Communication dans les systèmes répartis

Module 4

INF8480 Systèmes répartis et infonuagique

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

Communication dans les systèmes répartis

Avec ou sans connexion

- Sans connexion (UDP):
  - Send ([destinataire], message);
  - Receive ([émetteur], file de stockage);
- Avec connexion (TCP):
  - Connect (destinataire, émetteur);
  - Read, Write...
  - Disconnect (identificateur de connexion);

Synchrone ou asynchrone:

- Le send et receive sont bloquants; les processus se synchronisent à chaque message. Send attend l'accusé de réception!?!
- Le send est non-bloquant, si la file du système d'exploitation n'est pas pleine, tandis que receive est généralement bloquant. Si les paquets arrivent trop vite et la file du système d'exploitation est pleine, un message de ralentir est envoyé et les paquets en surplus peuvent être ignorés.
- Le receive aussi est non-bloquant, le processus vérifie de temps en temps, peut demander un signal, ou utilise un appel système de type select ou epoll.

Un ou plusieurs destinataires:

- Avec un bus, on peut envoyer un message à n destinataires sur le bus en 1 unité de temps au lieu de n unités de temps pour des messages séparés.
- Les protocoles Ethernet et IP supportent la diffusion générale (broadcast) ainsi que la multidiffusion (multicast).
- Gestion des membres d'un groupe de multi-diffusion plus complexe s'ils ne sont pas tous sur le réseau local.
- Messages sans connexion non fiables (UDP), ou séquences de messages fiables (Pragmatic General Multicast, PGM) avec accusés de réception négatifs.

Messages de groupe par multidiffusion

- Tolérance aux fautes basées sur des services répliqués sur plusieurs serveurs.
- Amélioration des performances par des services et données répliqués sur plusieurs serveurs.
- Découvertes de services dans un environnement de réseautage spontané.
- Economie de bande passante par multidiffusion des notifications d'évènements, vidéo ou fichiers identiques demandés par de nombreux clients.

Messages de groupe

- Envoi d'un message aux membres d'un groupe.
- Sur réception, le message est remis au processus destinataire.
- Communication non fiable, un seul message UDP est envoyé, par exemple pour annoncer une adresse Ethernet versus IP.
- Communication fiable, avec chaque message livré au moins une fois (valide) ou au plus une fois (intègre)
- Communication atomique, livré à tous ou à aucun.
- Communication ordonnancée: totalement, causalement, par origine.

Messages de groupe atomiques

- Tous le reçoivent ou personne ne le reçoit.
- Envoi à tous du message non-confirmé et demande d'accusé de réception. Retransmission au besoin si l'accusé de réception ne vient pas.
- Si tous ont reçu, message de confirmation pour rendre le message maintenant confirmé disponible aux applications.
- Si des accusés de réception manquent après un certain temps, annuler le message.
- Si un client ne reçoit pas de confirmation, il demande l'état à l'envoyeur puis au besoin vérifie avec un autre ordinateur qui peut prendre la relève.

Messages de groupe totalement ordonnancés

- Chaque message a un numéro de séquence unique, par exemple fourni par un processus séquenceur.
- Chaque récipiendaire attend d'avoir reçu les messages précédents avant de livrer un message à ses applications.
- Un récipiendaire peut devoir laisser tomber des messages plus récents s'il en a trop en attente.
- Un envoyeur peut bloquer s'il a trop de messages en attente d'accusés de réception.

Messages de groupe causalement ordonnancés

- Les messages en provenance d'un même ordinateur arrivent dans leur ordre d'envoi à chaque récipiendaire: numéro de séquence propre à chaque membre du groupe qui envoie des messages.
- Les messages reçus en provenance de plus d'un ordinateur arrivent en ordre causal: chaque processus maintient un vecteur de numéros de séquence de messages de groupe vus venant de chaque processus. Un message avec un certain vecteur n'est pas délivré avant que tous les messages de chaque processus n'aient été reçus, jusqu'au numéro dans le vecteur pour ce processus.

Communication dans les systèmes répartis

RPC versus appel de fonction

- Message de requête et de réponse; écritures et lectures réseau explicites ou appel de fonction distante (RPC).
- Paramètre d'entrée, de sortie ou les deux.
- Pas de passage de paramètre par pointeur.
- Pas de variable globale accessible à tous.
- Doit spécifier comment sérialiser les structures spéciales qui utilisent des pointeurs (chaîne de caractère, vecteur, liste, arbre...) par un langage de définition d'interface ou des attributs sur les déclarations.
- Si l'appelant et l'appelé sont programmés dans différents langages, il faut adapter les différences (e.g. ramasse-miette, exceptions...).
- Arguments supplémentaires: serveur à accéder, code d'erreur pour la requête réseau.

Catégories et historique

- Protocoles simples : SMTP (1982), HTTP (1991), Remote GDB...
- RPC basés sur un langage d'interface (multi-langage): Sun RPC (1984), ANSA (1985), MIG (1985), CORBA (1991), gRPC (2015).
- RPC basés sur un langage spécifique (meilleure intégration): Cedar (1981), Argus, Modula-3 (1993), Java (1995).
- RPC basé sur une machine virtuelle multi-langage (Common Language Runtime) C# (2000).

Langage de définition des interfaces (IDL)

- Déclaration des procédures (nom, et liste et type des arguments).
- Déclaration des types et des exceptions.
- Générateur de code à partir du IDL pour la librairie client (proxy), et la librairie serveur (squelette), dans le langage de programmation désiré.
- Le IDL et le générateur de code ne sont pas requis si la réflexivité permet d'obtenir cette information à l'exécution.

Sémantique des appels

- Mécanismes: envoi simple, accusé de réception, retransmission, filtrage des requêtes dupliquées, mémorisation des réponses.
- **Peut-être**: en l'absence de réponse on ne peut savoir si la requête ou la réponse fut perdue.
- **Au moins une fois**: retransmission sans filtrage des requêtes jusqu'à obtention d'une première réponse.
- **Au plus une fois**: avec retransmission, filtrage des requêtes, et mémorisation du résultat, la procédure est exécutée exactement une fois si une réponse est obtenue et au plus une fois en l'absence de réponse.
- **Dernière fois**: semblable à "Au moins une fois", mais le client n'accepte la réponse que si elle correspond au dernier appel réalisé (utilise un identifiant)

Sun RPC

- Définition d'interface et de types en XDR.
- Procédures avec un argument (entrée), une valeur de retour (sortie), et possiblement un numéro de version.
- rpcgen produit la librairie client (proxy/client stub) et le squelette (server stub) du programme serveur.
- Compléter le squelette du serveur avec l'implantation.
- Initialiser la connection dans le client et utiliser les fonctions proxy en vérifiant les erreurs.

Sun RPC

- Le DNS effectue la localisation du serveur.
- Portmap effectue la localisation du service.
- Les appels sans valeur de retour peuvent être accumulés et envoyés d'un coup pour plus de performance.
- Authentification.
- Sémantique au moins une fois.
- Sur UDP, chaque requête ou réponse est limitée à 8Koctets.

Exemple de fichier XDR

```
const MAX=1000;
typedef int FileId;
typedef int FilePointer;
typedef int Length;

struct data {
  int length;
  char buffer[MAX];
};

struct writeargs {
  FileId f;
  FilePointer position;
  Data data;
};
```

```
struct readargs {
  FileId f;
  FilePointer position;
  Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;
    data READ(readargs)=2
  }=2;
}=9999;
```

Serveur Sun RPC

```
// Serveur

#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite.h"

void *write_2(writeargs *a) {
  /* supply implementation */
}

Data *read_2(readargs *a) {
  /* supply implementation */
}
```

Client Sun RPC

```
// Client

#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite.h"

main(int argc, char *argv[]) {
  CLIENT cl;
  struct data *d;
  struct readargs r;

  cl = clnt_create("server.polymtl.ca",FILEREADWRITE,VERSI(
  if(cl == NULL) {...}
  d = READ(&r,cl);
  if(d == NULL) {...}
}
```

CORBA

- Langage de définition d'interface avec constantes, structures, méthodes, et exceptions.
- Générateur de proxy et de squelette multi-langage.
- Possibilité d'appels asynchrones en l'absence de valeur de retour.
- Mécanisme pour la découverte et l'invocation dynamique de procedures.
- Divers services de notification...
- Semblable à Sun RPC mais plus sophistiqué et objet.

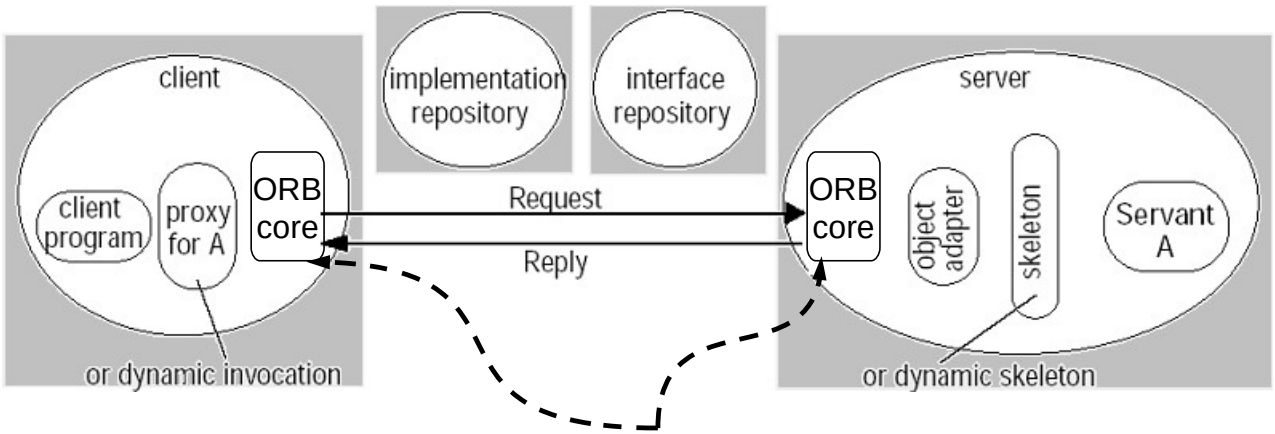
La norme CORBA

- Common Object Request Broker Architecture.
- L'OMG (Object management group) maintient la norme CORBA.
- C'est une architecture et infrastructure ouverte, indépendante du constructeur.
- CORBA n'est pas un Système Distribué mais sa spécification:
  - spécifications principale de plus de 700 pages.
  - Plus 1200 pages pour spécifier les services naviguant autour
  - 1991: v1: standardisation de IDL et OMA
  - 1996: v2: spécifications plus robustes, IIOP, POA, DynAny.
  - 2002: v3: ajout du component model, QoS (asynchronisme, tolérance de panne, RT Corba- Corba temps réel).

Composantes de CORBA

- Object Request Broker (ORB)
  - Initialisation et finalisation.
  - Conversion de références aux objets.
  - Envoi de requêtes.
- Portable Object Adapter (POA)
  - Activation des objets.
  - Répartition des requêtes vers les objets via les squelettes.
  - Gestion des adresses réseau.
  - Portable: fonctionne avec des implantations CORBA différentes.

Composantes de CORBA

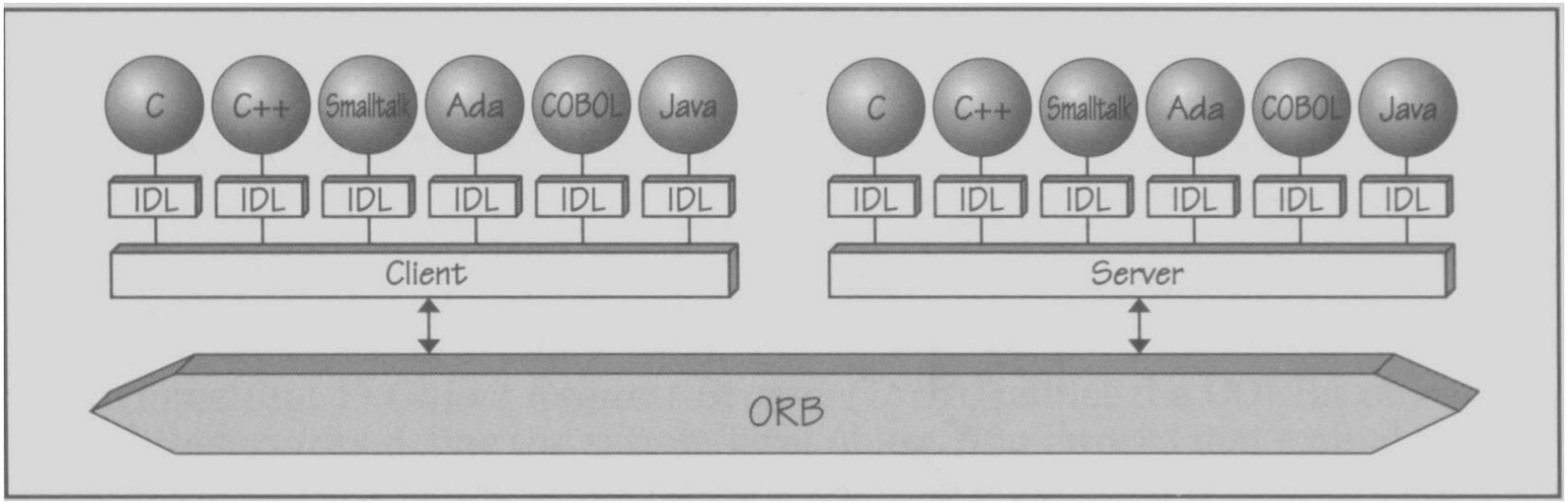


Fournit une interface incluant des opérations

Langage de définition d'interface

- Module: définit un espace de nom.
- Interface: liste de méthodes, attributs et déclarations de types ou exceptions. Héritage simple ou multiple.
- Méthode: nom de fonction, arguments in/out/inout, valeur de retour, exceptions. Attribut oneway pour invocation asynchrone.
- Types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet, any, struct, enum, union (tagged), array, string, sequence, object.
- Exception: peut contenir des variables.
- Attribut: variable de l'interface. Peut être readonly.

Le concept de CORBA



Référence à un objet

- IOR: Interoperable Object Reference.
- Format: identificateur du serveur, IIOP, hostname, port, nom de l'adapteur, nom de l'objet.
- IIOP: protocole particulier, interoperable et basé sur TCP/IP.
- IOR transitoire ou persistent.
- Plusieurs champs serveur/port sont possibles pour les services avec réplication.
- Un serveur peut répondre avec une redirection.

Support pour différents langages

- Chaque langage doit avoir son générateur idl et son interface de programation CORBA.
- En C, un argument est utilisé pour vérifier les exceptions.
- En Java les struct, enum, et unions sont implantés avec des classes.
- En C++ la relation entre les constructeurs/destructeurs et la gestion de la mémoire est subtile.

Services CORBA

- Noms: service hiérarchique (contexte, liste de (nom, contexte ou objet)). Part du contexte initial racine.
- Canal d'événement: fournisseurs et clients qui peuvent chacun être appelant ou appelé (push/pull). Plusieurs fournisseurs et clients peuvent se connecter au même canal. Un canal peut être un fournisseur, ou un client pour un autre.
- Notification: service enrichi pour un canal d'événements. Les clients peuvent spécifier les événements d'intérêt et interroger les types d'événements offerts. Les fournisseurs peuvent spécifier les types qu'ils offrent et savoir lesquels sont d'intérêt pour un ou plusieurs clients.
- Sécurité: authentification de l'envoyeur et de la réponse, journal, liste de contrôle des accès.
- Transactions: transactions et transactions imbriquées (begin/commit/abort).
- Persistence.

CORBA Common Data Representation (CDR)

- Permet de représenter tous les types de données qui peuvent être utilisées comme arguments ou valeurs de retour dans un appel à distance CORBA.
- Il existe 15 types primitifs: Short (16bit), long(32bit), unsigned short, unsigned long, float, char, ...
- CDR offre la possibilité d'avoir des *constructed types* qui sont des types construits à partir de types primitifs.
- Le type d'un item de données n'est pas donné avec sa représentation dans le message: l'émetteur et le récipiendaire du message connaissent l'ordre et le type des données du message.

Exemple du CDR

```
Struct Person {
        string  name;
        string  place;
        long    year;
};
```

index	contenu 4 octets
0-3	5
4-7	"Smit"
8-11	"h___"
12-15	6
16-19	"Lond"
20-23	"on__"
24-27	1934

SOAP

- Simple Object Activation Protocol.
- Requête HTTP, action POST, contenu XML (nom de la fonction et arguments d'entrée), réponse XML avec arguments de retour.
- Facile à déployer car réutilise toute l'infrastructure Web.
- Permet d'utiliser Web Services Security (WS-Security), specification qui définit l'implémentation des mesures de sécurité pour protéger un service web d'attaques externes.
  - Comment signer les messages SOAP pour en assurer l'intégrité et la non-répudiation
  - Comment chiffrer les messages SOAP pour en assurer la confidentialité
  - Comment attacher des jetons de sécurité pour garantir l'identité de l'émetteur
- Relativement inefficace.

Requête SOAP

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Réponse SOAP

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

RESTful API for Web Services

- Requête sans contexte (stateless), pour accéder une ressource représentée par un URL, en utilisant un certain format (e.g. JSON) et les méthodes HTTP (GET, PUT...).
- La méthode GET n'a aucun effet sur la ressource.
- Les méthodes PUT et DELETE sont idempotentes.
- Pas une norme, simplement un style architectural de service Web.
- De plus en plus populaire en raison de sa simplicité.
- Pas toujours bien utilisé (manques au niveau de la récupération d'erreurs, pas complètement sans contexte. . .).

Requête REST

- Requête

```
GET /api/StockPrice/IBM.json HTTP/1.1
Host: www.example.org
```

- Réponse

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: nnn

{ \CompanyName": \IBM",
  \Price": \34.5"
}
```

gRPC

- Acronyme pour **g**RPC **R**emote **P**rocedure **C**all.
- Rendu disponible par Google en 2015: "*A high performance, open-source universal RPC framework*" .
- Communication avec HTTP2 (compression des entêtes).
- Générateurs de code pour une dizaine de langages populaires (C++, Java, Python, Go, Ruby, C#...).
- Permet différents formats pour les messages mais implante initialement Protobuf (Protocol Buffers).
- Permet d'insérer le support pour l'équilibrage de charge, le traçage, le monitoring et l'authentification.
- Utilisé par Google, Netflix, Twitter, Cisco, Juniper...

Requête gRPC

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /google.pubsub.v2.Publisher
    Service/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer y235.
    wef315yfh138vh31hv93hv8h3v

DATA (flags = END_STREAM)
<Length-Prefixed Message>
```

```
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
content-type = application/grpc+proto

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc
```

Interface Protocol Buffers

- Semblable à SUN RPC, encodage binaire.

```
message Person {
  required string name = 1;
  optional string email = 3;
  enum PhoneType { MOBILE = 0; HOME = 1; WORK = 2; }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}

Message PersonInfo { int32 age = 1; int32 salary = 2; }

service Contact {
  rpc Check (Person) returns (PersonInfo) {}
}
```

Librairie d'accès aux objets Protocol Buffers

```
Person person;
person.set_name("John Doe");
person.set_email("jdoe@example.com");
person.SerializeToOstream(&output);

person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Protocol Buffers

- Format binaire compatible vers l'avant et l'arrière.
- Chaque champ peut être obligatoire ou optionnel et vient avec un numéro (tag) pour l'identifier.
- Un message est une séquence de: numéro de champ, type, valeur. Le type est un entier de 3 bits (varint, float, length delimited...). Le numéro de champ et le type sont groupés en un varint et prennent ensemble usuellement 1 octet.
- Les varint sont des entiers de longueur variable (0-127 ou 1b+0-127 +varint). Un premier bit à un indique qu'un octet supplémentaire est utilisé. Petit boutien.
- Fonctionne entre les langages et entre les plates-formes et est possible de traiter un message même s'il contient des champs inconnus.
- Protoc génère le code pour initialiser, sérialiser et lire les types décrits de même que pour faire des RPC (un type pour l'envoi et un pour la réponse).

Exemple de service de calcul de chemin

```
// route_guide.proto, (tiré de https://grpc.io/docs/tutorials/basic/c.html)

syntax = "proto3";
package routeguide;

service RouteGuide {
  rpc GetFeature(Point) returns (Feature) {}
  rpc ListFeatures(Rectangle) returns (stream Feature) {}
}
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}
message Rectangle {
  Point lo = 1;
  Point hi = 2;
}
message Feature {
  string name = 1;
  Point location = 2;
}
```

Fonctions de service de chemin

```
// route_guide_server.cc
...
class RouteGuideImpl final : public RouteGuide::Service {
 private:
  std::vector<Feature> feature_list_;
 public:
  explicit RouteGuideImpl(const std::string& db) {
    routeguide::ParseDb(db, &feature_list_);
  }
  Status GetFeature(ServerContext* context, const Point* point,
                    Feature* feature) override {
    feature->set_name(GetFeatureName(*point, feature_list_));
    feature->mutable_location()->CopyFrom(*point);
    return Status::OK;
  }
  Status ListFeatures(ServerContext* context, const routeguide::Rectangle*
      rectangle, ServerWriter<Feature>* writer) override {
    for (const Feature& f : feature_list_) {
      if (inside(f.location(), rectangle)) { writer->Write(f); }
    }
    return Status::OK;
} };
```

Activation du serveur de chemins

```
void RunServer(const std::string& db_path) {
  std::string server_address("0.0.0.0:50051");
  RouteGuideImpl service(db_path);

  ServerBuilder builder;
  builder.AddListeningPort(server_address,
      grpc::InsecureServerCredentials());
  builder.RegisterService(&service);
  std::unique_ptr<Server>
      server(builder.BuildAndStart());
  server->Wait();
}

int main(int argc, char** argv) {
  std::string db = routeguide::GetDbFileContent(argc, argv);
  RunServer(db);
  return 0;
}
```

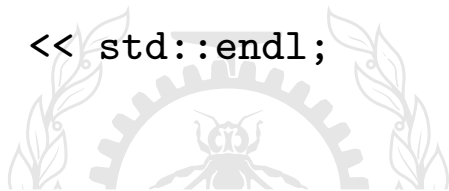
Client pour le service de chemins

```
class RouteGuideClient {
 public:
  RouteGuideClient(std::shared_ptr<Channel>
      channel, const std::string& db)
      : stub_(RouteGuide::NewStub(channel)) {}

  void ListFeatures() {
    routeguide::Rectangle rect;
    Feature feature;
    ClientContext context;

    rect.mutable_lo()->set_latitude(400000000);
    rect.mutable_lo()->set_longitude(-750000000);
    rect.mutable_hi()->set_latitude(420000000);
    rect.mutable_hi()->set_longitude(-730000000);

    std::unique_ptr<ClientReader<Feature> >
        reader(stub_->ListFeatures(&context, rect));
    while (reader->Read(&feature))
      std::cout << "Found feature called " << feature.name() << std::endl;
    Status status = reader->Finish();
  }
```



Client pour le service de chemins (suite)

```
    bool GetFeature(const Point& point, Feature* feature) {
      ClientContext context;

      Status status = stub_->GetFeature(&context, point, feature);
      if (!status.ok() || !feature->has_location() ||
          feature->name().empty()) return false;
      std::cout << "Found feature called " << feature->name() << std::endl;
      return true;
    }
  std::unique_ptr<RouteGuide::Stub> stub_;
  std::vector<Feature> feature_list_;
};

int main(int argc, char** argv) {
  RouteGuideClient guide(grpc::CreateChannel("localhost:50051",
      grpc::InsecureChannelCredentials()),db);
  Feature feature;

  guide.GetFeature(MakePoint(0,0), &feature);
  guide.ListFeatures();
  return 0;
}
```

Exemples d'utilisation des RPC

- Quelques services comme NFS sont basés sur les Sun RPC.
- Les principaux programmes dans le bureau GNOME (chiffrier, fureteur, éditeur, panneau...) offraient une interface CORBA mais se tournent maintenant vers D-BUS.
- CORBA est souvent utilisé par les compagnies de télécommunications pour leurs applications réparties de gestion de réseau.
- Java RMI est souvent utilisé dans des applications réparties internes.
- gRPC est utilisé à l'interne par Google et dans Kubernetes, ainsi que par quelques compagnies comme Netflix, Cisco, Morgan Stanley...

Communication dans les systèmes répartis

Autres mécanismes

- D-Bus : commun à GNOME et KDE, (en remplacement de CORBA et DCOP). Bus système (e.g., notification de batterie, réseau, clé USB. . .) et bus de session (e.g., sélection).
- WebSockets : communication duplex entre client et serveur avec entête HTTP.
- AMQP, ZeroMQ : solutions de remplacement plus performantes et plus légères pour la communication inter-processus et l'appel de procédures à distance (Request–reply, Publish–subscribe, Push–pull, Exclusive pair), avec option at-least-once, at-most-once, exactly-once. Utilisé pour le courtage, OpenStack...

ZeroMQ

- Alternative simplifiée et plus performante à AMPQ, sans agent de message (message broker). Semblable aux sockets mais de plus haut niveau.
- Plusieurs mécanismes de communication: intra-processus, inter-processus, TCP, PGM.
- Plusieurs patrons de communication:
  - Requête/réponse (request/reply): distribution de tâches, comme les appels de procédure à distance (RPC), ou les bus de service.
  - Publier/abonner (publish/subscribe): arbre de distribution de données.
  - Pousser/tirer (push/pull): distribution de tâches ou collecte de données, en parallèle.
  - Paire exclusive (exclusive pair): connexion point à point.

ZeroMQ: Serveur HW

```
void *context = zmq_ctx_new ();
void *responder = zmq_socket (context, ZMQ_REP);
int rc = zmq_bind (responder, "tcp://*:5555");
assert (rc == 0);

while (1) {
    char buffer [10];
    zmq_recv (responder, buffer, 10, 0);
    printf ("Received Hello\n");
    sleep (1);          //  Do some 'work'
    zmq_send (responder, "World", 5, 0);
}
```

ZeroMQ: Client HW

```
void *context = zmq_ctx_new ();
void *requester = zmq_socket (context, ZMQ_REQ);
// Pas besoin d'attendre que le serveur soit démarré
zmq_connect (requester, "tcp://localhost:5555");

int request_nbr;
for (request_nbr = 0; request_nbr != 10; request_nbr++) {
    char buffer [10];
    printf ("Sending Hello %d...\n", request_nbr);
    zmq_send (requester, "Hello", 5, 0);
    zmq_recv (requester, buffer, 10, 0);
    printf ("Received World %d\n", request_nbr);
}
zmq_close (requester);
zmq_ctx_destroy (context);
```

ZeroMQ: Serveur Météo

```
void *context = zmq_ctx_new ();
void *publisher = zmq_socket (context, ZMQ_PUB);
int rc = zmq_bind (publisher, "tcp://*:5556");

srandom ((unsigned) time (NULL));
while (1) {
    int zipcode = randof (100000);
    int temperature = randof (215) - 80;
    int relhumidity = randof (50) + 10;
    char update [20];

    sprintf (update, "%05d %d %d", zipcode, temperature, relhumidity);
    s_send (publisher, update);
}
zmq_close (publisher);
zmq_ctx_destroy (context);
```

ZeroMQ: Client météo

```
void *context = zmq_ctx_new ();
void *subscriber = zmq_socket (context, ZMQ_SUB);
int rc = zmq_connect (subscriber, "tcp://localhost:5556");

const char *filter = MyZipCode;
rc = zmq_setsockopt (subscriber, ZMQ_SUBSCRIBE,
        filter, strlen (filter));

int zipcode, temperature, relhumidity;
char *string;

while ((string = s_recv (subscriber)) != NULL) {
    sscanf (string, "%d %d %d",
          &zipcode, &temperature, &relhumidity);
    free (string);
}

zmq_close (subscriber);
zmq_ctx_destroy (context);
```

ZeroMQ: Distributeur de tâches

```
void *context = zmq_ctx_new ();
void *sender = zmq_socket (context, ZMQ_PUSH);
zmq_bind (sender, "tcp://*:5557");
srandom ((unsigned) time (NULL));

int total_msec = 0;
for (int task_nbr = 0; task_nbr < 100; task_nbr++) {
    int workload;
    workload = randof (100) + 1;
    total_msec += workload;
    char string [10];
    sprintf (string, "%d", workload);
    s_send (sender, string);
}
printf ("Total expected cost: %d msec\n", total_msec);
zmq_close (sink);
zmq_close (sender);
zmq_ctx_destroy (context);
```

ZeroMQ: Travailleurs de tâches

```
void *context = zmq_ctx_new ();
void *receiver = zmq_socket (context, ZMQ_PULL);
zmq_connect (receiver, "tcp://localhost:5557");
void *sender = zmq_socket (context, ZMQ_PUSH);
zmq_connect (sender, "tcp://localhost:5558");

while (1) {
    char *string = s_recv (receiver);
    printf ("%s.", string); fflush (stdout);
    s_sleep (atoi (string));
    free (string);
    s_send (sender, "");
}
zmq_close (receiver);
zmq_close (sender);
zmq_ctx_destroy (context);
```

ZeroMQ: Collecteur de résultats de tâches

```
void *context = zmq_ctx_new ();
void *receiver = zmq_socket (context, ZMQ_PULL);
zmq_bind (receiver, "tcp://*:5558");

int64_t start_time = s_clock ();

for (int task_nbr = 0; task_nbr < 100; task_nbr++) {
    char *string = s_recv (receiver);
    free (string);
}
printf ("Total elapsed time: %d msec\n",
      (int) (s_clock () - start_time));
zmq_close (receiver);
zmq_ctx_destroy (context);
```

ZeroMQ

- Fournit en arrière-plan des fils d'exécution et des queues, bloque l'envoi si la queue est pleine, attend pour la connexion, reconnecte au besoin...
- Les messages sont des séquences d'octets, un encodage comme protobuf peut être utilisé.
- Des patrons comme des proxy ou des routeurs sont directement supportés.
- Le code ne change pas selon le mécanisme de communication utilisé.
- Très utilisé dans de nombreuses compagnies.

Communication dans les systèmes répartis

Conclusion

- Les RPC sont un mécanisme intéressant pour formaliser les interfaces de manière à permettre la communication entre composantes possiblement distantes et dans un langage de programmation différent.
- Les Sun RPC sont encore assez répandus pour certains services.
- CORBA est depuis plusieurs années déployé à grande échelle mais des solutions plus simples et performantes sont populaires comme gRPC, AMQP et ZeroMQ.
- SOAP, REST, Ajax... sont très utilisés sur le Web.