Middleware and Distributed Systems
Messaging and Remote Procedures

Dr. Martin v. Löwis

Network Programming Models

- "Direct" network APIs: Expose all functionalities of the protocols; requires programmer to be aware of details of the respective protocol layer
  - e.g. TCP/IP and sockets: explicit establishment of connections, explicit transmission of byte stream
    - client: socket, connect, send, recv, select/poll, shutdown/close
    - server: socket, bind, listen, accept, send, recv, select/poll, shutdown/close
- message-style APIs: communication partners send messages (potentially structured byte sequences); middleware delivers to identified recipient
- RPC-style APIs: client invoked procedure that looks seemingly local; middleware transparently transfers invocation to remote node
- Other communication primitives, e.g. tuple spaces

Equivalence of Programming Paradigms

- Proposition: Any paradigm can be expressed in terms of any other paradigm.
- Example: building RPC on top of messaging
- Example: building messaging on top of RPC
- Example: building RPC on top of tuple spaces

Objective of Programming Models: Transparencies

- Language transparency
- Location transparency
- Service transparency
- Implementation transparency
- Architecture transparency
- Operating system transparency
- Protocol transparency
- Transport transparency

Functions of Messaging Middleware

- Basic functionality: applications may send messages (providing the message content in some form); other applications may wait for messages (blocking or non-blocking), then receive messages
- message queuing: messages are not directly sent to the receiver, but queued somewhere, decoupling sender and receiver in time
- unicast vs. multicast: a single message may have multiple receivers
- direct addressing: sender sends explicitly to receiver, or explicitly to queue
- indirect addressing: sender does not indicate receiver at all; receivers are determined implicitly
  - e.g. publish-subscribe communication
- Quality guarantees: timeliness of delivery, delivery guarantee
- filtering

Functions of RPC-style Middleware

- Basic functionality: a seemingly local procedure call is executed by a remote node; server procedure appears as local caller to server also
    - stubs (proxies) and skeletons
    - binding of client processes to server processes
- Object RPC: target of invocation is a method (including an object), not a plain procedure; server may offer multiple objects providing the same interface
    - extended functionality: distributed garbage collection
- exceptions: procedure may not terminate with a failure result
- object reference as data type: passing reference to objects across nodes
- quality guarantees: maybe semantics, at-least-once semantics, at-most-once semantics

Marshalling and MDE

- conversion of data into external representation suitable for transmission
  - typically byte sequence
- Model-driven engineering: generate marshalling code from abstract description
  - "abstract syntax" as opposed to "transfer syntax"
  - interface definition language

ONC RPC

- Open Network Computing
    - originally developed by Sun
    - RFC 1831: Remote Procedure Protocol Specification Version 2
    - RPC Language defined in RFC 1014
- IDL compiler: rpcgen
    - IDL files have typically .x extension
- Marshalling format: XDR (External Data Representation)
- Network protocol either TCP or UDP
    - Applications running on UDP must take unreliable nature into account

ONC RPC Language

- basic types int, unsigned int, enum, bool, hyper integer, unsigned hyper integer, float, double
- fixed-length uninterpreted data: opaque foo[n];
- variable-length uninterpreted data: opaque bar<n>; /* length optional */
- strings (ASCII?): string foobar<n>;
- fixed-length, variable-length arrays
- structs
- discriminated unions
    - also useful for optional data (equivalent to sequence of length 0/1)
- void, typedef, constant declarations

ONC RPC Language (cntd.)

- programs: named groups of versions, identified by program number

```
program NFS_PROGRAM {
    ...
} = 100003;
```

- version: named group of operations, identified by version number

```
version NFS_V3 {
    ...
} = 3;
```

- procedure: return type, name, parameters, operation number

```
LOOKUPres3 NFSPROC_LOOKUP(LOOKUP3args) = 3;
```

XDR

- All quantities encoded on multiples of 4 bytes
  - padding with 0 bytes if necessary (e.g. strings)
- integers encoded in network byte order
- fixed-length arrays: transmit just values
- variable-length sequences: transmit length, then contents
- structs: encode all fields
- unions: encode 4 bytes discriminant, then union value

Program Numbers

0 - 1fffffff   defined by rpc@sun.com
20000000 - 3fffffff   defined by user
40000000 - 5fffffff   transient
60000000 - 7fffffff   reserved
80000000 - 9fffffff   reserved
a0000000 - bfffffff   reserved
c0000000 - dfffffff   reserved
e0000000 - ffffffff   reserved

RPC Message Protocol

```
enum msg_type { CALL=0, REPLY = 1};
enum reply_stat { MSG_ACCEPTED = 0, MSG_DENIED = 1 };
enum accept_stat { ... }; ...
struct rpc_msg {
    unsigned int xid; /* transaction identifier */
    union switch(msg_type mtype){
        case CALL: call_body cbody;
        case REPLY: reply_body rbody;
    } body;
}
```

RPC Message Protocol (cntd.)

```
struct call_body {
    unsigned int rpcvers; /* (2) for RFC 1831 */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure-specific parameters start here */
};
union reply_body switch(reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED
        rejected_reply rreply;
};
```

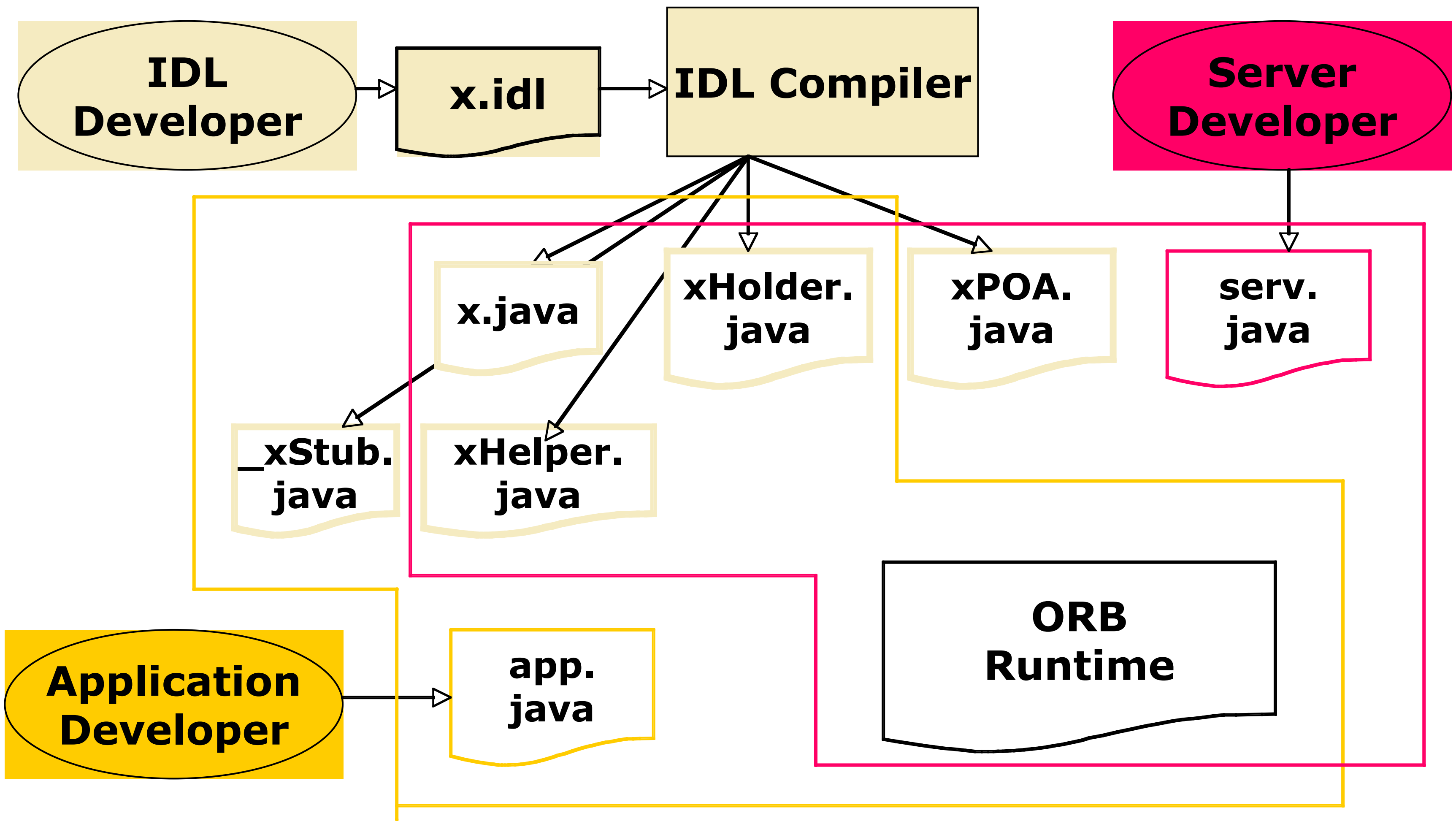
CORBA

- Common Object Request Broker Architecture
  - developed by OMG (Object Management Group)
- (OMG) IDL; compiler e.g. idlj (Sun JDK), fnidl (Fnorb)
- Marshalling Format: CDR (Common Data Representation)
- Wire Protocol: IIOP (Internet Inter-ORB Protocol)

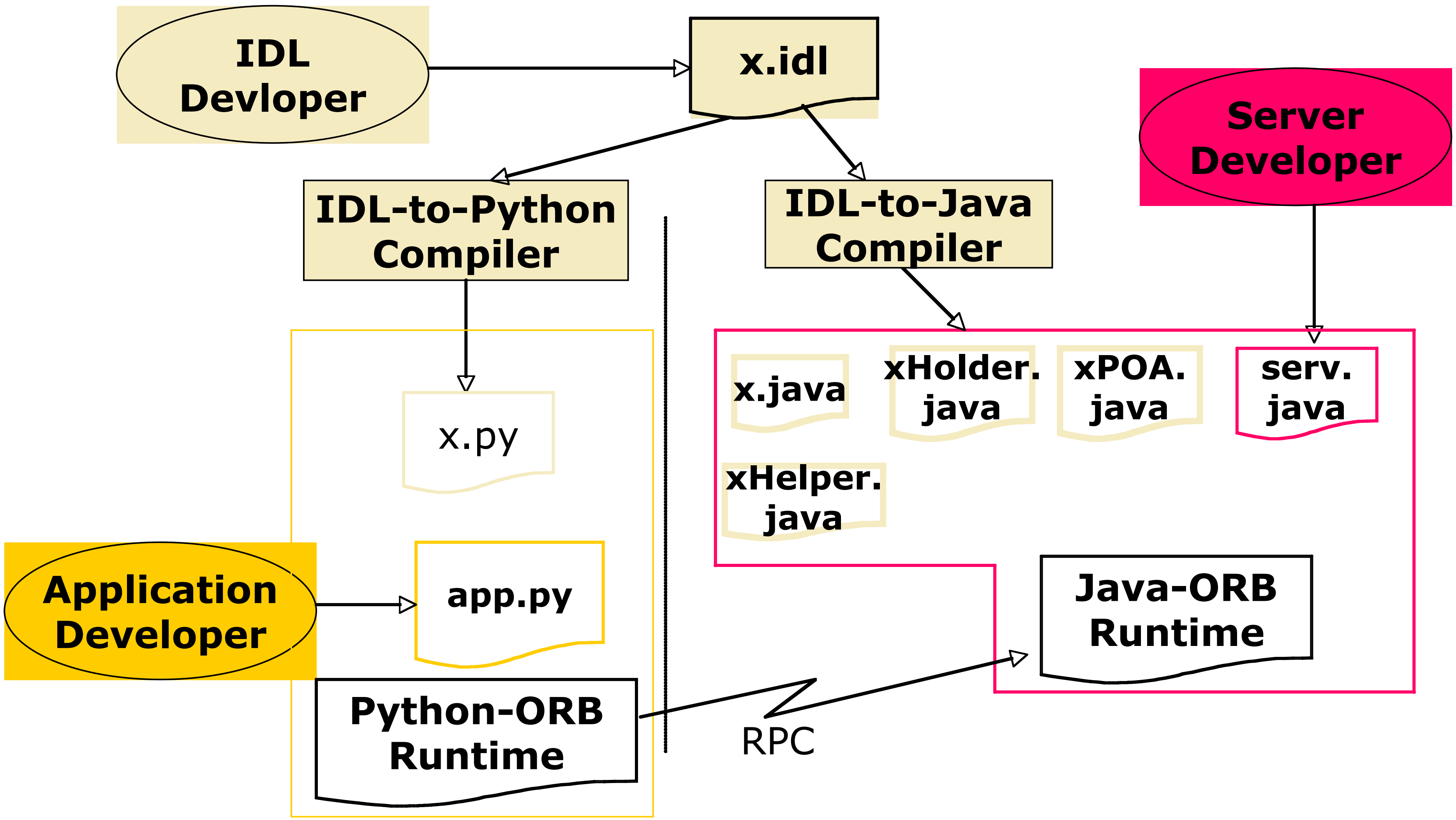
OMG IDL

- Interface Definition Language of CORBA (Common Object Request Broker ArchitectureIDL specifications separate language-independent interfaces from language-dependent implementations.
- IDL defines an <u>interface contract</u> between client and server.
- Language-independent IDL specifications are translated with an <u>IDL compiler</u> into APIs of the programming language.
- IDL is <u>purely</u> declarative (no actions, no statements about object state).
- IDL declarations are similar to Java interface definitions and to abstract classes.
- Data exchange between client and server is limited to the data types declared in IDL.

IDL Translation (Java)



IDL Translation (Multiple Languages)



IDL Source Files

- IDL files must end in .idl
- IDL is free-format: Line breaks and white space have no significance
  - Except for preprocessor
- Sources are processed by the preprocessor (#include, #define, …)
- Order of declarations is irrelevant
  - Definition must occur before use
  - Forward declarations

Comments and Keywords

- IDL supports C and C++ style comments

```
/*
 * A C comment
 */
// A C++ comment
```

- IDL keywords are all lower-case (e.g. interface), except for TRUE, FALSE, Object, and ValueBase.

Identifiers

- IDL identifiers can contain letters (A-Za-z), digits, and the underscore, e.g.

  Thermometer, nominal_temp

- IDL identifiers start with a letter. A leading underscore is allowed and ignored: **set_temp** and **_set_temp** are the same.

- Identifiers are case insensitive; **max** and **MAX** are the same identifier. The spelling must be consistent.

- Identifiers which are keywords in programming languages should be avoided (e.g. **class**, **package**, **template**).

-

Builtin Types

Type	Size	Value Range
short	>= 16 bits	$-2^{15} \dots 2^{15}-1$
unsigned short	>= 16 bits	$0 \dots 2^{16}-1$
long	>= 32 bits	$-2^{31} \dots 2^{31}-1$
unsigned long	>= 32 bits	$0 \dots 2^{32}-1$
long long	>= 64 bits	$-2^{63} \dots 2^{63}-1$
unsigned long long	>= 64 bits	$0 \dots 2^{64}-1$
float	>= 32 bits	IEEE single precision
double	>= 64 bits	IEEE double precision
long double	>= 79 bits	IEEE extended precision

Builtin Types (2)

- CORBA 2.1 adds fixed-point types:

```
typedef fixed<9,2> Total;            // up to 9 999 999,99
                                     // Precision 0,01
typedef fixed<9,4> InterestRate;     // up to 99 999,9999
                                     // Precision 0,0001
typedef fixed<31,0> BigInt;          // up to 10^31-1
```

- Fixed-point types have up to 31 digits
- No rounding effects in decimal system
- Computations use 62 digits

Builtin Types (3)

- IDL has two character types, **char** and **wchar**.
- **char** is an 8-bit type, **wchar** is wider (2 to 6 bytes).
- The standard code for **char** is ISO Latin-1; for **wchar**, it is 16-bit Unicode.
- Accordingly, there are two string types, **string** and **wstring**.
- Strings may contain arbitrary characters except for NUL.
- Strings can be limited in size

```
typedef string     City;
typedef string<3>  Abbreviation;
typedef wstring    Stadt;
typedef wstring<3> Abkuerzung;
```

Builtin Types (4)

- The IDL type **octet** is an 8-bit type which is transmitted without change.
  It is used to transmit binary data.
- **boolean** is a type with the values **TRUE** and **FALSE**.
- **any** is a universal type:
  - A value of type **any** can carry arbitrary values of other types, e.g. **boolean**, **double**, or user-defined types.
  - Values inside the **any** are type safe: extracting a value as a different type is not allowed.
  - any provides introspection: Given an any value, the type of the encapsulated value can be obtained.

Type Definitions

Using a **typedef**, new names for an existing type can be introduced:

typedef short YearType;

typedef short TempType;

typedef TempType TemperatureType

- Every type should have an application-specific name; these names should then be used consistently.
- Skilled use of typedefs increases readability.
- Unnecessary type aliases should be avoided; they are confusing and cause incompatibilities in some languages.

Enumeration Types

IDL allows the definition of enumerations:

enum Color {red, green, blue, black, mauve, orange};

- Color is a type of its own; no further typedef is needed.
- The type name must be provided.
- The enumerators are in the enclosing namespace and must be unique there:

  enum InteriorColor {white, beige, grey};

  enum ExteriorColor {yellow, beige, grey}; //Error!

- One cannot assign ordinals to enumerators:

  enum Wrong{ red = 0, blue = 8};

Structures

Structures are types with fields of arbitrary other types (including other user-defined types, excluding recursive types)

```
struct TimeOfDay{
    short hour;     // 0 – 23
    short minute;  // 0 – 59
    short second;  // 0 – 59
};
```

- A structure must contain atleast one field.
- The structure name must be provided.
- Member names must be unique within the structure.
- Structures form namespaces.
- Typedefs for structures should be avoided.

Unions

IDL supports “discriminated unions” with arbitrary fields

```
union ColorCount switch Color{
case red:
case green:
case blue:
   unsigned long    num_in_stock;
case black:
   float            discount;
default:
   string           order_details;
};
```

Unions (2)

- A union must have at least one field.
- The type name must be provided.
- Unions form namespaces with unique member names.
- Union Usage Guidelines:
  - char should not be used as the discriminator type.
  - Unions should not be used to model “type casts”.
  - There should be only one union field per case label.
  - The default branch should not be used.
  - Unions should be used sparingly.

Arrays

IDL supports one- and multi-dimensional array with arbitrary element type.

typedef Color ColorVector[10];

typedef string IdTable[10][20];

- Using a typedef here is mandatory;

  Color ColorVector[10];

  is ill-formed

- All dimensions must be provided;

  typedef string OpenTable[][20];

  is also ill-formed.

- Exercise caution when passing array indices across address spaces!

Sequences

Sequences are vectors of variable length.

- Sequences can be bounded or unbounded.

typedef sequence<Color> Colors;

typedef sequence<long, 100> Numbers;

- The bound must be a positive integral number.
- Sequences must be defined in a typedef.
- The element type can be an arbitrary type (including a recursive type)

typedef sequence<Node> ListOfNodes;

typedef sequence<ListOfNodes> TreeOfNodes;

- Sequences can be empty.

Arrays or sequences?

Arrays and sequences are similar, hence a few recommendations:

- If you have a fixed-size list of values, and all values are always present, use an array.
- If you have a variably-sized set of things, use a sequence.
- Use character arrays for strings of fixed size.
- Use sequences for sparse matrices (with (i, j, value) triples)

Other IDL Type Concepts

- recursive types (sequences inside structs)
- valuetypes
- constants

Interfaces

Interfaces (Schnittstellen) define object types:

interface Thermometer{
    string get_location();
    void set_location(**in string** loc);
};

- Invocation of an operation for an instance sends an RPC call to the server implementing the instance
- Interfaces define the “public” Interface. There are no private/protected parts.
- Interfaces have no data members.
- Interfaces define the smallest and only granularity of distribution (*unit of distribution*). Everything remotely accessible has an interface.

Interface Syntax

- Interface definitions may include exceptions, attributes, operations and type definitions

```
interface Haystack{
    exception NotFound{ unsigned long num_straws_searched;};
    const unsigned long MAX_SIZE = 1000000;
    readonly attribute unsigned long num_straws;
    typedef long Needle;
    typedef string Straw;
    void add(in Straw s);
    boolean remove(in Straw s);
    boolean find(in Needle n) raises(NotFound);
};
```

Interface Semantics

Interfaces are types and can be used as parameters

```
interface FeedShed{
   void        add(in Haystack s);
   void        eat(in Haystack s);
};
```

- Parameters of type Haystack are object reference parameters.
- Passing of objects always happens by reference.
- The object remains in its original location, only the reference is passed.
- Usage of a reference again happens by RPC calls.
- Nil reference: no object

Syntax of Operations

Every operation definition contains

- An operation name
- A return type (possibly **void**)
- Zero or more parameters (parameter directionality: **in**, **out**, or **inout**)

Optionally, a operation definition contains

- A **raises** declaration
- A **oneway** attribute
- A **context** clause

IDL provides no operation overloading; operation names must be unique within their interface.

Exceptions

- User-defined exceptions and system exceptions
- User-defined exceptions must be declared, and can carry parameters
- System exceptions are predefined (39 system exceptions), and carry
  - completion status (COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE)
  - minor exception code

oneway Operations

Operations can be defined as oneway:

interface Events{

  oneway void send(**in** EventData data);

};

- Semantic restrictions for oneway operations:
  - The return type must be void.
  - They must not have out or inout parameters.
  - There must be no raise clause.
- Oneway operations provide best-effort send-and-forget semantics.
- Oneway calls can be lost and can be delivered synchronously or asynchronously.

  In CORBA 2.3, oneway is not fully portable (2.4: async messaging spec)

Attributes

Interfaces may contain attributes

```
interface Thermostat{
    readonly  attribute short temperature;
              attribute short nominal_temp;
};
```

- Attributes imply an <u>operation pair</u>: a read operation and a write operation (readonly: no write operation)
- Attributes defined neither state nor members, they are merely a shorthand notation
- Attributes must not throw exceptions (up to CORBA 2.4) and must not be oneway
- Attributes should be read-only

Further Constructs

- Modules: namespaces, similar to C++ namespaces
  - Modules help avoiding naming conflicts
  - Modules can contain nearly arbitrary IDL constructs (interfaces, type definitions, modules, constants, ...)
  - Modules can be extended (module reopening)
- Valuetypes: "objects by value"
- Inheritance: interfaces and valuetypes can inherit from others
  - multiple inheritance for interfaces, restricted inheritance for valuetypes
  - polymorphism: instead of passing a reference to a base interface object, a derived interface reference may be passed

GIOP

- General Inter-ORB Protocol
- Prerequisites:
    - connection-oriented transport
    - full-duplex connections
    - connection is symmetric w.r.t. shutdown
    - transport is reliable
    - transport transmits byte streams
    - transport informs about connection loss (disorderly release)

GIOP (cntd.)

- GIOP defines
  - Message format: Common Data Representation (CDR)
  - Message types
  - Structure of object references: Interoperable Object Reference (IOR)
- GIOP message format and IOR format are defined in IDL
  - will be transmitted through CDR
- IIOP (Internet Inter-ORB Protocol)
  - Transport is TCP
  - IOR format is specialized (IOR profile)

CDR

- Common Data Representation
- Bi-endian: endianness is typically "native" for sender (JDK: always bigendian)
- Data encoded as sequence of primitive values
  - structures, parameter boundaries are not represented
- Each primitive type has fixed size:
  - char, (wchar), octet, boolean: 1
  - short, unsigned short: 2
  - long, unsigned long, float, enum: 4
  - long long, unsigned long long, double: 8
  - long double: 16

CDR (cntd.)

- Alignment: Each primitive value is aligned relative to the message start
  - usually a multiple of its size
  - long double: 8
  - Idea: allow direct copying from transport buffer into C structures

Encoding of Primitive Types

- integral types: binary, signed types in two's complement
- Floating-point types: IEEE-754
- octet: "as-is"
- boolean: TRUE=1, FALSE=0
- characters: according to "character set negotiation"

Encoding of Complex Types

- Strings: 4-byte length, then value, null-terminated
- structs: element for element, including padding if necessary
- unions: discriminator, then union branch
- arrays: element for element
  - multi-dimensional arrays: last index grows fastest
- sequence: 4 byte length, then values
- enum: like unsigned int, enumerators start at 0
- any: typecode, value
- exception: string (repository ID), struct(exception members)

Interoperable Object References (IOR)

```
module IOP {
 typedef unsigned long ProfileId;
 struct TaggedProfile {
     ProfileId tag;
     sequence <octet> profile_data;
 };
 struct IOR {
     string type_id;
     sequence <TaggedProfile> profiles;
 };
}
```

- IOR-String: hexified version of an encapsulation containing an IOP::IOR
  e.g. IOR:000000000000002849444C3A6F6D672E6F72672F436F734E616D696E672F4
  E616D696E67436F6E746578743A312E300000000001000000000000000640001020
  00000001B6466772E64636C2E6870692E756E692D706F747364616D2E64650000
  930600000000000B4E616D65536572766963650000000001000000010000002000
  00000000010020000000010001000100010109000000020501000100010100

IOR Profiles

- Define protocol independent contact information
- Defined in IDL
- encoded as an encapsulation

```
module IOP {
 const ProfileId TAG_INTERNET_IOP = 0;
 const ProfileId TAG_MULTIPLE_COMPONENTS = 1;
 const ProfileId TAG_SCCP_IOP = 2;
};
```

IIOP Profile

```
module IIOP {
 struct Version {
     octet major;
     octet minor;
 };
 struct ProfileBody_1_1 {// also used for 1.2
     Version iiop_version;
     string host;
     unsigned short port;
     sequence <octet> object_key;
     sequence <IOP::TaggedComponent> components;
 };
};
```

Tagged Components

- Define protocol-independent contact information
- represented as TAG_MULTIPLE_COMPONENTS or in the IIOP profile

```
module IOP{
   typedef unsigned long ComponentId;
   struct TaggedComponent {
      ComponentId tag;
      sequence <octet> component_data;
   };
   typedef sequence<TaggedComponent> TaggedComponentSeq;
};
```

Standard IOR Components

```
module IOP {
  const ComponentId TAG_ORB_TYPE = 0;
  const ComponentId TAG_CODE_SETS = 1;
  const ComponentId TAG_POLICIES = 2;
  const ComponentId TAG_ALTERNATE_IIOP_ADDRESS = 3;
  const ComponentId TAG_ASSOCIATION_OPTIONS = 13;
  const ComponentId TAG_SEC_NAME = 14;
  const ComponentId TAG_SPKM_1_SEC_MECH = 15;
  const ComponentId TAG_SPKM_2_SEC_MECH = 16;
  const ComponentId TAG_KerberosV5_SEC_MECH = 17;
  const ComponentId TAG_CSI_ECMA_Secret_SEC_MECH = 18;
  const ComponentId TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
  const ComponentId TAG_SSL_SEC_TRANS = 20;
  const ComponentId TAG_JAVA_CODEBASE = 25;
  ...
```

GIOP Messages

- Protocol assumes connection between client and server
- Connection management is invisible to the application (implicit binding)
- Different protocol versions:
    - GIOP 1.0 (CORBA 2.0)
    - GIOP 1.1 (CORBA 2.1): Fragmentation
    - GIOP 1.2 (CORBA 2.3): bidirectional communication
- downwards compatible: old clients can talk to new servers

Message Types

Message Type	Initiator	Value	GIOP Version
Request	Client	0	1.0, 1.1, 1.2
Reply	Server	1	1.0, 1.1, 1.2
CancelRequest	Client	2	1.0, 1.1, 1.2
LocateRequest	Client	3	1.0, 1.1, 1.2
LocateReply	Server	4	1.0, 1.1, 1.2
CloseConnection	Server	5	1.0, 1.1, 1.2
MessageError	beide	6	1.0, 1.1, 1.2
Fragment	beide	7	1.1, 1.2

Structure of a GIOP Message

- Basic Structure:
  - GIOP message header
  - message-specific header
  - message-specific body

```
module GIOP {
 struct Version {octet major; octet minor;    };
 enum MsgType_1_1 { Request, Reply, CancelRequest, ... };

 struct MessageHeader_1_1 {
     char magic [4]; // GIOP
     Version GIOP_version;
     octet flags;  // Bit 0: Endianness (0: big)
                   // Bit 1: more fragments
     octet message_type;
     unsigned long message_size;
};
```

Request

```
struct RequestHeader_1_1 {
 IOP::ServiceContextList service_context;
 unsigned long request_id;
 boolean response_expected;
 octet reserved[3];
 sequence <octet> object_key;
 string operation;
 CORBA::OctetSeq requesting_principal;
};
```

- followed by parameters (GIOP 1.2: 8-aligned)

Service Context

- Additional Information transmitted from ORB to ORB

```
module IOP {
 typedef unsigned long ServiceId;
 struct ServiceContext {
     ServiceId context_id;
     sequence <octet> context_data;
 };
 typedef sequence <ServiceContext>ServiceContextList;
 const ServiceId TransactionService = 0;
 const ServiceId CodeSets = 1;
 const ServiceId ChainBypassCheck = 2;
 const ServiceId ChainBypassInfo = 3;
 const ServiceId LogicalThreadId = 4;
 const ServiceId BI_DIR_IIOP = 5;
 ...
```

Reply

```
enum ReplyStatusType_1_0 {
 NO_EXCEPTION,
 USER_EXCEPTION,
 SYSTEM_EXCEPTION,
 LOCATION_FORWARD
};
struct ReplyHeader_1_0 {
 IOP::ServiceContextList service_context;
 unsigned long request_id;
 ReplyStatusType_1_0 reply_status;
};
```

ASN.1

- Abstract Syntax Notation 1
- ITU-T Recommendation X.680, X.681, X.682, X.683, X.690, X.691, X.692, X.693
  - Simultaneously ISO/IEC 8824:1-4, 8825:1-4
- 68x: Notation (Basic, Information Objects, Constraints, Parametrization)
- 69x: Encoding Rules (Basic (BER), Canonical (CER), Distinguished (DER), Packed (PER), XML (XER), Encoding Control Notation (ECN))
- Example applications in telco domain: ISDN, GSM, ,
- Example applications in the internet: SNMP, LDAP, PKI (X.509)

Notation

- Primitive types (INTEGER, BOOLEAN, IA5String, OBJECT IDENTIFIER, ...)
- Type constructors (SEQUENCE, SET, SEQUENCE OF, SET OF, CHOICE)
  - fields can be tagged, declared as OPTIONAL, or with a DEFAULT value
- Standard only implies encodings, not language mappings
  - various tools implement language mappings
- Modules: Grouping of type definitions belonging together
- No inherent binding to transport protocols or messaging protocols
  - originally designed for OSI protocols
  - original RPC protocol: X.880 ROSE (Remote Operation Service Element)

Object Identifiers

- Allocated globally unique identification of "objects"
  - data types, algorithms, organizations, physical entities, ...
- Hierarchical tree, with arcs identified by numbers
  - Root: ccitt(0), iso(1), or joined-iso-ccitt(2)
  - Notations: arc.arc.arc... or { name(number) name(number) ... }
- Examples:
  - {joint-iso-itu-t(2) asn1(1) basic-encoding(1)} aka 2.1.1
  - {iso(1) member-body(2) de(276)} aka 1.2.276
  - {iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) microsoft(311)} aka 1.3.6.1.4.1.311
    - 1.3.6.1.4.1.311.10.3.4: Certificate is usable for encrypted file systems

Basic Encoding Rules

- Tag-Length-Value (TLV) encoding
  - each of tag, length, and value may be variable number of octets
- Tag encoding: four kinds of tags
  - Universal: data type is predefined in ASN.1
  - Application: data type is defined for ASN.1 module, using [APPLICATION n]
  - Context: Tag is only valid within the specific data structure, specified as [n]
  - Private: data is not defined in ASN.1
- Tag byte 1: kkcvvvvv
  - kk: kind (universal: 00, application: 01, context: 10, private: 11)
  - c = 0: data is of primitive type; c = 1: constructed type

Universal Tags

Typ	Tag	Typ	Tag	Typ	Tag
0	EOC				
1	BOOLEAN	11	EMBEDDED PDV	21	VideotexString
2	INTEGER	12	UTF8String	22	IA5String
3	BIT STRING	13	RELATIVE-OID	23	UTCTime
4	OCTET STRING	14		24	GeneralizedTime
5	NULL	15		25	GraphicString
6	OBJECT IDENTIFIER	16	SEQUENCE(OF)	26	VisibleString
7	Object Descriptor	17	SET(OF)	27	GeneralString
8	EXTERNAL	18	NumericString	28	UniversalString
9	REAL	19	PrintableString	29	CHARACTER STRING
10	ENUMERATED	20	T61String	30	BMPString

Basic Encoding Rules (cntd.)

- Tag encoding: tag 0..30: 1 Byte
    - tag 31: Multi-byte, terminating byte has MSB=0
- Length encoding:
    - short form (0..127): 1 Byte
    - 0x80: indefinite length encoding (until EOC)
    - 1xxxxxxx: length of length, followed by length bytes
- value encoding: depending on data type

DER and CER

- BER is ambiguous
  - length encoding: short form, long form (leading zeroes?), indefinite form
  - integers: leading zero bytes?
  - SET, SET OF: order of elements?
- cryptographic uses require 1:1 relationship between data and encoding
  - DER and CER subset BER, adding requirements
- must always use minimum number of bytes to represent values
- unused bits in bitstring must be zero, TRUE has all bits set
- fields with default value must be omitted
- DER: use always definite length; CER: use always indefinite length

PER

- Tags normally omitted
- optional fields not identified by tag, but with a bitmap at the beginning of the SEQUENCE
- CHOICE alternative not identified by tag, but by number
- Length omitted except for truly variable-sized values (SEQUENCE, strings)
- integers may consume less than 1 byte if the range allows it (BOOLEAN: 1 bit)
  - fields don't necessarily start at byte boundary
  - two forms: aligned PER and unaligned PER

Encoding Control Notation

- Try to represent "legacy" protocols in ASN.1
- Customize encoding of data types for specific applications
- declares formulae to compute bit strings out of ASN.1 abstract values
  - maps types recursively to bit strings, then concatenates them
- attempt to generalize protocol across all data types of a protocol
  - may need to hand-craft encoding of specific structures