virtual public Ice::Object {
   Public:
virtual std ::string Op(const Ice::Current& = Ice::Current()) = 0;
   };
```

ICE : Mapping C++

- Servant :

```
#include <monAppliIce.h> // Slice-generated header

class InterI : public virtual Mod::Inter {
public:
    InterI(const std::string&);
    virtual std::string op(const Ice::Current&);
private:
    std::string _op;
};
```

ICE : Mapping C++

- In*stanciation du servant :*
  - *InterPtr monservant=new InterI(« toto ») ;*
- *Ice::Identity id ; id.name = « toto » ;*
- *Activation du servant :*

  _adapter->add(monServant,id) ;
- *La destruction est gérée automatiquement, par comptage des références*

ICE : Mapping C++

- Exemple (exercice)
- Serveur de reconnaissance de la parole (commande vocale)
- Principe :
  - un serveur connaît un ensemble limité de commandes
  - cet ensemble peut être modifié par l'utilisateur
  - la reconnaissance consiste à transmettre un signal. Le serveur renvoie la transcription de la commande contenue (une chaîne de caractères)

ICE : Asynchronous Method Invocation

- Invocation **asynchrone :** ne bloque pas le *thread* appelant
- A partir de la version 3.4
- Concerne le coté client
  - Le serveur de connaît pas le mode d'invocation
- *Onewav* et *twoway* invocations
  - *Oneway* : aller simple vers le serveur
  - *Twoway* : aller retour

ICE : AMI

- API standard
- Principes :
  - Mode *twoway* par défaut
  - Dissocie l'appel du retour
    - Méthodes *begin()* et *end()*...
  - Basé sur des pointeurs intelligents
    - Objet structuré donnant des infos sur l'état d'avancement et les acteurs de la requête!
  - Attention : ça n'est pas possible systématiquement sur les middlewares objets, par exemple pas en CORBA

ICE : AMI

- Mode twoway par défaut : exemple

```
// annuaire.ice
module Annuaire {
interface tel {
        void ajout(string nom, string num);
        void suppression(string nom);
        string numero(string nom);
};};
```

```
// annuaire.h

::Ice::AsyncResultPtr begin_numero(const ::std::string& nom)
::std::string end_numero(const ::Ice::AsyncResultPtr&);
```

ICE : AMI

- Mode twoway par défaut : exemple

```
// client.cc

AnnuairePrx a ;

Ice::AsyncResultPtr r= a->begin_numero(« toto ») ;    // appel non bloquant
……..
string nom=a->end_numero(r) ;                         // récuperation du résultat
```

ICE : AMI

- Coté proxy : la classe *AsyncResultPtr*
  - Encapsule les informations liées au processus asynchrone
    - *bool operator==(const AsyncResult&) const*
    - *bool operator<(const AsyncResult&) const*
    - *int getHash() const*
    - Gestion d'une collection de requêtes en cours

ICE : AMI

- Coté proxy : la classe *AsyncResultPtr*
  - Encapsule les informations liées au processus asynchrone
    - *CommunicatorPtr getCommunicator() const ;*
    - *virtual ConnectionPtr getConnection() const*
    - *virtual ObjectPrx getProxy() const*
    - *const string& getOperation() const*
    - *LocalObjectPtr getCookie() const*

ICE : AMI

- Coté proxy : la classe *AsyncResultPtr*
  - Encapsule les informations liées au processus asynchrone
    - *bool isCompleted() const*
      - *vrai si la requête est achevée*
    - *void waitForCompleted()*
      - *attention l'achévement*

ICE : AMI

- Coté proxy : la classe *AsyncResultPtr*
  - *bool isSent() const*
  - *void waitForSent()*
    - *Attend (interroge au sujet de) l'envoi de la requête*
  - *Les requêtes sont empilées dans un buffer en attendant le transport... ces méthodes informent sur le dépilement de la requête*

ICE : AMI

- Coté proxy : la classe *AsyncResultPtr*
  - Encapsule les informations liées au processus asynchrone
    - *void throwLocalException( ) const*
      - *Force l'exception*
    - *bool sentSynchronously( ) const*
      - *La requête est-t-elle synchrone ?*

ICE : AMI

- Coté proxy : la classe *AsyncResultPtr*
  - Encapsule les informations liées au processus asynchrone
    - *void throwLocalException( ) const*
      - *Force l'exception*
    - *bool sentSynchronously( ) const*
      - *La requête est-t-elle synchrone ?*

ICE : Les services

- *IceGrid :* service pour le calcul en grille
- *IceStorm :* service de communication par messages
- *Freeze :* service persistance d'objets
- *Glacier2 : firewall*
- *IcePAtch2 :* déployement

Les services ICE : IceGrid

- *IceGrid :* service pour le calcul en grille
- Objectifs : calcul parallèle, exploitation de ressources de calcul
- Fonctionnalités :
  - Découverte de nouvelles ressources (serveurs de calcul)
  - Découplage clients/serveurs
  - Équilibrage de charge
  - Réplication de serveurs
  - Stratégies d'activation
  - Outils d'administration