Plan

- Rappels (middleware, Ice)
- Invocations asynchrones : AMI
- Configurer l'architecture coté serveur :
  - *Threads, communicators, adapters*
- Un service pour le calcul intensif : IceGrid
- Communication par messages : IceStorm
- Un middleWare orienté Message : ActiveMQ

Rappel :
Un MiddleWare Objet : ICE

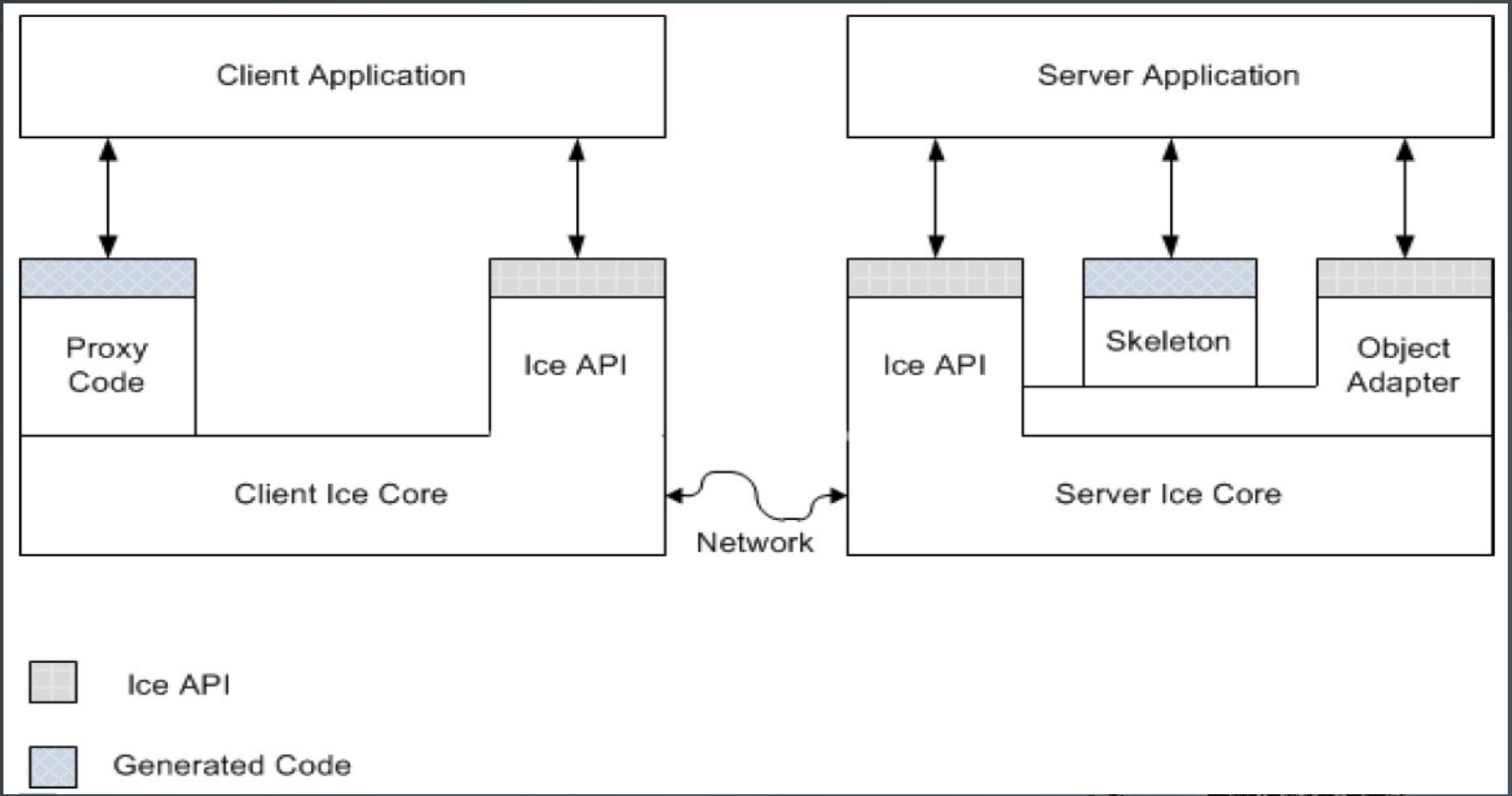
- Qu'est ce que c'est ?
  - Middleware objet
  - Architecture client/serveur pour le développement d'applications distribuées
  - Support des l'hétérogénéité :
    - Des environnements de développement
    - Des environnement d'exécution

Rappel :
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 : 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 : SLICE

- langage de spécification
- Indépendant de la plateform
- Mappings :

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

ICE : Mapping C++

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

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

- Un exemple :
  - Définitions *Slice*

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

ICE : AMI

- Un exemple (serveur) :

```
int main(int argc, char* argv[]){
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter = ic-
>createObjectAdapterWithEndpoints("MonServeurAdapter", "default -p 10000");
        Ice::ObjectPtr object = (Ice::ObjectPtr)new anServerI;
        adapter->add(object, ic->stringToIdentity("MonServeur"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    ......
```

ICE : AMI

- Un exemple (client) :

```
...
        ic = Ice::initialize(argc, argv);
        Ice::ObjectPrx base=ic->stringToProxy("MonServeur:default -p 10000");
        telPrx an = telPrx::checkedCast(base);
        if (!an)
            throw "Invalid proxy";
                an->ajout("linares","123435343");
                cout << an->numero("linares")<<endl;

                Ice::AsyncResultPtr asptr= an->begin_numero("linares");

                cout <<"la ça je fais ce que je veux"<<endl;
                cout<< "requete envoyée? "<<asptr->isSent()<<endl;
                cout <<"  requete complete? "<<asptr->isCompleted()<<endl;
                cout << "resultat : "<<an->end_numero(asptr)<<endl;

...
```

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

ICE : Threads

- Par défaut, ICE est *Multithread*
  - *Tous les systèmes de supportent pas le multithreading*
  - *Il y a une grande variété de technologies pour le multithreading*
  - *Ice propose une couche d'abstraction spécifique*

ICE : Threads

- Rappel : qu'est ce qu'un thread ?
  - *C'est un **processus léger** hébergé dans un processus lourd*
  - *Un thread possède :*
    - *- sa propre pile d'execution*
    - *Un identificateur de thread*
    - *Un pointeur d'instructions*

ICE : Threads

- Rappel : qu'est ce qu'un thread ?
  - *Les threads issus du même processus partagent :*
    - *Le code*
    - *La mémoire*
    - *Les droits (Unix)*
    - *L'environnement (shell, rep. De travail, etc.).*

ICE : Threads

- *Multithreading* en Ice
  - *Fonctionalités :*
    - *Gestion de la concurrence d'accès*
    - *Gestion des threads : création, suppression, contrôle, etc...*
    - *Partie du package IceUtil*

ICE : Threads

- Multithreading en Ice : concurrence d'accès
  - *Multithreading : un thread par invocation*
  - *Problème : accès concurrents à une ressource critique*
  - *Solutions classiques :*
    - *Drapeaux de verrouillage des sections critiques*
  - *5 classes :*
    - *Mutex, remutex : exclusion mutuelle, récursive*
    - *Monitor, Cond : exclusions conditionnelles*

ICE : Threads

- Multithreading en Ice : exemple d'utilisation de Mutex, en C++ :

```
namespace Filesystem {
    // ...
    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        // As before...
    private:
        Lines _lines;
        IceUtil::Mutex _fileMutex;
    };
    // ...
}
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    _fileMutex.lock();
    Lines l = _lines;
    _fileMutex.unlock();
    return l;
}
```

ICE : Threads

- Multithreading en Ice : thread Pools
  - Un *communicator* est un gestionnaire de canaux de communications (cotés serveurs **et** clients)
  - Un communicateur gère deux paquets (pools) de threads :
    - Un *client thread pool*
    - Un *server thread pool*
  - Tous les adaptateurs d'un même communicateur partagent ces pools.

ICE : Runtime

- Communicators
  - Thread pools
  - Properties : ensemble de caractérisitques
    - Couples nom-valeur
    - Configuration possible via un fchier de configuration
    - Propriétés très diverses : niveau de log, theaeds, piles, etc..

ICE : Runtime

- Communicators
  - Object factories :
    - Instanciation de classes
  - Logger object : gestion des traces
  - Default router : firewall
  - Default locator : lien proxy-servant (résolution)
  - Plug-in manager : gestion des extensions du communicator
  - Adapters : adaptateurs d'objets.

ICE : Runtime

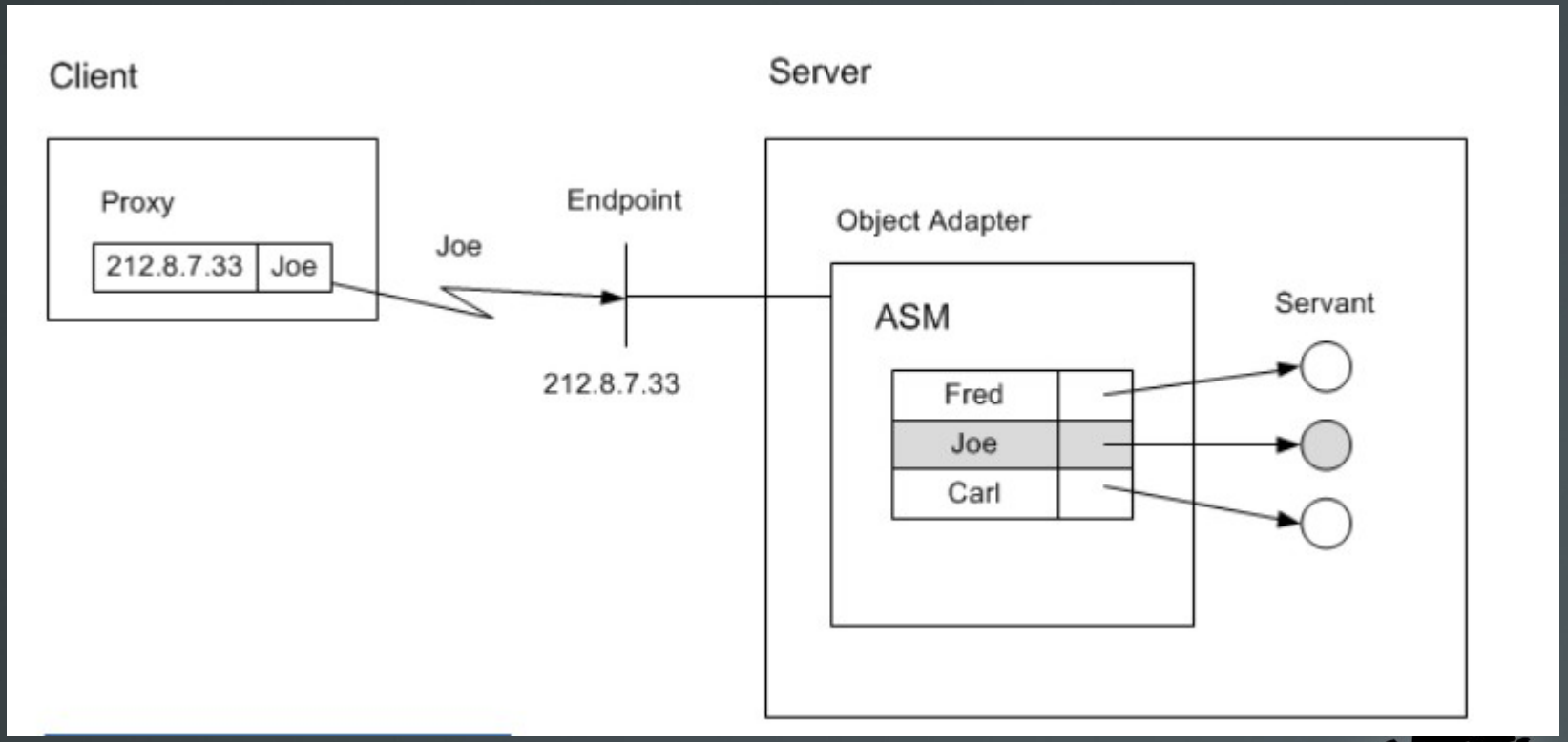
- Adaptateurs d'objets
  - Un adaptateur est attaché à une communicateur
  - L'adaptateur est à l'interface des serveurs et du bus de communication
  - Rôles de l'adaptateur :
    - Routage des requêtes
    - Mapping I*ce object - servants*

ICE : Runtime

- Adaptateurs d'objets
  - Un adaptateur gère un ou plusieurs servants
  - Un servant incarne un objet serveur (Ice Object)
  - Il gère les activations (bindings)
    - Utilisation d'une *Active Servant Map*

ICE : Runtime

- Adaptateurs d'objets : *Active Servant Map*



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