Middleware

Petr Tůma

Middleware

Petr Tůma

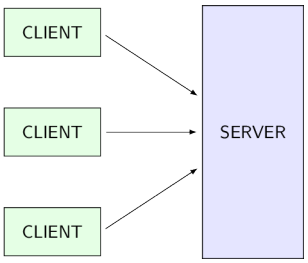
This is version 0d4922f166c7b5b2739aff2e10953b52d2ba75e8 (modified) generated on 2023-05-12 07:49:00. For the latest version, check http://d3s.mff.cuni.cz/teaching/nswi080.

Table of Contents

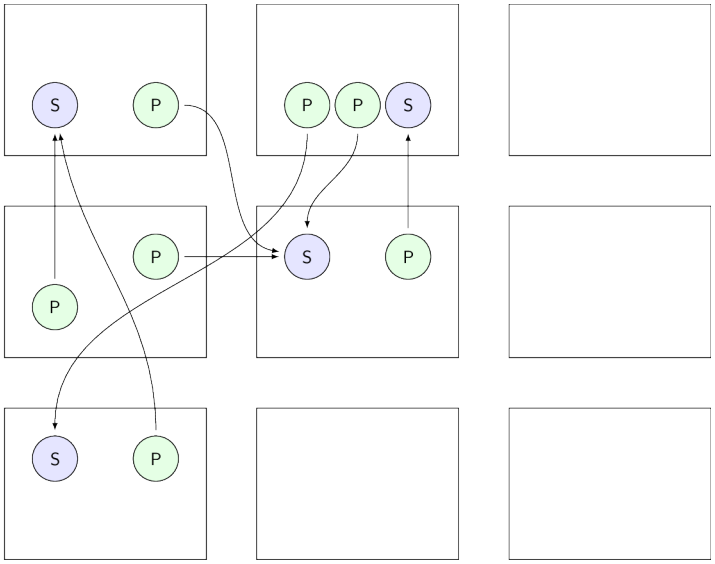
Chapter 1. Concepts

1.1. Architectures

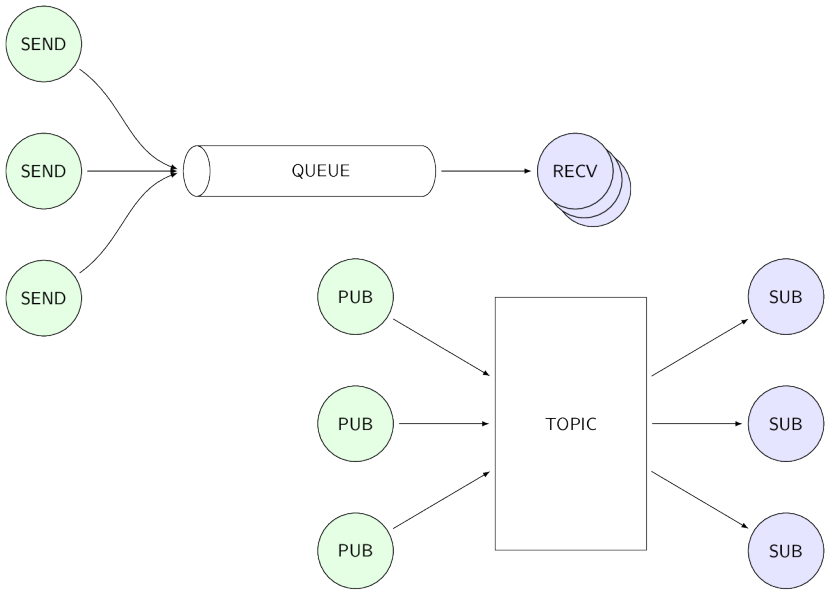
Client-Server



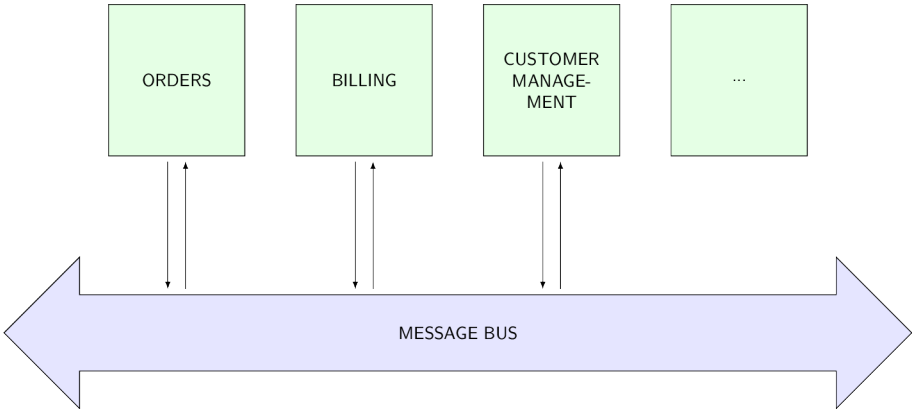
Distributed Objects



Messaging



Message Bus



1.2. Serialization

1.2.1. Textual

XML Object Serialization Example

Data Instance.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<anExampleDataClass>
    <anIntField>123</anIntField>
    <aFloatField>12.34</aFloatField>
    <aDoubleField>1.234E57</aDoubleField>
    <aBoxedIntField>987</aBoxedIntField>

    <aRequiredStringField>a string</aRequiredStringField>
    <anArrayWithoutAWrapper>1</anArrayWithoutAWrapper>
    <anArrayWithoutAWrapper>2</anArrayWithoutAWrapper>
    <anArrayWithoutAWrapper>3</anArrayWithoutAWrapper>

    <anArrayWithAWrapper>
        <anArrayElement>12</anArrayElement>
        <anArrayElement>34</anArrayElement>
        <anArrayElement>56</anArrayElement>
    </anArrayWithAWrapper>

    <aListElement>
        <anIntField>0</anIntField>
        <aFloatField>0.0</aFloatField>
        <aDoubleField>0.0</aDoubleField>
    </aListElement>

    <aSetElement>
        <anIntField>0</anIntField>
        <aFloatField>0.0</aFloatField>
        <aDoubleField>0.0</aDoubleField>
    </aSetElement>

    <aMapElement>
        <entry>
            <key>456</key>
            <value>
                <anIntField>0</anIntField>
                <aFloatField>0.0</aFloatField>
                <aDoubleField>0.0</aDoubleField>
            </value>
        </entry>
        <entry>
```

```
            <key>123</key>
            <value>
                <anIntField>0</anIntField>
                <aFloatField>0.0</aFloatField>
                <aDoubleField>0.0</aDoubleField>
            </value>
        </entry>
    </aMapElement>
</anExampleDataClass>
```

Possible Schema.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">

    <xs:element name="anExampleDataClass" type="anExampleDataClass"/>

    <xs:complexType name="anExampleDataClass">
        <xs:annotation>
            <xs:documentation>
                An example class.
                Contains various field types to illustrate the mapping.
            </xs:documentation>
        </xs:annotation>

        <xs:all>
            <xs:element name="anIntField" type="xs:int"/>
            <xs:element name="aFloatField" type="xs:float"/>
            <xs:element name="aDoubleField" type="xs:double"/>

            <xs:element minOccurs="0" name="aBoxedIntField" type="xs:int"/>
            <xs:element name="aRequiredStringField" type="xs:string"/>
            <xs:element minOccurs="0" name="anOptionalStringField" type="xs:string"/>
            <xs:element default="default" minOccurs="0" name="aStringFieldWithDefaultValue" type="x

            <xs:element maxOccurs="unbounded" minOccurs="0" name="anArrayWithoutAWrapper" type="xs:
            <xs:element minOccurs="0" name="anArrayWithAWrapper">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element maxOccurs="unbounded" minOccurs="0" name="anArrayElement" type=
                    </xs:sequence>
                </xs:complexType>
            </xs:element>

            <xs:element maxOccurs="unbounded" minOccurs="0" name="aListElement" type="anExampleData
            <xs:element maxOccurs="unbounded" minOccurs="0" name="aSetElement" type="anExampleDataC

            <xs:element name="aMapElement">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element maxOccurs="unbounded" minOccurs="0" name="entry">
                            <xs:complexType>
                                <xs:sequence>
                                    <xs:element minOccurs="0" name="key" type="xs:int"/>
                                    <xs:element minOccurs="0" name="value" type="anExampleDataClass
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:all>
    </xs:complexType>
</xs:schema>
```

1.2.1.1. References

1. Eclipse: JAXB Reference Implementation. https://github.com/eclipse-ee4j/jaxb-ri

JSON Object Serialization Example

Data Instance.

```
{
    "an_int_field" : 123,
    "a_float_field" : 12.34,
    "a_string_field" : "a string",
    "an_array" : [1, 2, 3]
}
```

YAML Object Serialization Example

Data Instance.

```
an_int_field: 123
a_float_field: 12.34
a_string_field: a string
an_array:
- 1
- 2
- 3
a_mapping_field:
    &some_name a_nested_field: a string
    a_reference: *some_name
...
```

Anchors and Aliases in YAML

Anchors and aliases in YAML are considered a serialization detail and are not generarally preserved in the representation graph. An anchor can appear at any position before the node content, an alias appears instead of the node content. The alias referes to the last anchor of that name, names can be reused.

Native Object Serialization with YAML Tags

To represent native types, YAML relies on the use of tags. YAML distinguishes local and global tags, local tags are simply application specific strings starting with an exclamation mark that can be attached to any node. For example, the following demonstrates the use of tags to distinguish Python tuples from Python lists, which would otherwise both end as the same array:

```
> import yaml
> print (yaml.dump ((1, 2, 3)))
!!python/tuple
- 1
- 2
- 3
> print (yaml.dump ([1, 2, 3]))
- 1
- 2
- 3
```

Only trusted code should be allowed to use all serialization tags:

```
> import yaml
> yaml.unsafe_load ('!!python/object/apply:os.system ["echo Hello from shell !"]')
Hello from shell !
0
```

See the list of serialization tags in the module documentation. The `!!python/object/apply:module.function` tag expands into the result of calling the module function.

Serialization Security Issues

General object serialization mechanisms such as Java serialization or Python pickling should only be used with trusted content. To handle arbitrary object types, such mechanisms can sometimes execute user code as a part of the serialization process, and such code can sometimes be tricked to execute arbitrary commands.

See https://github.com/frohoff/ysoserial for multiple examples of arbitrary code execution through Java serialization. The examples rely on Java serialization invoking `readObject` on the user type. In one of the examples, this type is `AnnotationInvocationHandler`, whose `readObject` method iterates over a collection making up its state. In turn, this collection can be an instance of `LazyMap`, an Apache Commons Collections class that invokes item factory when iterated upon. This factory can be another Apache Commons Collections class, an `InvokerTransformer`, which can invoke arbitrary methods, such as the runtime `exec` method. See https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections1.java for this particular payload example.

1.2.1.2. References

1. Colm O'Connor: The Norway Problem. https://hitchdev.com/strictyaml/why/implicit-typing-removed
2. Chris Frohoff: The ysoserial Project Repository. https://github.com/frohoff/ysoserial
3. Moritz Bechler: The marshalsec Project Repository. https://github.com/mbechler/marshalsec

1.2.2. Binary

1.2.2.1. Concise Binary Object Representation (CBOR)

The CBOR format stores basic types, arrays of basic types, and maps of basic types. Basic types are null, booleans, integers, floats, byte and text strings. An item can be wrapped in a tag that specifies additional information, which can identify date and time, big num, URI and so on. References are not supported.

The CBOR data stream is a sequence of items. Each item starts with a single byte header that carries the item type (3 bits) and additional argument value (5 bits). The rest of the item data depends on the type and the value.

CBOR Serialization Examples

Integer Data Items.

```
00h ~ 000-00000b ~ positive integer type (0) value 0
01h ~ 000-00001b ~ positive integer type (0) value 1
17h ~ 000-10111b ~ positive integer type (0) value 23

18h ~ 000-11000b ~ positive integer type (0) value in next byte (24)
18h              ~ value 24 (18h)
18h ~ 000-11000b ~ positive integer type (0) value in next byte (24)
19h              ~ value 25 (19h)

19h ~ 000-11001b ~ positive integer type (0) value in next two bytes (25)
01h 00h          ~ value 256 (network order)

1Ah ~ 000-11010b ~ positive integer type (0) value in next four bytes (26)
00h 01h 00h 00h  ~ value 65536 (network order)

20h ~ 001-00000b ~ negative integer type (1) value -1
21h ~ 001-00001b ~ negative integer type (1) value -2

38h ~ 001-11000b ~ negative integer type (1) value in next byte (24)
```

```
FFh             ~ value -256
```

CBOR Playground

See https://cbor.me for a CBOR playground that can convert between textual and binary representations. Apart from experimenting with basic types of various sizes, these are some other values with interesting serialization:

- `0("2020-01-01T00:00Z")` for a string that contains date and time
- `18446744073709551616` for the first integer big enough to use the bignum encoding
- `4([-1, 1])` for value `0.1` encoded as decimal fraction
- `5([-1, 1])` for value `1/2` encoded as binary fraction
- `[_ [1, 2], [3, 4, 5]]` for an indefinite length array

1.2.2.1.1. References

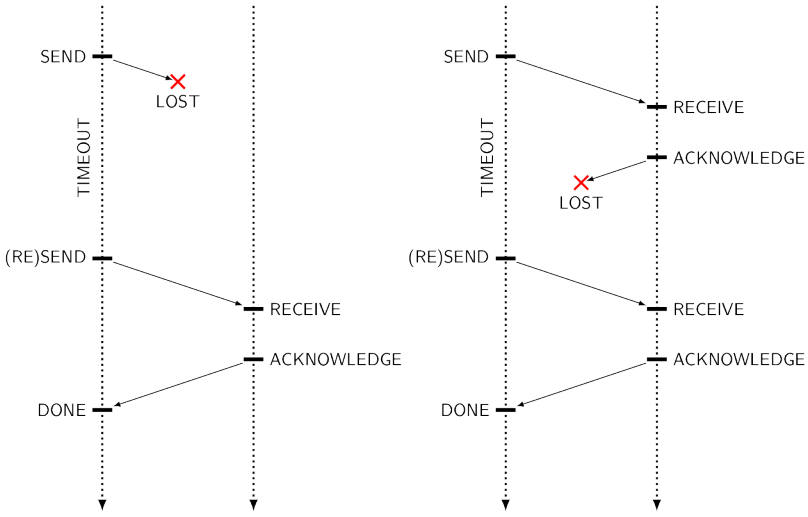
1. Carsten Bormann: Concise Binary Object Representation Website. https://cbor.io
2. IETF: Concise Binary Object Representation (CBOR) (RFC 8949). https://tools.ietf.org/html/rfc8949
3. IETF: Concise Data Definition Language (CDDL) (RFC 8610). https://tools.ietf.org/html/rfc8610

1.3. Protocols

1.3.1. Reliability

One standard way of ensuring reliable delivery is introducing acknowledgments. Note how the loss of a message and the loss of an acknowledgment cannot be distinguished by the sender. That is why the messages must possess identities, and why amnesia failures, where the record of received message identities is lost, can cause repeated message delivery.

Message Acknowledgment

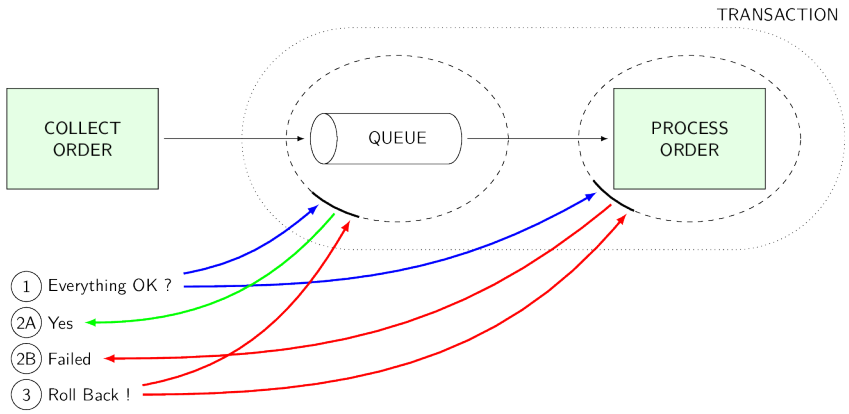


Another way of increasing delivery reliability is forward error correction, where redundant information is transmitted in order to increase the probability of receiving enough information to reconstruct the original message.

1.3.2. Atomicity

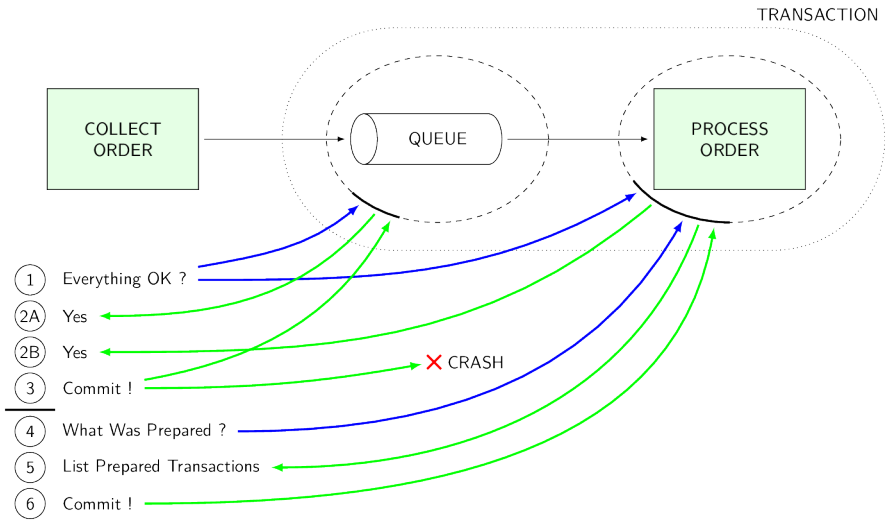
Two phase commit protocol ensures atomic completion of transactions. If any transaction participant fails during transaction, a rollback command is issued to all participants.

Transactional Messaging Rollback



If any transaction participant fails after commit has been decided, it must recover and proceed as directed by the coordinator.

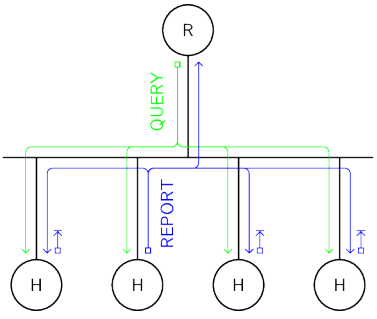
Transactional Messaging Commit



1.3.3. Multicast Membership

In the multicast listener discovery protocol, the router responsible for a link periodically multicasts General Multicast Listener Queries, inviting nodes to report what multicast addresses they listen to. The nodes respond with Multicast Listener Reports, which are multicast after random delay. If a node observes a report with the same multicast address within the random delay, it drops its own report.

Group Membership Query

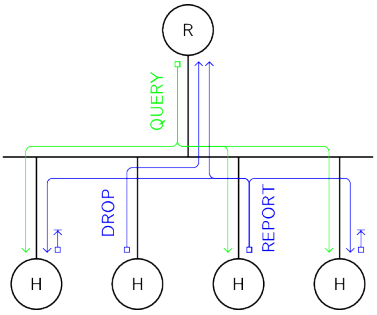


Protocol.

QUERY	General Multicast Listener Query periodically multicast by router
REPORT	Multicast Listener Report multicast after random delay

A node that stops listening to a multicast address sends a Multicast Listener Drop message to the router. In response, the router multicasts Specific Multicast Listener Query to inquire about remaining listeners.

Group Membership Done



Protocol.

DROP	Multicast Listener Drop sent from host to routers
QUERY	Specific Multicast Listener Query multicast by router
REPORT	Multicast Listener Report multicast after random delay

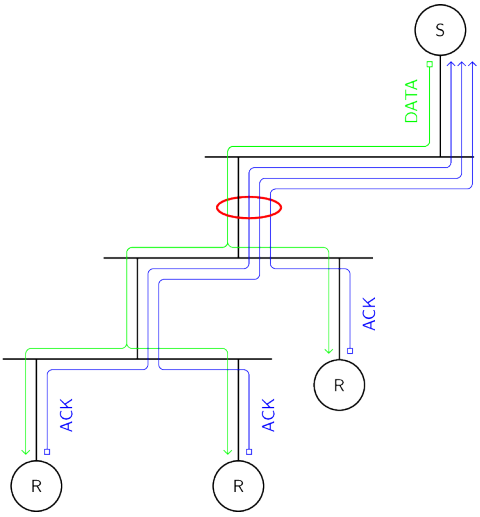
1.3.3.1. References

1. IETF: Multicast Listener Discovery (RFC 2710). https://tools.ietf.org/html/rfc2710

1.3.4. Multicast Reliability

In sender initiated error recovery, tracking of lost messages is the responsibility of the sender. To do that, the sender requires a positive acknowledgment from each receiver on message delivery.

Multicast Sender Initiated Error Recovery



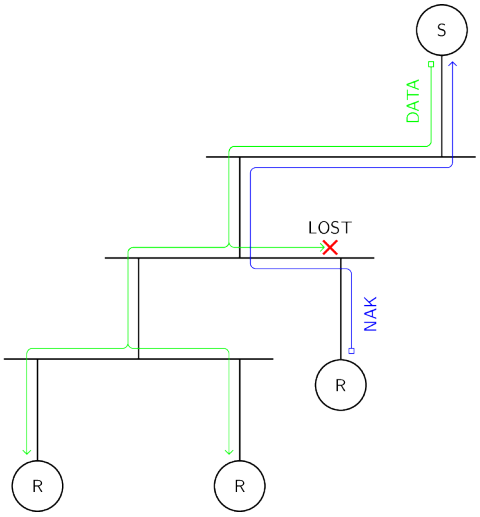
Features.

- Can suffer from ACK implosion
- Sender must know all receivers

- Sender knows when data can be dropped

In receiver initiated error recovery, tracking of lost messages is the responsibility of the receiver. To do that, the receiver transmits a negative acknowledgment on detected message loss.

Multicast Receiver Initiated Error Recovery

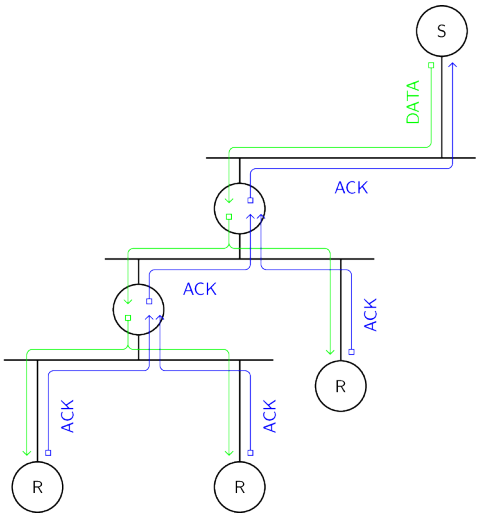


Features.

- Can suffer from NAK implosion
- Sender must transmit keepalive messages.
- Sender does not know when data can be dropped

Acknowledgment messages can be aggregated alongside the network topology.

Aggregated Multicast Error Recovery

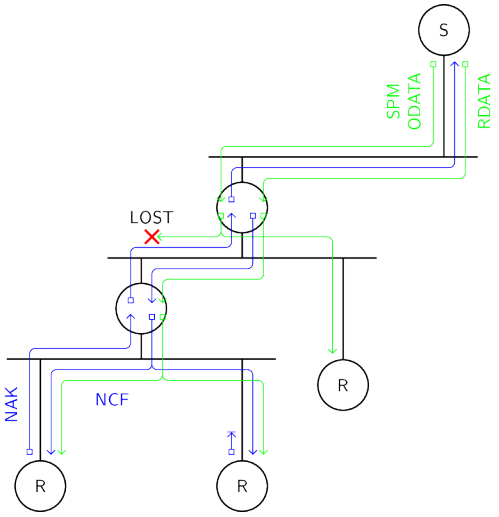


Features.

- Multiple variants with different acknowledgments possible
- Requires cooperation from network elements
- Can be substituted with overlay network

The Pragmatic General Multicast Protocol relies on receiver initiated error recovery with acknowledgment aggregation.

Pragmatic General Multicast



Protocol.

SPM	Source Path Messages establish path information and perform keepalive function
ODATA	Original data packets multicast to all receivers
NAK	Negative Acknowledgment unicast to nearest parent along path
NCF	Negative Acknowledgment Confirmation multicast to children along path
RDATA	Repair data packets multicast to selected receivers

1.3.4.1. References

1. IETF: PGM Reliable Transport Protocol Specification (RFC 3208). https://tools.ietf.org/html/rfc3208

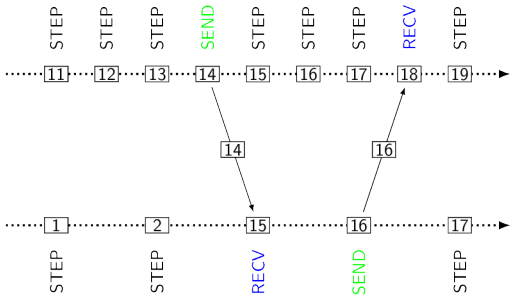
1.3.5. Multicast Ordering

Message Ordering

Source Ordering	Each node defines local order of SEND operations
	Message delivery observes union of the local orderings
Causal Ordering	Each node defines local order of SEND operations
	Each node defines local order of RECV-SEND operation pairs
	Message delivery semantics defines global order of SEND-RECV operation pairs
	Message delivery observes transitive closure of the orderings
Total Ordering	All nodes observe the same order of SEND and RECV operations

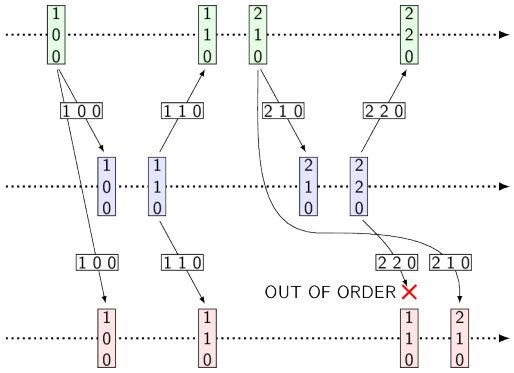
Lamport clock is a type of logical clock that reflects causality in timestamp order.

Lamport Clock



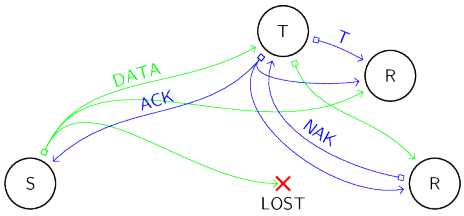
Vector clock is a type of logical clock that captures causality in timestamp values. When message transmissions are counted as significant events, vector clock can be used to provide causal ordering.

Vector Clock



The Token Ring Based Multicast Protocol provides ordering and resiliency guarantees. The current token holder is responsible for totally ordering all messages. The token rotation rules ensure resiliency.

Token Ring Based Multicast



Chapter 2. Systems

2.1. CORBA

CORBA (Common Object Request Broker Architecture) is a standard architecture of a remote procedure call framework that supports heterogeneous object oriented applications. The CORBA standard has evolved through several major revisions, the text in this section is mostly relevant for the later 2.x and early 3.x versions.

2.1.1. Interface Definition Language

The interface definition language is used to describe types used by CORBA, from the basic types of individual arguments to the complex types of interfaces and objects. The language is similar in syntax to C++.

2.1.1.1. Basic Types

The integer types are short, long and long long for signed integer numbers of 16, 32 and 64 bits and unsigned short, unsigned long and unsigned long long for their unsigned counterparts.

Integer Types

short	16 bit signed integer
long	32 bit signed integer
long long	64 bit signed integer
unsigned short	16 bit unsigned integer
unsigned long	32 bit unsigned integer
unsigned long long	64 bit unsigned integer

Values.

```
18, 022, 0x12, 0X12
```

Constants.

```
const short aShortConstant = 6 * 7;
```

The floating point types are float, double and long double for ANSI/IEEE 754-1985 single precision, double precision and double extended precision floating point numbers.

Floating Point Types

float	24 bit signed fraction, 8 bit signed exponent
double	53 bit signed fraction, 11 bit signed exponent
long double	113 bit signed fraction, 15 bit signed exponent

Values.

```
3.14, 12.34e5, 1.2E-4
```

Constants.

```
const float aFloatConstant = 3.141593;
```

The character types are char for a single character in a single-byte character set and wchar for a single character in a multiple-byte character set. The interface definition language itself uses ISO 8859-1 Latin 1.

Character Types

char	character in single-byte character set
wchar	character in multiple-byte character set

Values.

```
'a', '\n', '\000', '\x12'
```

Constants.

```
const char aTab = '\t';
const wchar aWideTab = L'\t';
```

The logical type is boolean with values of true and false.

Logical Types

boolean	logical value

Values.

```
TRUE, FALSE
```

Constants.

```
const boolean aTrueValue = TRUE;
const boolean aFalseValue = FALSE;
```

The special types are octet for 8 bits of raw data and any for a container of another arbitrary type.

Special Types

octet	8 bits of raw data
any	container of another arbitrary type

2.1.1.2. Constructed Data Types

A structure represents a classical compound type with named members that all contain a value.

Structures

Declaration.

```
struct aPerson
{
  string firstName;
  string lastName;
  short  age;
};
```

An exception is a structure that can be returned as an exceptional result of an operation. A number of standard exceptions is defined. Note there is no inheritance in exception declarations, however, language mappings do add inheritance to make it easier to catch standard exceptions.

Exceptions

Declaration.

```
exception anException
{
  string reason;
  string severity;
};
```

Standard System Exception.

```
exception COMM_FAILURE
{
  unsigned long minor;
  completion_status completed;
};
```

A union represents a classical compound type with named members out of which one contains a value. A discriminator is used to determine which of the members contaisn the value.

Unions

Declaration.

```
union aSillyUnion switch (short)
{
  case 1  : long aLongValue;
  case 2  : float aFloatValue;
  default : string aStringValue;
};
```

An enum represents a classical enumerated type with distinct identifiers stored as 32 bit unsigned integers.

Enums

Declaration.

```
enum aBaseColor { red, green, blue }
```

An array is a container for a fixed number of items of the same type addressed by integer indices.

Arrays

Declaration.

```
typedef long aLongArray [10];
```

A sequence is a container for a variable number of items of the same type addressed by integer indices. The maximum number of items in the container can be limited explicitly.

Sequences

Declaration.

```
typedef sequence<long,10> aBoundedVector;
typedef sequence<long> anUnboundedVector;
```

A string is a sequence of char items. A wstring is a sequence of wchar items.

Strings

Declaration.

```
typedef string<10> aBoundedString;
typedef string anUnboundedString;
```

Constants.

```
const string aHello = "Hello\n";
const wstring aWideHello = L"Hello\n";
```

A fixed point type represents a fixed point number of upto 31 significant digits.

Fixed Point Types

Declaration.

```
typedef fixed<10,2> aPrice;
```

Constants.

```
const fixed aPrice = 12.34D;
```

2.1.1.3. Constructed Object Types

An interface type represents an object that is passed by reference and accessed remotely. The declaration of an interface type can specify multiple interface inheritance, attributes and operations. Apart from this, the declaration also creates a lexical scope within which other declarations can appear.

Interface Types

Declaration.

```
abstract interface aParentInterface
{
  attribute string aStringAttribute;
  short aMethod (in long aLongArgument, inout float aFloatArgument);
}

interface aChildInterface : aParentInterface
{
  readonly attribute short aShortAttribute;
  oneway void aOnewayMethod (in long anArgument);
  void aTwowayMethod () raises anException;
}
```

Keywords.

local	interface not invoked remotely
abstract	runtime determines passing semantics
oneway	best effort delivery
readonly	attribute without setter

In some situations, it might be useful to have an interface type that can represent both an object passed by reference and an object passed by value. This is possible when the interface is denoted as abstract.

It is also possible to use interface types to describe objects that are not invoked through CORBA, the interface types are then denoted as local.

A value type represents an object that is passed by value and accessed locally. The declaration of a value type can specify single value type inheritance, single interface and multiple abstract interface support, attributes with private or public visibility, operations and initializers. Apart from this, the declaration also creates a lexical scope within which other declarations can appear.

Value Types

Declaration.

```
valuetype aChildValue : truncatable aParentValue, supports anInterface
{
  private short aShortMember;
  public aParentValue aValueMember;
  factory aFactory (in string anArgument);
  short aLocalMethod (in long aLongArgument, in float aFloatArgument);
}
```

Keywords.

custom	custom marshalling
abstract	base type not instantiated
truncatable	state compatible with parent

public	value used by clients
private	value used by implementation
factory	portable initializer

A value type can support multiple abstract interfaces but only a single interface that is not abstract. When used as an instance of one of the supported abstract interfaces, the value type is passed by value. When used as an instance of the supported interface that is not abstract, the value type is passed by reference.

When an object is passed by value, it might happen that an implementation of its type is not available to the receiver, but an implementation of its parent type is. When a value type is denoted as truncatable, its implementation is considered compatible with the implementation of its parent to the degree that the state of the type can be truncated to the portion inherited from its parent and used by its parent.

A value type that is declared custom will rely on user defined marshalling implementation. A custom value type may not be truncatable.

2.1.2. Language Mapping

The section on language mapping discusses C++ and Java as two major examples. For mostly historical reasons, some mapping constructs do not rely on all the latest features of the target languages, making the language mapping more portable but perhaps potentially less elegant.

Since the goal of the text is to illustrate the issues encountered in language mapping, it outlines the mapping for selected representative types only. Mapping of other types is roughly analogous. Note how in C++, the mapping can use overloading to achieve syntactically simple constructs, but struggles to cope with memory management. In contrast, the mapping to Java sometimes struggles to map types without native counterparts, but memory management is completely transparent.

2.1.2.1. Integer And Floating Point Types

The goal of the integer types mapping is to use native types with matching precision. The use of native types means no conversion is necessary during argument passing. The requirement of matching precision is obviously necessary for correctness.

C++. Because the early versions of the language do not standardize the precision of native integer types, the mapping introduces CORBA integer types that the implementation should use. These types are mapped to native integer types using typedef.

The mapping for C++11 uses standard integer types with explicit precision.

Java. Because the language does not provide unsigned integer types, the mapping uses signed integer types and indicates conversion errors by throwing an exception.

Because the language lacks the ability to pass mutable integer types by reference, special `Holder` classes are defined for all integer types.

Holder Class Example

```
public final class IntHolder
implements org.omg.CORBA.portable.Streamable {
    public int value;

    public IntHolder () { }
    public IntHolder (int o) { value = o; }

    public TypeCode _type () {
        return ORB.init ().get_primitive_tc (TCKind.tk_long);
    }

    public void _read (org.omg.CORBA.portable.InputStream in) {
        value = in.read_long ();
    }

    public void _write (org.omg.CORBA.portable.OutputStream out) {
        out.write_long (value);
    }
}
```

The mapping of floating point types encounters similar problems as the mapping of integer types. These problems are also solved in a similar manner in both C++ and Java.

2.1.2.2. Character And String Types

Besides the usual goal of using native types, mapping of character types also attempts to preserve the meaning of characters in presence of multiple potential encodings.

C++. Because the language does not standardize the encoding of native character types, the mapping assumes that platform specific information will be used to derive the appropriate encoding as necessary.

The language also lacks automated memory management. Special `var` classes and allocator methods are introduced.

Var Class Example

```
class String_var {
private:

    char *data;

public:

    inline String_var ()        { data = 0; }
    inline String_var (char *p) { data = p; }

    inline String_var (const char *p) {
        if (p) data = CORBA::string_dup (p);
        else   data = 0;
    }

    inline ~String_var () {
        CORBA::string_free (data);
    }

    inline String_var &operator = (char *p) {
        CORBA::string_free (data);
        data = p;
        return (*this);
    }
```

```
    inline operator char * () { return (data); }

    inline char &operator [] (CORBA::ULong index) {
        return (data [index]);
    }

    ...
}
```

The `var` classes and allocator methods help prevent memory management errors in common programming constructs.

Var Class Usage

```
void FunctionWithoutLeaks (void) {
    // All strings must be allocated using specific functions
    String_var vSmartPointer = string_dup ("A string ...");

    // Except assignment from const string which copies
    const char *pConstPointer = "A const string ...";
    vSmartPointer = pConstPointer;

    // Assignment releases rather than overwrites
    vSmartPointer = string_dup ("Another string ...");

    // Going out of scope releases too
    throw (0);
}
```

The mapping for C++11 provides reference types whose semantics is equal to that of std::shared_ptr and std::weak_ptr, available through the IDL::traits<T>::ref_type and IDL::traits<T>::weak_ref_type traits. The basic string type is std::string.

2.1.2.3. Any Type

The paramount concern of the any type mapping is making it type safe, that is, making sure the type of the content is always known.

C++. The mapping relies on operator overloading and defines a class with accessor operators for all types that can be stored inside any. This includes accessors for user defined types.

Any Class Example

```
class Any {
public:

    // Types passed by value are easy
    void operator <<= (Any &, Short);
    Boolean operator >>= (const Any &, Short &);
    ...

    // Types passed by reference introduce ownership issues
    void operator <<= (Any &, const Any &);
    void operator <<= (Any &, Any *);
    ...

    // Types where overloading fails introduce resolution issues
    struct from_boolean { from_boolean (Boolean b) : val (b) { } Boolean val; };
    struct from_octet { from_octet (Octet o) : val (o) { } Octet val; };
    struct from_char { from_char (Char c) : val (c) { } Char val; };
    ...

    void operator <<= (from_boolean);
    void operator <<= (from_octet);
    void operator <<= (from_char);
    ...
```

```
    struct to_boolean { to_boolean (Boolean &b) : ref (b) { } Boolean &ref; };
    ...

    Boolean operator >>= (to_boolean) const;
    ...

private:

    // Private operators can detect resolution issues
    unsigned char void operator <<= (unsigned char);
    Boolean operator >>= (unsigned char &) const;
}
```

Operator overloading fails to distinguish IDL types that map to the same native type. This is true for example with the char and octet IDL types, which both map to the char native type. In such situations, wrapping in a distinct type is used.

The any type is assumed to own its content.

Any Class Insertion

```
Any oContainer;

// Small types can be stored easily
Long iLongValue = 1234;
Float fFloatValue = 12.34;
oContainer <<= iLongValue;
oContainer <<= fFloatValue;

// Constant references have copying semantics
const char *pConstString = "A string ...";
oContainer <<= pConstString;

// Non constant references have adoption semantics
String_var vString = string_dup ("A string ...");
oContainer <<= Any::from_string (vString, 0, FALSE);
oContainer <<= Any::from_string (vString._retn (), 0, TRUE);

// Some types need to be resolved explicitly
Char cChar = 'X';
Octet bOctet = 0x55;
oContainer <<= Any::from_char (cChar);
oContainer <<= Any::from_octet (bOctet);
```

Any Class Extraction

```
Any oContainer;

// Small types can be retrieved easily
Long iLongValue;
Float fFloatValue;
if (oContainer >>= iLongValue) ...;
if (oContainer >>= fFloatValue) ...;

// References remain owned by container
const char *pConstString;
if (oContainer >>= Any::to_string (pConstString, 0)) ...;

// Some types need to be resolved explicitly
Char cChar;
Octet bOctet;
if (oContainer >>= Any::to_char (cChar)) ...;
if (oContainer >>= Any::to_octet (bOctet)) ...;
```

Java. The mapping defines a class with accessor methods for all standard types. To keep the any class independent of user defined types, methods for inserting and extracting a user defined type are implemented by helper classes associated with that type.

2.1.2.4. Structures And Exceptions

The mapping of structures and exceptions uses the corresponding object types.

C++. A structure is assumed to own its content.

An exception is equipped with a method to throw its most derived type.

Exception Class Example

```
class Exception {
public:

    // Method for throwing most derived type
    virtual void _raise () const = 0;
    ...
}
```

2.1.2.5. Unions

The paramount concern of the union type mapping is making it type safe, that is, making sure the type of the content is always known.

C++. The mapping defines a class with accessor methods for all types that can be stored inside the union. Each setter method also sets the discriminator as appropriate. Each getter method also tests the discriminator.

Union Class Example

```
class AUnion {
public:

    ...

    void _d (Short);    // Set discriminator
    Short _d() const;   // Get discriminator

    void ShortItem (Short);     // Store ShortItem and set discriminator
    Short ShortItem () const;   // Read ShortItem if stored

    void LongItem (Long);       // Store LongItem and set discriminator
    Long LongItem () const;     // Read LongItem if stored

    ...
}
```

Union Class Usage

```
AUnion oUnion;
Short iShortValue = 1234;
Long iLongValue = 5678;

// Storing sets discriminator
oUnion.ShortItem (iShortValue);
oUnion.LongItem (iLongValue);

// Retrieving must check discriminator
if (oUnion._d () == 1) iShortValue = oUnion.ShortItem ();
if (oUnion._d () == 2) iLongValue = oUnion.LongItem ();
```

Java. The mapping defines a class with accessor methods for all types that can be stored inside the union. Each setter method also sets the discriminator as appropriate. Each getter method also tests the discriminator.

2.1.2.6. Enum Types

C++. The only catch to mapping the enum type is making sure of its size. This is achieved by defining an extra enum member that dictates the size.

Java. The mapping of the enum type should be type safe, that is, instances of different enum types should not be interchangeable among themselves or interchangeable with integer types. This, however, would prevent using instances of enum types in the switch statement. That is why the mapping uses a class to represent an enum but also defines integer constants corresponding to enum instances.

Enum Class Example

```
public class AnEnum {
    public static final int _red = 0;
    public static final AnEnum red = new AnEnum (_red);

    public static final int _green = 1;
    public static final AnEnum green = new AnEnum (_green);

    ...

    public int value () {...};
    public static AnEnum from_int (int value) {...};
}
```

Enum Class Usage

```
AnEnum oEnum;

// Assignments are type safe
oEnum = AnEnum.red;
oEnum = AnEnum.green;

// Switch statements use ordinal values
switch (oEnum.value ()) {
    case AnEnum._red: ...;
    case AnEmum._green: ...;
}
```

2.1.2.7. Sequences

C++. Because the language lacks variable length arrays, sequences are mapped to classes with an overloaded indexing operator. Special `var` classes and allocator methods are introduced.

Sequence Class Example

```
class ASequence {
public:

    ASequence ();
    ASequence (ULong max);
    ASequence (ULong max, ULong length, Short *data, Boolean release = FALSE);

    ...

    ULong maximum () const;
    Boolean release () const;

    void length (ULong);
    ULong length () const;

    T &operator [] (ULong index);
    const T &operator [] (ULong index) const;

    ...
```

```
}
```

The mapping for C++11 provides reference types whose semantics is equal to that of std::shared_ptr and std::weak_ptr, available through the IDL::traits<T>::ref_type and IDL::traits<T>::weak_ref_type traits. The basic sequence type is std::vector.

2.1.2.8. Fixed Point Types

C++. The mapping relies on operator overloading and defines a class with common arithmetic operators. Because the language does not support fixed point constants, the mapping also adds a conversion from a string.

Fixed Class Example

```
class Fixed {
public:

    // Constructors

    Fixed (Long val);
    Fixed (ULong val);
    Fixed (LongLong val);
    Fixed (ULongLong val);
    ...
    Fixed (const char *);

    // Conversions

    operator LongLong () const;
    operator LongDouble () const;
    Fixed round (UShort scale) const;
    Fixed truncate (UShort scale) const;

    // Operators

    Fixed &operator = (const Fixed &val);
    Fixed &operator += (const Fixed &val);
    Fixed &operator -= (const Fixed &val);
    ...
}

Fixed operator + (const Fixed &val1, const Fixed &val2);
Fixed operator - (const Fixed &val1, const Fixed &val2);
...
```

Java. The mapping simply uses the `BigDecimal` class.

2.1.2.9. Proxies

Since the proxy should resemble an implementation of the interface that it represents, the mapping will generally use the native interface and object constructs of the target language in a straightforward manner. What makes proxies interesting are the subtle typing issues that arise.

C++. The IDL interface is represented by a C++ class with virtual methods for IDL operations. The proxy is a platform specific class that inherits from the interface class. Safe type casting over remote types requires the addition of the `narrow` method.

Proxy Interface Class Example

```
class AnInterface;
typedef AnInterface *AnInterface_ptr;
class AnInterface_var;


class AnInterface : public virtual Object {
```

```
public:

    typedef AnInterface_ptr _ptr_type;
    typedef AnInterface_var _var_type;

    static AnInterface_ptr _duplicate (AnInterface_ptr obj);
    static AnInterface_ptr _narrow (Object_ptr obj);
    static AnInterface_ptr _nil ();

    virtual ... AnOperation (...) = 0;

protected:

    AnInterface ();
    virtual ~AnInterface ();

    ...
}
```

Memory management issues are solved by introducing reference counting and `var` classes.

Proxy Var Class Example

```
class AnInterface_var : public _var {
protected:

    AnInterface_ptr ptr;

public:

    AnInterface_var () { ptr = AnInterface::_nil (); }
    AnInterface_var (AnInterface_ptr p) { ptr = p; }

    ...

    ~AnInterface_var () {
        release (ptr);
    }

    AnInterface_var &operator = (AnInterface_ptr p) {
        release (ptr);
        ptr = p;
        return (*this);
    }

    AnInterface_var &operator = (const AnInterface_var &var) {
        if (this != &var) {
            release (ptr);
            ptr = AnInterface::_duplicate (AnInterface_ptr (var));
        }
        return (*this);
    }

    operator AnInterface_ptr & () { return (ptr); }
    AnInterface _ptr operator -> () const { return (ptr); }

    ...
}
```

The mapping for C++11 provides reference types whose semantics is equal to that of std::shared_ptr and std::weak_ptr, available through the IDL::traits<T>::ref_type and IDL::traits<T>::weak_ref_type traits. Casting to derived interfaces is supported through a `IDL::traits<T>::narrow` method.

Java. The IDL interface is represented by a Java interface with methods for IDL operations. The proxy is a platform specific class that implements the Java interface. Safe type casting over remote types requires the addition of the `narrow` method. Still more methods are present in a helper class that facilitates insertion and extraction to and from the any type together with the marshalling operations. The standardization of the marshalling operations makes it possible to use proxy classes in a platform independent manner.

Proxy Class Example

```
public interface AnInterfaceOperations {
    ... AnOperation (...) throws ...;
}

public interface AnInterface extends AnInterfaceOperations ... { }

abstract public class AnInterfaceHelper {
    public static void insert (Any a, AnInterface t) {...}
    public static AnInterface extract (Any a) {...}
    public static AnInterface read (InputStream is) {...}
    public static void write (OutputStream os, AnInterface val) {...}
    ...

    public static AnInterface narrow (org.omg.CORBA.Object obj) {...}
    public static AnInterface narrow (java.lang.Object obj) {...}
}

final public class AnInterfaceHolder implements Streamable {
    public AnInterface value;
    public AnInterfaceHolder () { }
    public AnInterfaceHolder (AnInterface initial) {...}

    ...
}
```

2.1.2.10. Servants

Where the mapping of the proxy selects the target type with transparency in mind, the mapping of the servant provides enough freedom in situations where strict typing constraints are not desirable. This is achieved by coupling servants to interfaces either by inheritance or by delegation.

C++. The servant mapping starts with a reference counted servant base class. The reference counting of servants is distinct from the reference counting of proxies.

Servant Base Class

```
class ServantBase {
public:

    virtual ~ServantBase ();

    virtual InterfaceDef_ptr _get_interface () throw (SystemException);
    virtual Boolean _is_a (const char *logical_type_id) throw (SystemException);
    virtual Boolean _non_existent () throw (SystemException);

    virtual void _add_ref ();
    virtual void _remove_ref ();

    ...
}
```

An abstract C++ class is generated for each IDL interface, the servant implementation can inherit from this abstract class and implement its methods as necessary. Alternatively, templates can be used to tie the servant implementation to a type that inherits from the abstract class.

Servant Class Example

```
class POA_AnInterface : public virtual ServantBase {
public:

    virtual ... AnOperation (...) = 0;

    ...
}
```

```
template <class T> class POA_AnInterface_tie : public POA_AnInterface {
public:

    POA_AnInterface_tie (T &t) : _ptr (t) { }

    ...

    ... AnOperation (...) { return (_ptr->AnOperation (...); }
}
```

C++11. The servant mapping starts with a servant base class.

Servant Base Class

```
class Servant {
public:

    virtual IDL::traits<CORBA::InterfaceDef>::ref_type _get_interface ();
    virtual bool _is_a (const std::string &logical_type_id);
    virtual bool _non_existent ();

    ...

protected:

    virtual ~Servant ();
}
```

An abstract C++ class is generated for each IDL interface, the servant implementation can inherit from this abstract class and implement its methods as necessary.

Servant Class Example

```
class _AnInterface_Servant_Base : public virtual Servant {
public:

    virtual ... AnOperation (...) = 0;

    ...
}

class AnInterface_Servant : public virtual CORBA::servant_traits<AnInterface>::base_type {
public:

    virtual ... AnOperation (...) override;
}
```

Java. The servant mapping starts with a servant base class.

Servant Base Class

```
abstract public class Servant {
    final public Delegate _get_delegate () { ... }
    final public void _set_delegate (Delegate delegate) { ... }
    ...
}
```

An abstract Java class is generated for each IDL interface, the servant implementation can inherit from this class and implement its methods as necessary. Alternatively, delegation can be used to tie the servant implementation to a type that inherits from the abstract class.

Servant Class Example

```
abstract public class AnInterfacePOA implements AnInterfaceOperations {
    public AnInterface _this () { ... }
```

```
    ...
}

public class AnInterfacePOATie extends AnInterfacePOA {
    private AnInterfaceOperations _delegate;

    public AnInterfacePOATie (AnInterfaceOperations delegate)
    { _delegate = delegate; }

    public AnInterfaceOperations _delegate ()
    { return (_delegate); }

    public void _delegate (AnInterfaceOperations delegate)
    { _delegate = delegate; }

    public ... AnOperation (...) { return (_delegate.AnOperation (...); }
}
```

2.1.2.11. Value Types

C++. The language lacks both dynamic type creation and instance state access. The mapping therefore implements both, the type creation by factories and the state access by accessor methods. Custom marshalling interface is available for situations where generated marshalling code based on accessor methods is not appropriate.

Value Mapping Example

```
class AValue : public virtual ValueBase {
public:

    virtual void ShortItem (Short) = 0;
    virtual Short ShortItem () const = 0;

    virtual void LongItem (Long) = 0;
    virtual Long LongItem () const = 0;

    ...

    virtual ... AnOperation (...) = 0;
}

class OBV_AValue : public virtual AValue {
public:

    virtual void ShortItem (Short) { ... };
    virtual Short ShortItem () const { ... };

    virtual void LongItem (Long) { ... };
    virtual Long LongItem () const { ... };

    ...

    virtual ... AnOperation (...) = 0;
}

class ValueFactoryBase {
private:

    virtual ValueBase *create_for_unmarshal () = 0;

    ...
}

class AValue_init : public ValueFactoryBase {
public:

    virtual AValue *AConstructor (...) = 0;

    ...
```

```
}
```

Java. The language provides both dynamic type creation and instance state access. The mapping therefore only provides a custom marshalling interface for situations where generated marshalling code based on serialization is not appropriate.

2.1.2.12. Argument Passing

It is also worth noting some broader aspects of argument passing.

C++. The language mapping attempts to minimize copying by preferring stack allocation to heap allocation whenever possible. The caller often allocates memory for values returned by the callee, otherwise stack allocation would not be possible. As an unfortunate complication, fixed size types and variable size types have to be distinguished.

The mapping for C++11 simplifies the argument passing rules. All primitive types are passed by value when input and by reference when output. All other types are passed by constant reference when input and by reference when output.

Java. Since the language does not allow passing some types by reference, holder classes are generated to solve the need for mapping output arguments.

2.1.3. Object Adapter

The object adapter delivers requests to servants using a mapping from object ID values to servant references. An object ID is an opaque sequence of octets assigned to each object by the server. Incoming requests identify the target objects using their object ID.

The object adapter specification supports multiple configurations that govern the process of delivering requests to servants. Some configurations use an active object map to map object ID values to servant references. Other configurations use custom servant managers to determine the mapping. It is also possible to configure the threading model used to invoke servants. The configuration is set using policies.

Object Adapter Configuration

```
local interface POA {
  POA create_POA (in string adapter_name,
                  in POAManager manager,
                  in CORBA::PolicyList policies);

    ThreadPolicy create_thread_policy (in ThreadPolicyValue value);
    LifespanPolicy create_lifespan_policy (in LifespanPolicyValue value);
    ServantRetentionPolicy create_servant_retention_policy (in ServantRetentionPolicyValue value);
    RequestProcessingPolicy create_request_processing_policy (in RequestProcessingPolicyValue value

    ...
};

local interface POAManager {
    enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };
    State get_state ();

    void activate () raises (AdapterInactive);
    void hold_requests (in boolean wait_for_completion) raises (AdapterInactive);
    void discard_requests (in boolean wait_for_completion) raises (AdapterInactive);

    void deactivate (in boolean etherealize_objects,
                     in boolean wait_for_completion);
};
```

The threading model configuration is restricted to general categories. A particular object adapter implementation can provide more detailed threading model configuration. Typical configurations include the single threaded model and the leader-follower thread pool model.

Thread Policy

Thread Policy Values.

SINGLE_THREAD_MODEL	calls to servants and managers are serialized
MAIN_THREAD_MODEL	calls to servants are using single main thread
ORB_CTRL_MODEL	calls use arbitrary threading model

The object identity policies default to an automatically assigned system identity. Explicit configuration allows for custom identities, useful especially when object state is external, rather than encapsulated in the servant. Each servant can query the object identity associated with current request.

Object Identity Policies

ID Uniqueness Policy Values.

UNIQUE_ID	servants have exactly one object ID
MULTIPLE_ID	servants have at least one object ID

ID Assignment Policy Values.

USER_ID	object ID is assigned by application
SYSTEM_ID	object ID is assigned by object adapter

Implicit Activation Policy Values.

IMPLICIT_ACTIVATION	assign object ID on demand
NO_IMPLICIT_ACTIVATION	do not assign object ID on demand

Object Activation

```
ObjectId activate_object (in Servant servant) raises (ServantAlreadyActive, WrongPolicy);

void activate_object_with_id (in ObjectId oid, in Servant servant)
raises (ObjectAlreadyActive, ServantAlreadyActive, WrongPolicy);

void deactivate_object (in ObjectId oid) raises (ObjectNotActive, WrongPolicy);

Object create_reference (in CORBA::RepositoryId ifc) raises (WrongPolicy);
Object create_reference_with_id (in ObjectId oid, in CORBA::RepositoryId ifc);

Object servant_to_reference (in Servant servant) raises (ServantNotActive, WrongPolicy);
Servant reference_to_servant (in Object reference) raises (ObjectNotActive, WrongAdapter, WrongPoli
```

Current Object Interface

```
local interface Current {
    POA get_POA () raises (NoContext);
    ObjectId get_object_id () raises (NoContext);
    Object get_reference () raises (NoContext);
    Servant get_servant () raises (NoContext);
};
```

A request can be delivered to a servant tracked in the active object map, a servant identified by one of the two servant manager types, or a default servant.

Servant Lookup Policies

Servant Retention Policy Values.

RETAIN	keep track of active servants

NON_RETAIN	do not keep track of active servants

Request Processing Policy Values.

USE_ACTIVE_OBJECT_MAP_ONLY	only deliver to tracked servants
USE_DEFAULT_SERVANT	alternatively deliver to default servant
USE_SERVANT_MANAGER	alternatively activate servants on demand

Servant Activator Interface

```
local interface ServantActivator : ServantManager {

    Servant incarnate (in ObjectId oid,
                       in POA adapter)
    raises (ForwardRequest);

    void etherealize (in ObjectId oid,
                      in POA adapter,
                      in Servant servant,
                      in boolean cleanup_in_progress,
                      in boolean remaining_activations};
};
```

Servant Locator Interface

```
local interface ServantLocator : ServantManager {

    native Cookie;

    Servant preinvoke (in ObjectId oid,
                       in POA adapter,
                       in CORBA::Identifier operation,
                       out Cookie cookie)
    raises (ForwardRequest);

    void postinvoke (in ObjectId oid,
                     in POA adapter,
                     in CORBA::Identifier operation,
                     in Cookie cookie,
                     in Servant servant);
};
```

A request forwarding mechanism supports creating object references whose lifetime exceeds that of the server.

Lifespan Policy

Lifespan Policy Values.

TRANSIENT	object references have lifetime of object adapter
PERSISTENT	object references have potentially unlimited lifetime

Request Forward Exception

```
exception ForwardRequest {
    Object forward_reference;
};
```

2.1.4. Network Protocol

The network protocol is defined in two layers. The lower layer introduces the General Inter-ORB Protocol (GIOP), which defines the Common Data Representation (CDR), the message formats and

the transport assumptions. The upper layer introduces the Internet Inter-ORB Protocol (IIOP), which specializes the lower layer for IP networks.

The Common Data Representation supports both byte orderings. Among interesting features are type codes, which serve to recursively describe the transported types where needed, and encapsulations, which serve to wrap already encoded data. Object references support multiple profiles, each profile describes one way to access the remote object.

2.1.5. Messaging

Synchronization Scope Policy

SYNC_NONE
SYNC_WITH_TRANSPORT
SYNC_WITH_SERVER
SYNC_WITH_TARGET

Routing Policy

ROUTE_NONE
ROUTE_FORWARD
ROUTE_STORE_AND_FORWARD

Asynchronous Messaging Mapping Example

Interface.

```
interface StockManager {
    attribute string stock_exchange_name;
    boolean add_stock (in string symbol, in double quote);
    void remove_stock (in string symbol, out double quote) raises (InvalidStock);
};
```

Callback Mapping.

```
void sendc_get_stock_exchange_name (
    in AMI_StockManagerHandler ami_handler);
void sendc_set_stock_exchange_name (
    in AMI_StockManagerHandler ami_handler,
    in string attr_stock_exchange_name);

void sendc_add_stock (
    in AMI_StockManagerHandler ami_handler,
    in string symbol,
    in double quote);

void sendc_remove_stock (
    in AMI_StockManagerHandler ami_handler,
    in string symbol);


interface AMI_StockManagerHandler : Messaging::ReplyHandler {
    void get_stock_exchange_name (
        in string ami_return_val);
    void get_stock_exchange_name_excep (
        in Messaging::ExceptionHolder excep_holder);

    void set_stock_exchange_name ();
    void set_stock_exchange_name_excep (
        in Messaging::ExceptionHolder excep_holder);

    void add_stock (in boolean ami_return_val);
    void add_stock_excep (
```

```
        in Messaging::ExceptionHolder excep_holder);

    void remove_stock (in double quote);
    void remove_stock_excep (
        in Messaging::ExceptionHolder excep_holder);
};
```

Poller Mapping.

```
AMI_StockManagerPoller sendp_get_stock_exchange_name ();
AMI_StockManagerPoller sendp_set_stock_exchange_name (
    in string attr_stock_exchange_name);

AMI_StockManagerPoller sendp_add_stock (
    in string symbol, in double quote);
AMI_StockManagerPoller sendp_remove_stock (
    in string symbol);


valuetype AMI_StockManagerPoller : Messaging::Poller {
    void get_stock_exchange_name (
        in unsigned long timeout,
        out string ami_return_val);
    void set_stock_exchange_name (
        in unsigned long timeout);

    void add_stock (
        in unsigned long timeout,
        out boolean ami_return_val);

    void remove_stock (
        in unsigned long timeout,
        out double quote) raises (InvalidStock);
};
```

2.1.6. Components

Component Features

attributes	denote configurable properties
supported interface	inherited in all interfaces
facets	interfaces provided to the outside
receptacles	interfaces required from the outside
sources	events produced to the outside
sinks	events consumed from the outside

Component Definition Example

```
module DiningPhilosophers {
  interface IFork {
    void pick_up () raises (ForkNotAvailable);
    void release ();
  };

  component AFork {
    provides IFork fork;
  };

  eventtype PhilosopherStatus {
      public string name;
      public PhilosopherState state;
      public boolean has_left_fork;
      public boolean has_right_fork;
  };

  component APhilosopher {
```

```
    attribute string name;

    // Receptacles for forks
    uses Fork left;
    uses Fork right;

    // Source for status
    publishes PhilosopherStatus status;
  };

  component AnObserver {
    // Sink for status
    consumes PhilosopherStatus status;
  };

  ...
};
```

Navigation Interfaces

```
module Components {
    typedef string FeatureName;
    typedef sequence<FeatureName> NameList;

    valuetype PortDescription {
        public FeatureName name;
        public CORBA::RepositoryId type_id;
    };

  ...


    valuetype FacetDescription : PortDescription {
        public Object facet_ref;
    };
    typedef sequence<FacetDescription> FacetDescriptions;

    interface Navigation {
        FacetDescriptions get_all_facets ();
        Object provide_facet (in FeatureName name) raises (InvalidName);
        FacetDescriptions get_named_facets (in NameList names) raises (InvalidName);

    ...
    };

    ...


    valuetype PublisherDescription : PortDescription {
        public SubscriberDescriptions consumers;
    };
    typedef sequence<PublisherDescription> PublisherDescriptions;

    valuetype ConsumerDescription : PortDescription {
        public EventConsumerBase consumer;
    };
    typedef sequence<ConsumerDescription> ConsumerDescriptions;

    PublisherDescriptions get_all_publishers ();
    PublisherDescriptions get_named_publishers (in NameList names) raises (InvalidName);

    ConsumerDescriptions get_all_consumers ();
    ConsumerDescriptions get_named_consumers (in NameList names) raises (InvalidName);

    ...
};
```

Assembly Interfaces

```
uses AnInterface AReceptacle;
consumes AnEvent ASink;


void connect_AReceptacle (in AnInterface connection)
    raises (AlreadyConnected, InvalidConnection);
AnInterface disconnect_AReceptacle ()
    raises (NoConnection);
AnInterface get_connection_AReceptacle ();


AnEventConsumer get_consumer_ASink ();


module Components {
    ...

    interface Receptacles {
        Cookie connect (in FeatureName name, in Object connection)
            raises (InvalidName, InvalidConnection, AlreadyConnected, ExceededConnectionLimit);
        Object disconnect (in FeatureName name, in Cookie ck)
            raises (InvalidName, InvalidConnection, CookieRequired, NoConnection);
        ConnectionDescriptions get_connections (in FeatureName name)
            raises (InvalidName);

        ...
    };

    ...


    valuetype Cookie {
        private CORBA::OctetSeq cookieValue;
    };

    valuetype SubscriberDescription {
        public Cookie ck;
        public EventConsumerBase consumer;
    };
    typedef sequence<SubscriberDescription> SubscriberDescriptions;

    interface Events {
        void connect_consumer (in FeatureName emitter_name, in EventConsumerBase consumer)
            raises (InvalidName, AlreadyConnected, InvalidConnection);
        EventConsumerBase disconnect_consumer (in FeatureName source_name)
            raises (InvalidName, NoConnection);
        EventConsumerBase get_consumer (in FeatureName sink_name) raises (InvalidName);

        Cookie subscribe (in FeatureName publisher_name, in EventConsumerBase subscriber)
            raises (InvalidName, InvalidConnection, ExceededConnectionLimit);
        EventConsumerBase unsubscribe (in FeatureName publisher_name, in Cookie ck)
            raises (InvalidName, InvalidConnection);

        ...
    };

    ...
};
```

2.1.7. References

1. Remedy IT: CORBA Programmers Guide. http://www.remedy.nl/opensource/corbapg.html

2. OMG: CORBA Interface Definition Language Specifications. https://www.omg.org/spec/category/interface-definition-language

3. OMG: CORBA Language Mapping Specifications. https://www.omg.org/spec/category/language-mapping

4. OMG: CORBA Platform Specifications. https://www.omg.org/spec/category/corba-platform

2.2. Data Distribution Service (DDS)

Reliability Related Policies

RELIABILITY. Selects either best effort or guaranteed delivery mechanism.

BEST_EFFORT	no special mechanism to guarantee delivery
RELIABLE	guarantee delivery at transport protocol level

OWNERSHIP. Configures potential redundancy at publisher side.

SHARED	deliver messages from all writers
EXCLUSIVE	deliver messages from live writer with highest strength

OWNERSHIP_STRENGTH. Set writer strength to be used with EXCLUSIVE OWNERSHIP policy.

Presentation Related Policies

PRESENTATION. Determines how change messages are presented to application.

coherent access	group change messages into explicitly delimited transactions
ordered access	preserve order of change messages
access scope	scope access options

INSTANCE	access options apply at instance scope
TOPIC	access options apply at Topic object scope
GROUP	access options apply at Publisher or Subscriber object scope

DESTINATION_ORDER. Determines how to handle concurrent updates.

BY_SOURCE_TIMESTAMP	value with highest source timestamp will be visible
BY_RECEPTION_TIMESTAMP	value with highest reception timestamp will be visible

History Related Policies

DURABILITY. Availability of data for late joining readers.

VOLATILE	writer does not keep any history
TRANSIENT_LOCAL	history kept in writer local memory
TRANSIENT	history kept in session local memory
PERSISTENT	history kept in persistent storage

HISTORY. How much history to keep.

KEEP_LAST	keep limited history with configurable depth
KEEP_ALL	keep all history within resource limits

RESOURCE_LIMITS. What are the available resource limits.

max_samples	maximum number of samples managed across all instances
max_instances	maximum number of managed instances
max_samples_per_instance	maximum number of samples managed per single instance

Timing Related Policies

DEADLINE. Guarantee periodic updates to all topic instances.

- publisher can guarantee maximum update period
- subscriber can require maximum update period

LATENCY_BUDGET. Hint on available latency reserve.

TRANSPORT_PRIORITY. Hint on requested transport priority.

LIVELINESS. Configure how entity liveliness is determined.

AUTOMATIC	service tracks Entity object liveliness
MANUAL_BY_TOPIC	publisher must periodically assert liveliness per Topic object
MANUAL_BY_PARTICIPANT	publisher must periodically assert liveliness per Participant object

LIFESPAN. Message expiration time. Relies on having synchronized clock.

TIME_BASED_FILTER. Minimum separation time for incoming messages.

Miscellaneous Policies

USER_DATA	attaches arbitrary data to Entity objects
TOPIC_DATA	attaches arbitrary data to Topic objects
GROUP_DATA	attaches arbitrary data to Publisher and Subscriber objects

PARTITION define a partition name for logical domain partitioning

2.3. Enterprise JavaBeans (EJB)

EJB (Enterprise JavaBeans) is a standard architecture of an environment for hosting server tiers of enterprise component applications. The EJB standard has evolved through several major revisions, which, besides introducing particular features, have also included changing the programming interface due to the introduction of language annotations. The text in this section deals mostly with the latest version.

An EJB application consists of components called enterprise beans that reside in a container. The beans implement the business logic of the application, the container provides the beans with standard services including lifecycle, dependency management, persistency, transactions, and makes the beans accessible to the clients.

Beans come in three distinct types, namely session objects, entity objects and message driven objects. The session objects are further split into stateless, stateful and singleton variants. Each type is tailored to fill a specific role in a component application.

The dependency management can use the CDI (Contexts and Dependency Injection) framework. The framework defines scopes that associate beans with particular execution context. An execution context is responsible for obtaining contextual instances of beans, this typically involves creating instances and looking up created instances.

EJB Architecture

Containers. Environment providing services to enterprise application objects

- Lifecycle management (creating and deleting instances)
- Dependency management (resource and dependency injection)

- Persistence and transactions
- ...

Enterprise Beans. Enterprise application objects managed by container

Stateful session bean	an object that lives within user session scope, state survives method invocation
Stateless session bean	an object that lives within user session scope, state only within method invocation
Singleton session bean	a singleton application object
Entity bean	an object representing persistent database state
Message driven bean	an object that handles messages, state only within message handling

Scopes and Contexts.

```
public interface Context {
    public Class <? extends Annotation> getScope ();
    public <T> T get (Contextual <T> bean);
    public <T> T get (Contextual <T> bean, CreationalContext <T> creationalContext);
    ...
}
```

@Dependent	instance bound to single injection point
@RequestScoped	context associated with single method invocation or request handling
@SessionScoped	context associated with an HTTP session
@ApplicationScoped	context associated with application execution
@ConversationScoped	context associated with explicitly delimited UI interaction

2.3.1. Session Objects

Stateful Session Bean Example

```
@Stateful public class ASessionBean implements ABusinessInterface {

    // Injected reference to standard session context object
    @Resource public SessionContext sessionContext;

    // Method that is called after construction or activation
    @PostConstruct @PostActivate
    public void myInitMethod () { ... }

    // Method that is called before passivation or destruction
    @PreDestroy @PrePassivate
    public void myDoneMethod () { ... }

    // Some business methods ...
    public void myMethodOne (int iArgument) { ... }
    public int myMethodTwo (Object oArgument) { ... }

    // Business method with asynchronous interface
    @Asynchronous public Future<String> myAsynchronousMethod (String sArgument) { ... }

    // Business method that removes the bean instance
    @Remove public void myRemovalMethod () { ... }

    // Interceptor method that can also be in separate interceptor class
    @AroundInvoke
    public Object myInterceptor (InvocationContext inv)
    throws Exception {
        ...
        Object result = inv.proceed ();
```

```
        ...
        return (result);
    }
}
```

- method invocations serialized by the container
- asynchronous method invocations handled by the container
- business interface can be accessed both locally and remotely
- references obtained through dependency injection or JNDI lookup

Client life cycle view.

- accessible when reference first obtained
- removed through explicit removal method

Container life cycle view.

- instance may be passivated and activated

Singleton Session Bean Example

```
@Startup @Singleton public class ASingletonBean {

    // Injected reference to JNDI resource
    @Resource (lookup = "java:comp/env/jdbc/SomeDataSource") DataSource dataSource;

    // Method that is called after construction
    @PostConstruct
    public void myInitMethod () { ... }

    // Method that is called before destruction
    @PreDestroy
    public void myDoneMethod () { ... }

    // Some business methods ...
    public synchronized void myMethodOne (int iArgument) { ... }
    public synchronized int myMethodTwo (Object oArgument) { ... }
}
```

- concurrent method invocations possible

Container life cycle view.

- construction timing determined by container
- eager construction can be requested through annotation

EJB Sessions and CDI Sessions

Unless explicitly disabled, each EJB session bean instance is bound to some CDI context. The default scope is `@Dependent`, which ties the EJB session bean instance lifecycle to the lifecycle of the injection point. An EJB session bean is therefore not necessarily associated with CDI session scope.

2.3.2. Message Driven Objects

Message Driven Bean Example

```
@MessageDriven (activationConfig = {
    @ActivationConfigProperty (
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty (
        propertyName = "destinationLookup",
```

```
        propertyValue = "jms/SomeQueue")
})
public class AMessageBean implements MessageListener {

    public AMessageBean () { ... }

    @Override public void onMessage (Message aMessage) { ... }

    ...
}
```

- method invocations serialized by the container
- multiple instances can be created by the container
- message delivery order between instances is not defined

2.3.3. Entity Objects

Field Based Entity Bean Class Example

```
@Entity public class AnEntity {

    // With field based access fields are persistent by default
    private int someField;
    private String someOtherField;

    // Relationships among entities must be annotated
    @OneToMany private Collection <AnotherEntity> relatedEntities;

    // Every entity must have a primary key
    @Id private long aKeyField;

    // Field that is not persistent
    @Transient private String aTransientString;

    // Version field for optimistic concurrency
    @Version private long version = 0L;

    // Obligatory constructor with no arguments
    public AnEntity () { ... }

    // Additional business methods ...
    public void myMethodOne (int iArgument) { ... }
    public int myMethodTwo (Object oArgument) { ... }
}
```

Property Based Entity Bean Class Example

```
@Entity public class AnEntity {

    // With property based access fields are not persistent themselves
    private int someTransientField;
    private String someOtherTransientField;

    // Relationships among entities must be annotated
    private Collection <AnotherEntity> relatedEntities;
    @OneToMany public Collection <AnotherEntity> getRelatedEntities () {
        return (relatedEntities);
    }
    public void setRelatedEntities (Collection <AnotherEntity> entityCollection) {
        relatedEntities = entityCollection;
    }

    // Getter and setter methods for primary key
    private long aKeyField;
    @Id Long getAKeyField () { return (aKeyField); }
    public void setAKeyField (Long aKeyField) { this.aKeyField = aKeyField; }
```

```
    // Obligatory constructor with no arguments
    public AnEntity () { ... }

    // Additional business methods ...
    public void myMethodOne (int iArgument) { ... }
    public int myMethodTwo (Object oArgument) { ... }
}
```

Entity Manager Interface

```
public interface EntityManager {

    void persist (Object entity);
    void refresh (Object entity);
    void remove (Object entity);

    void detach (Object entity);
    <T> T merge (T entity);

    void lock (Object entity, LockModeType lockMode);

    // Find by primary key
    <T> T find (Class <T> entityClass, Object primaryKey);

    // Find by primary key and return lazy reference
    <T> T getReference (Class <T> entityClass, Object primaryKey);

    // Clear persistence context and detach all entities
    void clear ();

    // Check whether persistence context contains managed entity
    boolean contains (Object entity);

    // Synchronize persistence context with database
    // Flush mode governs automatic synchronization
    // upon query execution or upon commit
    void flush ();
    FlushModeType getFlushMode ();
    void setFlushMode (FlushModeType flushMode);

    Query createQuery (String ejbqlString);
    Query createNamedQuery (String name);
    Query createNativeQuery (String sqlString);
    ...
}
```

Query Interface

```
public interface Query {

    // Execute a query that returns a result list
    List getResultList ();
    // Execute a query that returns a single result
    Object getSingleResult();
    // Execute an update query
    int executeUpdate ();

    // Methods used to fetch results step by step
    Query setMaxResults (int maxResult);
    Query setFirstResult (int startPosition);

    // Bind a parameter in a query
    Query setParameter (String name, Object value);
    Query setParameter (String name, Date value, TemporalType temporalType);
    Query setParameter (String name, Calendar value, TemporalType temporalType);
    Query setParameter (int position, Object value);
    Query setParameter (int position, Date value, TemporalType temporalType);
    Query setParameter (int position, Calendar value, TemporalType temporalType);
}
```

2.3.4. Transactions

The flat transaction model is supported. Transactions are demarcated either by the beans or by the container. When bean managed transaction demarcation is used, the individual methods of a bean can explicitly begin and commit or rollback a transaction. When container managed transaction demarcation is used, the individual methods of a bean can use transaction attributes, specified either in the method annotations or in the deployment descriptor. The transaction attributes tell the container how to demarcate the transactions.

Transaction Annotations

```
@Stateful @TransactionManagement (BEAN) public class SomeBeanClass implements SomeBeanInterface {
    @Resource javax.Transaction.UserTransaction transaction;

    public void myMethodOne () {
        transaction.begin ();
        ...
    }

    public void myMethodTwo () {
        ...
        transaction.commit ();
    }
}

@Stateless public class SomeBeanClass implements SomeBeanInterface {
    @TransactionAttribute (REQUIRED) public void myMethod () {
        ...
    }
}
```

BEAN	bean demarcated transactions
NEVER	transaction must not be active or exception is thrown
MANDATORY	transaction must be active or exception is thrown
SUPPORTS	no specific demarcation performed by the container
NOT_SUPPORTED	active transaction suspended during call
REQUIRED	active transaction used or new transaction started during call
REQUIRES_NEW	active transaction suspended and new transaction started during call

The state of a session bean is not a transactional resource and therefore is not influenced by transaction commit or rollback. A session bean can implement the `SessionSynchronization` interface to receive the `afterBegin`, `beforeCompletion`, `afterCompletion` notifications. These can be used to commit or rollback the state of the bean explicitly.

2.3.5. References

1. JSR 345: Enterprise JavaBeans 3.2. http://jcp.org/en/jsr/detail?id=345

2. JSR 346: Contexts and Dependency Injection for Java EE 1.2. http://jcp.org/en/jsr/detail?id=346

3. JSR 338: Java Persistence 2.2. http://jcp.org/en/jsr/detail?id=338

2.4. etcd

Etcd is a distributed service configuration repository, natively accessible through gRPC.

Etcd Data Model

Single cluster wide key value store

- Keys and values are byte arrays
- Keys are sorted lexicographically
- Store has monotonically increasing revisions
- Used to timestamp key creation and modification
- Historical revisions available until compaction
- Keys have increasing versions
- Keys can have leases

```
message KeyValue {
    bytes key = 1;
    int64 create_revision = 2;
    int64 mod_revision = 3;
    int64 version = 4;
    bytes value = 5;
    int64 lease = 6;
}
```

Put Request

```
rpc Put (PutRequest) returns (PutResponse) { }

message PutRequest {
    bytes key = 1;
    bytes value = 2;

    // Lease identifier or 0 for no lease
    int64 lease = 3;

    // Optionally return previous key value pair
    bool prev_kv = 4;

    // Optionally update using existing value
    bool ignore_value = 5;

    // Optionally update using existing lease
    bool ignore_lease = 6;
}

message PutResponse {
    ResponseHeader header = 1;
    KeyValue prev_kv = 2;
}
```

Range Request

```
rpc Range (RangeRequest) returns (RangeResponse) { }

message RangeRequest {

    enum SortOrder {
        NONE = 0;
        ASCEND = 1;
        DESCEND = 2;
    }

    enum SortTarget {
        KEY = 0;
        VERSION = 1;
        CREATE = 2;
        MOD = 3;
        VALUE = 4;
    }

    bytes key = 1;
    bytes range_end = 2;
```

```
    // Restrict number of keys returned
    int64 limit = 3;

    // Possibly query historical revision
    int64 revision = 4;

    SortOrder sort_order = 5;
    SortTarget sort_target = 6;

    // Linearizable returns cluster consensus
    // Serializable can return stale data
    bool serializable = 7;

    bool keys_only = 8;
    bool count_only = 9;

    int64 min_mod_revision = 10;
    int64 max_mod_revision = 11;

    int64 min_create_revision = 12;
    int64 max_create_revision = 13;
}

message RangeResponse {
    ResponseHeader header = 1;
    repeated KeyValue kvs = 2;
    bool more = 3;
    int64 count = 4;
}
```

- also delete range request

Transaction Request

```
rpc Txn (TxnRequest) returns (TxnResponse) { }

message TxnRequest {
    // List of tests to perform before transaction
    repeated Compare compare = 1;
    // List of operations to perform when all tests succeed
    repeated RequestOp success = 2;
    // List of operations to perform when any test fails
    repeated RequestOp failure = 3;
}

message Compare {

    enum CompareResult {
        EQUAL = 0;
        GREATER = 1;
        LESS = 2;
        NOT_EQUAL = 3;
    }

    enum CompareTarget {
        VERSION = 0;
        CREATE = 1;
        MOD = 2;
        VALUE = 3;
        LEASE = 4;
    }

    CompareResult result = 1;
    CompareTarget target = 2;

    bytes key = 3;
    oneof target_union {
        int64 version = 4;
        int64 create_revision = 5;
        int64 mod_revision = 6;
```

```
        bytes value = 7;
        int64 lease = 8;
    }

    // Can compare key range rather than just one key
    bytes range_end = 64;
}

message RequestOp {
    oneof request {
        RangeRequest request_range = 1;
        PutRequest request_put = 2;
        DeleteRangeRequest request_delete_range = 3;
        TxnRequest request_txn = 4;
    }
}

message TxnResponse {
    ResponseHeader header = 1;
    bool succeeded = 2;
    repeated ResponseOp responses = 3;
}
```

The `Watch` interface returns a stream of key change events.

Watch Request

```
rpc Watch (stream WatchRequest) returns (stream WatchResponse) { }

message WatchRequest {
    oneof request_union {
        WatchCreateRequest create_request = 1;
        WatchCancelRequest cancel_request = 2;
        WatchProgressRequest progress_request = 3;
    }
}

message WatchCreateRequest {

    enum FilterType {
        NOPUT = 0;
        NODELETE = 1;
    }

    bytes key = 1;
    bytes range_end = 2;
    int64 start_revision = 3;

    // Request keepalive notifications
    bool progress_notify = 4;

    repeated FilterType filters = 5;

    bool prev_kv = 6;
    int64 watch_id = 7;
    bool fragment = 8;
}

message WatchResponse {
    ResponseHeader header = 1;
    int64 watch_id = 2;
    bool created = 3;
    bool canceled = 4;

    // Indicates attempt to watch already compacted revision
    int64 compact_revision = 5;

    string cancel_reason = 6;

    bool fragment = 7;
```

```
    repeated Event events = 11;
}

message Event {

    enum EventType {
        PUT = 0;
        DELETE = 1;
    }

    EventType type = 1;
    KeyValue kv = 2;
    KeyValue prev_kv = 3;
}
```

- events reported before cluster consensus

The `Lease` interface monitors client liveness. Entries can be associated with leases, such entries are deleted on lease expiration.

Lease Request

```
rpc LeaseGrant (LeaseGrantRequest) returns (LeaseGrantResponse) { }

message LeaseGrantRequest {
    // Advisory time to live in seconds
    int64 TTL = 1;
    int64 ID = 2;
}

message LeaseGrantResponse {
    ResponseHeader header = 1;
    int64 ID = 2;
    int64 TTL = 3;
    string error = 4;
}
```

- also lease revoke request
- also lease keep alive request
- also lease time to live query request

Lock Request

```
rpc Lock (LockRequest) returns (LockResponse) { }
rpc Unlock (UnlockRequest) returns (UnlockResponse) { }

message LockRequest {
    bytes name = 1;
    int64 lease = 2;
}

message LockResponse {
    ResponseHeader header = 1;
    bytes key = 2;
}

message UnlockRequest {
    bytes key = 1;
}

message UnlockResponse {
    ResponseHeader header = 1;
}
```

- lock `name` creates key `name/uuid`
- lock key must be checked in exclusive operations to ensure consistency

Leader Election Request

```
rpc Campaign (CampaignRequest) returns (CampaignResponse) { }
rpc Proclaim (ProclaimRequest) returns (ProclaimResponse) { }
rpc Leader (LeaderRequest) returns (LeaderResponse) { }
rpc Observe (LeaderRequest) returns (stream LeaderResponse) { }
rpc Resign (ResignRequest) returns (ResignResponse) { }

message CampaignRequest {
    bytes name = 1;
    int64 lease = 2;
    bytes value = 3;
}

message CampaignResponse {
    ResponseHeader header = 1;
    LeaderKey leader = 2;
}

message LeaderKey {
    bytes name = 1;
    bytes key = 2;
    int64 rev = 3;
    int64 lease = 4;
}

message ProclaimRequest {
    LeaderKey leader = 1;
    bytes value = 2;
}

message ProclaimResponse {
    ResponseHeader header = 1;
}

message LeaderRequest {
    bytes name = 1;
}

message LeaderResponse {
    ResponseHeader header = 1;
    KeyValue kv = 2;
}
```

- leader proclaims shared value
- followers observe proclaimed value

Server cluster is either configured statically, with list of server addresses provided in server configuration or in SRV DNS records, or through server discovery, where another etcd server is used to store current server list. Target cluster size must be given but can be changed at runtime.

Experiments

```
# This assumes starting from revision 1

> etcdctl put somekey somevalue
OK
> etcdctl put anotherkey anothervalue
OK
> etcdctl get a b
anotherkey
anothervalue
> etcdctl get a z
anotherkey
anothervalue
somekey
somevalue
> etcdctl put somekey updatedvalue
```

```
OK

> etcdctl get --rev=1 a z
> etcdctl get --rev=2 a z
somekey
somevalue
> etcdctl get --rev=3 a z
anotherkey
anothervalue
somekey
somevalue
> etcdctl get --rev=4 a z
anotherkey
anothervalue
somekey
updatedvalue

> etcdctl lock somelock
somelock/536f4d65436f4465
# Launched from another window
> etcdctl get somelock somelocl
somelock/536f4d65436f4465
# Empty value
^C

> etcdctl elect someelection --listen
# Launched from another window
> etcdctl elect someelection someproposal
someelection/536f4d65436f4465
someproposal
# Launched from another window
> etcdctl elect someelection anotherproposal
# Waits until current leader resigns
...
```

2.4.1. References

1. Diego Ongaro, John Ousterhouth: In Search of an Understandable Consensus Algorithm. https://raft.github.io/raft.pdf

2.5. Felix

Felix is a platform for component applications that implements the OSGi standard with extensions.

iPOJO Service Requirement

```
// Service reference injected into field.
@Requires private LogService log;

// Service reference injected into constructor argument.
public MyComponent (@Requires LogService log) { ... }

// Service reference injected through method invocation.
@Bind public void bindLogService(LogService log) { ... }
@Unbind public void unbindLogService(LogService log) { ... }
@Modified public void modifiedLogService(LogService log) { ... }

// Example adjusted from documentation, see references.
```

iPOJO Service Provision

```
// Service provision with implicitly declared interfaces.
@Component @Provides
public class FooProvider implements FooService {
    ...
```

```
}

// Service provision with explicitly declared interfaces.
@Component @Provides (specifications={FooService.class})
public class FooProvider implements FooService {
    ...

    // Public service property declaration.
    @ServiceProperty (name="foo", value="foo")
    private String aFoo;

    // Private component property declaration.
    @Property (name="bar", falue="bar")
    private String aBar;

    // Property change notification.
    @Updated public void updated (Dictionary properties) {
        ...
    }
}

// Example adjusted from documentation, see references.
```

iPOJO Lifecycle Management

```
// Component with requirements and lifecycle management.
@Component @Instantiate
public class FooComponent {
    @Requires private LogService log;

    @Validate private void start () {
        // Called when all instance requirements become available.
        ...
    }

    @Invalidate private void stop () {
        // Called when some instance requirement ceases being available.
        ...
    }

    // Setting controller field to false disables component instance.
    @Controller private boolean enabled;
}

// Example adjusted from documentation, see references.
```

2.6. FlatBuffers

FlatBuffers is a framework that generates accessor code for serialized messages described in platform independent message description language. Individual language bindings are provided for multiple languages including C++, Java, Python, JavaScript.

2.6.1. Schema Language

The schema language defines the root type of a buffer, and any user defined types. Basic types are integer and floating point numbers of common sizes and booleans. Constructed types include vectors of basic types (also strings), enums with given backing types, structures and tables.

Structures and tables are the basic object representations with named and typed fields. All fields in a structure are mandatory, a structure cannot be estended later. All fields in a table are optional unless explicitly declared as required, a table can be extended later by adding fields at the end.

FlatBuffers Schema Example

```
namespace SomeNamespace;

enum SomeEnum: int { One = 1, Two, Three }

struct SomeStructure {

    // Basic integer types.
    aByte: byte;
    aShort: short;
    anInt: int;
    aLong: long;
    anUnsignedShort: ushort;
    anUnsignedInt: uint;
    anUnsignedLong: ulong;

    // Basic float types.
    aFloat: float;
    aDouble: double;

    // Array field only supported in structures.
    anIntArray: [int: 123];
}

table SomeTable {

    // Optional fields.
    aByte: byte;
    anInt: int;
    aString: string;

    // Required field of non basic types.
    aString: string (required);
    enums: [SomeEnum] (required);
}

table AnotherTable {

    // Explicitly specified field identifiers.
    // Must form continuous range from 0.
    // Must be specified everywhere.
    anInt: int (id: 2);
    aLong: long (id: 0);
    aString: string (id: 1);

    // Fields with default values.
    aFloat: float = 0.0 (id: 3);
}

// Union types only for tables.
union SomeUnion { SomeTable, AnotherTable }

root_type SomeUnion;
```

- A spectrum of basic types
- Structures with mandatory fields.
- Extensible tables with optional fields.

2.6.2. References

1. The FlatBuffers Project Home Page. https://google.github.io/flatbuffers

2.7. gRPC

gRPC is a remote procedure call mechanism for heterogeneous environments. A platform independent message format description language, called Protocol Buffers, is used to describe the remotely accessible interfaces. Individual language bindings rely on Protocol Buffers to provide standard message encoding and add language specific invocation interfaces.

2.7.1. Interface Description Language

Protocol Buffers Message Specification Example

```
syntax = "proto3";

package org.example;

message SomeMessage {

    // Field identifiers reserved after message changes.
    reserved 8, 100;

    // Many integer types with specific encodings.
    int32 aMostlyPositiveInteger = 1;
    sint64 aSignedInteger = 2;
    uint64 anUnsignedInteger = 3;
    fixed32 anOftenBigUnsignedInteger = 4;
    sfixed32 anOftenBigSignedInteger = 5;

    // String always with UTF 8 encoding.
    string aString = 10;

    // Another message type.
    AnotherMessage aMessage = 111;

    // Variable length content supported.
    repeated string aStringList = 200;
    map <int32, string> aMap = 222;
}
```

- A spectrum of basic types
- Packages and nested types
- Fields can be repeated
- Fields are optional
- Explicit field identifiers for versioning

Protocol Buffer Service Specification Example

```
syntax = "proto3";

service AnInterface {
    rpc someMethod (SomeRequest) returns (SomeResponse) { }
    rpc secondMethod (SecondRequest) returns (stream SecondResponse) { }
    rpc thirdMethod (stream ThirdRequest) returns (ThirdResponse) { }
}

message SomeRequest { ... }
message SomeResponse { ... }
...
```

- Single or stream arguments
- Stream open during entire call

2.7.2. C++ Server Code Basics

C++ Server Implementation

Single Argument Method Implementation.

```
class MyService : public AnExampleService::Service {
    grpc.Status OneToOne (grpc.ServerContext *context,
        const AnExampleRequest *request, AnExampleResponse *response) {
```

```
        // Method implementation goes here ...

        return (grpc.Status::OK);
    }
    ...
}
```

Server Initialization.

```
MyService service;
grpc.ServerBuilder builder;
builder.AddListeningPort ("localhost:8888", grpc.InsecureServerCredentials ());
builder.RegisterService (&service);
std::unique_ptr<grpc.Server> server (builder.BuildAndStart ());

server->Wait ();
```

- Sync mode uses internal thread pool
- Async mode uses completion queues

Asynchronous Server Internals

This note looks at the example from https://github.com/grpc/grpc/tree/master/examples/cpp/helloworld in more detail, the code snippets were taken from gRPC 1.44. During initialization, the server requests that a completion queue is used with the server instance:

```
std::unique_ptr<ServerCompletionQueue> cq_;
ServerBuilder builder;

cq_ = builder.AddCompletionQueue ();
```

The completion queue is a thread safe object that one or more threads can query for events. The completion queue will deliver two events per remote procedure call, one when the call arrives and one when the call completes. The delivery of the events has to be requested explicitly, for the call arrival event through a generated asynchronous service object, for the call completion event through a template asynchronous writer object. In the code snippet, the `CallData` instances represent pending calls and the `this` reference serves as a tag that uniquely identifies the events:

```
class CallData {
    public:
        CallData (Greeter::AsyncService* service, ServerCompletionQueue* cq)
            : service_ (service), cq_ (cq), responder_ (&ctx_) ...
        ...
    private:
        ServerAsyncResponseWriter<HelloReply> responder_;
        ServerContext ctx_;
        HelloRequest request_;
        HelloReply reply_;
        ...


// Requesting call arrival event.
service_->RequestSayHello (&ctx_, &request_, &responder_, cq_, cq_, this);


// Requesting call termination event.
responder_.Finish(reply_, Status::OK, this);


// Waiting for (any) event.
void* tag;
bool ok;
while (true) {
    GPR_ASSERT (cq_->Next (&tag, &ok));
    GPR_ASSERT (ok);
```

```
        ...
    }
```

2.7.3. Java Server Code Basics

Java Server Implementation

Single Argument Method Implementation.

```
class MyService extends AnExampleServiceGrpc.AnExampleServiceImplBase {
    @Override public void OneToOne (
        AnExampleRequest request,
        io.grpc.stub.StreamObserver<AnExampleResponse> responseObserver) {

        // Method implementation goes here ...

        responseObserver.onNext (response);
        responseObserver.onCompleted ();
    }
    ...
}
```

Server Initialization.

```
io.grpc.Server server = io.grpc.ServerBuilder
    .forPort (8888).addService (new MyService ()).build ().start ();

server.awaitTermination ();
```

- Uses static cached thread pool by default
- Can use provided executor
- Can use transport thread

2.7.4. Python Server Code Basics

Python Server Implementation

Single Argument Method Implementation.

```
class MyServicer (AnExampleServiceServicer):
    def OneToOne (self, request, context):

        # Method implementation goes here ...

        return response
```

Server Initialization.

```
server = grpc.server (
    futures.ThreadPoolExecutor (
        max_workers = SERVER_THREAD_COUNT))
add_AnExampleServiceServicer_to_server (MyServicer (), server)
server.add_insecure_port ("localhost:8888")
server.start ()

server.wait_for_termination ()
```

Asynchronous Server Internals

This note looks at the example from https://github.com/grpc/grpc/tree/master/examples/python/helloworld in more detail, the code snippets were taken from gRPC 1.44. The implementation is an awaitable object:

```
class Greeter (helloworld_pb2_grpc.GreeterServicer):
    async def SayHello (self, request: helloworld_pb2.HelloRequest, context: grpc.aio.ServicerC
        return helloworld_pb2.HelloReply (...)
```

The initialization uses the asynchronous version of the server interface:

```
async def serve ():
    server = grpc.aio.server ()
    ...
    await server.start ()
    await server.wait_for_termination ()
```

The main thread then launches the awaitable:

```
asyncio.run (serve ())
```

2.7.5. C++ Client Code Basics

C++ Client Implementation

Client Initialization.

```
std::shared_ptr<grpc.Channel> channel = grpc.CreateChannel (
    "localhost:8888", grpc.InsecureChannelCredentials ());
```

Single Argument Method Call.

```
grpc.ClientContext context;
AnExampleResponse response;
std::shared_ptr<AnExampleService::Stub> stub = AnExampleService::NewStub (channel);
grpc.Status status = stub->OneToOne (&context, request, &response);
if (status.ok ()) {

    // Response available here ...

}
```

Asynchronous Client Internals

This note looks at the example from https://github.com/grpc/grpc/tree/master/examples/cpp/helloworld in more detail, the code snippets were taken from gRPC 1.44. During invocation, the client represents an asynchronous invocation with a template asynchronous reader object connected to a completion queue:

```
HelloRequest request;
ClientContext context;
CompletionQueue cq;

std::unique_ptr<ClientAsyncResponseReader<HelloReply>> rpc (stub_->PrepareAsyncSayHello (&contex
rpc->StartCall ();
```

The call completion event is delivered through the completion queue. Again, the delivery of the event has to be requested explicitly, the `1` serves as a tag that uniquely identifies the event:

```
rpc->Finish (&reply, &status, (void*) 1);
void* got_tag;
bool ok = false;
GPR_ASSERT (cq.Next (&got_tag, &ok));
GPR_ASSERT (got_tag == (void*) 1);
GPR_ASSERT (ok);
```

```
if (status.ok ()) {
    ...
}
```

2.7.6. Java Client Code Basics

Java Client Implementation

Client Initialization.

```
io.grpc.ManagedChannel channel = io.grpc.ManagedChannelBuilder
    .forAddress ("localhost", 8888)
    .usePlaintext (true)
    .build ();
```

Single Argument Method Call.

```
AnExampleServiceGrpc.AnExampleServiceBlockingStub stub =
    AnExampleServiceGrpc.newBlockingStub (channel);
AnExampleResponse response = stub.oneToOne (request);

// Response available here ...
```

2.7.7. Python Client Code Basics

Python Client Implementation

Client Initialization.

```
with grpc.insecure_channel ("localhost:8888") as channel:
```

Single Argument Method Call.

```
stub = AnExampleServiceStub (channel)
response = stub.OneToOne (request)

# Response available here ...
```

Asynchronous Client Internals

This note looks at the example from https://github.com/grpc/grpc/tree/master/examples/python/helloworld in more detail, the code snippets were taken from gRPC 1.44. The initialization uses the asynchronous version of the channel interface:

```
async def run ():
    async with grpc.aio.insecure_channel ('localhost:50051') as channel:
        ...
```

The invocation is asynchronous, the stub is asynchronous by virtue of being created on an asynchronous channel:

```
stub = helloworld_pb2_grpc.GreeterStub (channel)
response = await stub.SayHello (helloworld_pb2.HelloRequest (...))
```

The main thread then launches the awaitable:

```
asyncio.run (run ())
```

2.7.8. References

1. The gRPC Project Home Page. https://www.grpc.io

2.8. Hazelcast

Hazelcast is an in memory datastore cluster with replication and processing support.

Hazelcast Architecture

Topologies.

Embedded	Node is part of client (Java)
Client Server	Nodes are separate servers with connected clients (Java, Python, C++, C# ...)
Smart Client	Connects to all server nodes and distributes requests
Single Socket Client	Connects to one server node that mediates requests

Partitioning. Partitioned data structures split between nodes

- By default 271 partitions
- Per instance configurable backup copies
- Per instance configurable synchronization with backup copies
- Read of backup copies possible with reduced consistency guarantees

Nodes can form partition groups

- Used to distribute partitions across failure domains
- Can be derived from deployment architecture in cloud

Partitioning uses consistent hash algorithm

- Smart clients can communicate with relevant nodes directly
- Non partitioned data structures can specify partition manually

Hazelcast Member Discovery

Hazelcast supports multiple member discovery mechanisms. When cluster membership changes, partitions are migrated accordingly. Standard member discovery mechanisms include multicast, bootstrap from existing cluster member, discovery through shared registry (Active Directory, ZooKeeper, Consul, etcd) and discovery through resource enumeration (Amazon Elastic Cloud, Google Cloud Platform).

Distributed Collections

Map	distributed hash map with possible persistency
Set	distributed hash set with possible persistency
Multi Map	a hash map variant that supports multiple values per key
Replicated Map	a hash map variant that stores all entries everywhere
Queue	distributed blocking queue
List	ordered list stored on one node
Ring Buffer	distributed circular buffer

Event Journal	distributed map update journal
Cardinality Estimator	distributed set cardinality estimator

IMap Interface

```
public interface IMap <K,V> extends ConcurrentMap <K,V>, BaseMap<K,V>, Iterable <Map.Entry <K,V>> {

    // Synchronous access methods

    V get (Object key);
    Map <K,V> getAll (Set <K> keys);

    void set (K key, V value);
    void setAll (Map <? extends K, ? extends V> map);

    V put (K key, V value);
    V putIfAbsent (K key, V value);
    void putAll (Map <? extends K, ? extends V> map);

    V replace (K key, V value);
    boolean replace (K key, V oldValue, V newValue);

    V remove (Object key);
    boolean remove (Object key, Object value);
    void removeAll (Predicate <K,V> predicate);

    void delete (Object key);
    void clear ();

    boolean containsKey (Object key);
    boolean containsValue (Object value);

    Iterator <Entry <K,V>> iterator ();
    Iterator <Entry <K,V>> iterator (int fetchSize);

    Set <K> keySet ();
    Set <K> keySet (Predicate <K,V> predicate);
    Set <K> localKeySet ();
    Set <K> localKeySet (Predicate <K,V> predicate);
    Collection <V> values ();
    Collection <V> values (Predicate <K,V> predicate);
    Set <Map.Entry <K,V>> entrySet ();
    Set <Map.Entry <K,V>> entrySet (Predicate <K,V> predicate);

    // Entries can have limited lifetime

    boolean setTtl (K key, long ttl, TimeUnit timeunit);

    void set (K key, V value, long ttl, TimeUnit ttlUnit);
    void set (K key, V value, long ttl, TimeUnit ttlUnit, long maxIdle, TimeUnit maxIdleUnit);

    V put (K key, V value, long ttl, TimeUnit ttlUnit);
    V put (K key, V value, long ttl, TimeUnit ttlUnit, long maxIdle, TimeUnit maxIdleUnit);
    V putIfAbsent (K key, V value, long ttl, TimeUnit ttlUnit);
    V putIfAbsent (K key, V value, long ttl, TimeUnit ttlUnit, long maxIdle, TimeUnit maxIdleUnit);

    // Keys can be locked even if absent

    void lock (K key);
    void lock (K key, long leaseTime, TimeUnit timeUnit);
    boolean tryLock (K key);
    boolean tryLock (K key, long time, TimeUnit timeunit);
    boolean tryLock (K key, long time, TimeUnit timeunit, long leaseTime, TimeUnit leaseTimeunit);

    void unlock (K key);
    void forceUnlock (K key);

    boolean isLocked (K key);

    boolean tryPut (K key, V value, long timeout, TimeUnit timeunit);
```

```
boolean tryRemove (K key, long timeout, TimeUnit timeunit);

// Asynchronous access methods

CompletionStage <V> getAsync (K key);

CompletionStage <Void> setAsync (K key, V value);
CompletionStage <Void> setAsync (K key, V value, long ttl, TimeUnit ttlUnit);
CompletionStage <Void> setAsync (K key, V value, long ttl, TimeUnit ttlUnit, long maxIdle, Time
CompletionStage <Void> setAllAsync (Map <? extends K, ? extends V> map);

CompletionStage <V> putAsync (K key, V value);
CompletionStage <V> putAsync (K key, V value, long ttl, TimeUnit ttlUnit);
CompletionStage <V> putAsync (K key, V value, long ttl, TimeUnit ttlUnit, long maxIdle, TimeUni
CompletionStage <Void> putAllAsync (Map <? extends K, ? extends V> map);

CompletionStage <V> removeAsync (K key);

// Non transient entries can have backing store

void loadAll(boolean replaceExistingValues);
void loadAll(Set<K> keys, boolean replaceExistingValues);

boolean evict (K key);
void evictAll ();
void flush();

void putTransient (K key, V value, long ttl, TimeUnit ttlUnit);
void putTransient (K key, V value, long ttl, TimeUnit ttlUnit, long maxIdle, TimeUnit maxIdleUn

// Local and global state change listeners with predicate filtering are supported

UUID addLocalEntryListener (MapListener listener);
UUID addLocalEntryListener (MapListener listener, Predicate <K,V> predicate, boolean includeVal
UUID addLocalEntryListener (MapListener listener, Predicate <K,V> predicate, K key, boolean inc

UUID addEntryListener (MapListener listener, boolean includeValue);
UUID addEntryListener (MapListener listener, K key, boolean includeValue);
UUID addEntryListener (MapListener listener, Predicate <K,V> predicate, boolean includeValue);
UUID addEntryListener (MapListener listener, Predicate <K,V> predicate, K key, boolean includeV

boolean removeEntryListener (UUID id);

UUID addPartitionLostListener (MapPartitionLostListener listener);
boolean removePartitionLostListener (UUID id);

// Interceptors can modify or cancel operations

String addInterceptor (MapInterceptor interceptor);
boolean removeInterceptor (String id);

// Entry view provides entry access statistics

EntryView <K,V> getEntryView (K key);
LocalMapStats getLocalMapStats ();

// Distributed processing

<R> R executeOnKey (K key, EntryProcessor <K,V,R> entryProcessor);
<R> Map <K,R> executeOnKeys (Set<K> keys, EntryProcessor <K,V,R> entryProcessor);
<R> Map <K,R> executeOnEntries (EntryProcessor <K,V,R> entryProcessor);
<R> Map <K,R> executeOnEntries (EntryProcessor <K,V,R> entryProcessor, Predicate <K,V> predicat

<R> Collection <R> project (Projection <? super Map.Entry<K,V>,R> projection);
<R> Collection <R> project (Projection <? super Map.Entry<K,V>,R> projection, Predicate <K,V> p

<R> R aggregate (Aggregator <? super Map.Entry <K,V>,R> aggregator);
<R> R aggregate (Aggregator <? super Map.Entry <K,V>,R> aggregator, Predicate <K,V> predicate);

<R> CompletionStage <R> submitToKey (K key, EntryProcessor <K,V,R> entryProcessor);
<R> CompletionStage <Map<K,R>> submitToKeys (Set<K> keys, EntryProcessor <K,V,R> entryProcessor
```

```
    // Cache for continuous queries defined by predicates

    QueryCache <K,V> getQueryCache (String name);
    QueryCache <K,V> getQueryCache (String name, Predicate <K,V> predicate, boolean includeValue);
    QueryCache <K,V> getQueryCache (String name, MapListener listener, Predicate <K,V> predicate, b

    ...
}
```

Hazelcast Map SQL Query

Primitive entries in a map are accessible as an SQL table with key and value columns. Object entries in a map are accessible as an SQL table with a key column and value columns with object fields. Field access via SQL restricts permitted value serializers and does not support nested objects.

Distributed Communication

Topic	publish subscribe messaging pattern implementation • configurable for sender or total ordering • totally ordered sends through topic owner

Reliable Topic	publish subscribe with backup ring buffer
	• configurable slow consumer handling
	  • `DISCARD_OLDEST` or `DISCARD_NEWEST`
	  • `BLOCK`
	  • `ERROR`

Distributed Coordination

Lock	distributed recursive unfair lock
Semaphore	distributed semaphore
Atomic Long	distributed counter
Atomic Reference	distributed atomic object storage (not quite reference)
Positive Negative Counter	distributed counter with relaxed consistency
Countdown Latch	distributed counter with wait for zero support
ID Generator	cluster wide unique identifier generator (for long integers) • embeds node identity to avoid communication • provides rough time ordering (k-ordering)

Hazelcast Positive Negative Counter

The counter is a conflict free replicated type represented by an array of positive and negative updates aggregated per node. Each node can compute the current counter value by summing the array items, updates to the array do not conflict and therefore do not need immediate synchronization.

2.9. JGroups

JGroups is a middleware for reliable multicast communication in Java. JGroups provides both low level communication primitives, such as message transport and group membership, and high level communication functions, such as synchronous message exchange or distributed mutual exclusion. The architecture of JGroups is configurable to allow tailoring to application requirements.

2.9.1. Channels

The low level functions of the communication mechanism, such as group membership and message transport, are provided by channels.

Channel Class

```
public class JChannel implements Closeable {

    // Initialization accepts configuration options
    public JChannel ();
    public JChannel (String url);
    public JChannel (InputStream stream);

    // Join a group with a given name
    public void connect (String cluster);
    public String clusterName ();
    public void disconnect ();

    // View is the current list of members
    public View getView ();

    // Send a message to all or one group member.
    public void send (Message msg);
    public void send (Address dst, Object obj);
    public void send (Address dst, byte [] buf);
    public void send (Address dst, byte [] buf, int offset, int length);

    // Asynchronous notification about messages and membership is available
    public void setReceiver (Receiver r);
    public Receiver getReceiver ();

    ...
}
```

The addresses used in the channel methods are internal identifiers typically assigned by the transport protocol modules.

Receiver Interface

```
public interface Receiver {

    // Receive individual messages or batches of messages
    default void receive (Message msg) { ... }
    default void receive (MessageBatch batch) { ... }

    // Notification about membership view change
    default void viewAccepted (View new_view) { ... }

    // Notification to temporarily suspend sending messages
    default void block () { ... }
    default void unblock () { ... }

    // Group members can share state
    default void getState (OutputStream output) { ... }
    default void setState (InputStream input) { ... }
}
```

Message Classes

```
public interface Message ... {

    short BYTES_MSG   = 0,
        NIO_MSG       = 1,
        EMPTY_MSG     = 2,
```

```
        OBJ_MSG       = 3,
        LONG_MSG      = 4,
        COMPOSITE_MSG = 5,
        FRAG_MSG      = 6;

    short getType ();

    Address getDest ();
    Message setDest (Address new_dest);
    Address getSrc ();
    Message setSrc (Address new_src);

    // Headers are internal and interpreted by individual protocol modules
    Message putHeader (short id, Header hdr);
    <T extends Header> T getHeader (short id);
    Map<Short,Header> getHeaders ();

    // Flags are interpreted by individual protocol modules
    // Examples include disabling flow control or reliability
    short getFlags (boolean transient_flags);
    Message setFlag (short flag, boolean transient_flags);

    // Convenience methods on the interface
    // May not make sense for all message classes

    byte [] getArray ();
    int getOffset ();
    int getLength ();
    public Message setBuffer (byte [] b);
    Message setArray (byte [] b, int offset, int length);

    <T extends Object> T getObject ();
    Message setObject (Object obj);

    <T extends Object> T getPayload ();
    Message setPayload (Object pl);

    ...
}

public class BytesMessage ... {
    public BytesMessage (Address dest, byte [] array) { ... }
    public BytesMessage (Address dest, byte [] array, int offset, int length) { ... }
    ...
}

public class NioMessage ... {
    // Uses java.nio.ByteBuffer that can reduce copying overhead
    public NioMessage (Address dest, ByteBuffer buf) { ... }
    public ByteBuffer getBuf () { ... }
    public NioMessage setBuf (ByteBuffer b) { ... }
    ...
}

public class ObjectMessage ... {
    public ObjectMessage(Address dest, Object obj) {
    ...
}

public class CompositeMessage ... implements Iterable<Message> {
    public CompositeMessage (Address dest, Message ... messages) { ... }
    public CompositeMessage add (Message msg) { ... }
    public <T extends Message> T get (int index) { ... }
    public Iterator<Message> iterator () { ... }
    ...
}
```

2.9.2. Building Blocks

Somewhat inaptly named, building blocks use channels to provide high level functions of the communication mechanism, such as synchronous message exchange or group mutual exclusion.

2.9.3. Protocol Modules

A stack of protocol modules is used to implement various aspects of the reliable multicast communication mechanism.

The transport modules are responsible for transporting messages. The UDP module uses IP multicast to deliver multicast messages and IP unicast to deliver unicast messages. The TCP and TCP_NIO2 modules use a mesh of TCP connections to deliver both multicast and unicast messages, with thread per connection and asynchronous single thread models. The TUNNEL module can tunnel other transport to a specialized router.

Transport Protocol Modules

UDP	uses IP multicast to deliver multicast messages
TCP	uses mesh of TCP connections, thread per connection model
TCP_NIO2	uses mesh of TCP connections, asynchronous single thread model
TUNNEL	tunnels transport to specialized router

The discovery modules are responsible for locating the group upon initialization. The PING, MPING and BPING modules use IP multicast or IP broadcast over UDP. The TCPPING module attempts to contact members from a given list. The TCPGOSSIP module attempts to contact members using a specialized router. The FILE_PING, JDBC_PING, RACKSPACE_PING, SWIFT_PING and S3_PING keep track of members in various places ranging from shared file systems and shared database tables to cloud storage services. The DNS_PING module relies on A and SRV records in DNS. The PDC module provides persistent cache of discovered members.

Discovery Protocol Modules

PING	uses IP multicast over existing UDP transport
MPING	uses IP multicast over separate UDP transport
BPING	uses IP broadcast
TCPPING	uses list of member addresses
TCPGOSSIP	uses specialized router
FILE_PING	uses shared directory to keep track of members
JDBC_PING	uses shared database to keep track of members
RACKSPACE_PING	uses Rackspace Cloud File Storage
SWIFT_PING	uses Openstack Swift object storage
S3_PING	uses Amazon Simple Storage Service
DNS_PING	uses A and SRV records in DNS
PDC	caches discovered members

The merge modules are responsible for merging groups during recovery from network partitioning failures. The MERGE2 module has group coordinators periodically multicast presence and membership information, distinct subgroups are merged upon discovery (versions 3.X only). The MERGE3 module has all members periodically multicast membership information hash, inconsistent membership information is retrieved and merged upon discovery.

Merge Protocol Modules

MERGE2	group coordinator multicasts presence and membership view (3.X)
MERGE3	all members multicast presence and membership view

The failure detection modules are responsible for detecting failed members. The FD module uses periodic ping with acknowledgment between neighboring members in a ring. The FD_ALL and FD_ALL2 modules use multicast heartbeat among all members in a group. The FD_SOCK module uses a TCP socket ring, socket close indicates suspect. The FD_HOST module augments member failure detection with host failure detection through internal library method (version 4.X only). The VERIFY_SUSPECT module provides additional verification of suspect members.

Failure Detection Modules

FD	uses periodic ping in logical ring
FD_ALL	uses multicast heartbeat
FD_ALL2	uses multicast heartbeat
FD_SOCK	uses TCP socket ring
FD_HOST	uses internal library method to ping hosts (4.X)
VERIFY_SUSPECT	verify suspect members additionally

The reliable message transmission modules are responsible for providing reliable ordered message delivery.

Reliable Message Transmission Modules

NAKACK	uses negative acknowledgments and sequence numbering, old version (3.X)
NAKACK2	uses negative acknowledgments and sequence numbering, new version
UNICAST	uses positive acknowledgments and sequence numbering, for unicast messages
UNICAST2	uses negative acknowledgments and sequence numbering, for unicast messages (3.X)
UNICAST3	uses both positive and negative acknowledgments and sequence numbering, for unicast messages (4.X)

Other modules provide functions such as authentication, encryption, compression, fragmentation, flow control, atomic delivery, totally ordered delivery, and other.

Miscellaneous Modules

UFC	rate limiting flow control for unicast
MFC	rate limiting flow control for multicast
FRAG	message fragmentation
FRAG2	message fragmentation (4.X)
STABLE	atomic delivery in group
BARRIER	helper for shared state transfer
SEQUENCER	totally ordered delivery through coordinator

AUTH	member authentication
ENCRYPT	message body encryption
COMPRESS	message body compression

2.9.4. References

1. The JGroups Project Home Page. https://www.jgroups.org

2.10. Java Message Service (JMS)

JMS (Java Message Service) is a standard interface for enterprise messaging middleware. JMS is a part of the J2EE platform, integrated with a wide spectrum of technologies including EJB (Enterprise Java Beans) and JNDI (Java Naming and Directory Interface). The JMS standard exists in two major revisions, 1.x and 2.x, the text in this section deals separately with the two versions where necessary.

2.10.1. Architecture

The architecture of JMS assumes an existence of an enterprise messaging service provider, which needs to be connected to before it can be used. The act of connecting can be as simple as initializing a local library, or as complex as connecting to a remote enterprise messaging service provider. The details are hidden from the client, who simply creates a connection using a connection factory obtained from JNDI.

Connection Creation Example

```
// Get an initial naming context
Context initialContext = new InitialContext ();

// Look up the connection factory using
// a well known name in the initial context
ConnectionFactory connectionFactory;
connectionFactory = (ConnectionFactory) initialContext.lookup ("ConnectionFactory");

// Create a connection using the factory
Connection connection;
connection = ConnectionFactory.createConnection ();

// A connection only delivers messages
// once it is explicitly started
connection.start ();
```

All enterprise messaging communication takes place within the context of a session. The session context keeps track of things such as ordering, listeners and transactions. A session and its resources - producers and consumers but not destinations - are restricted for use by a single thread at any particular time. Multiple sessions can be used to allow multiple threads to communicate concurrently, however, there is no support for concurrent processing of messages delivered to a single consumer.

Session Creation Example

```
// Create a session for a connection, requesting
// no transaction support and automatic message
// acknowledgement
Session session;
session = connection.createSession (false, Session.AUTO_ACKNOWLEDGE);
```

The simplified API (JMS 2.0 and above) introduces context objects, which represent a single session in a single connection. The threading model restrictions still apply.

Context Creation Example

```
// Create a context that includes a connection and a session.
// Use try with resources to close the context when done.
try (JMSContext context = connectionFactory.createContext ()) {
    // Create another context reusing the same connection.
    try (JMSContext another = context.createContext ()) {
        ...
    } catch (JMSRuntimeException ex) { ... }
} catch (JMSRuntimeException ex) { ... }
```

2.10.2. Destinations

`Destination` objects are used to represent addresses. The standard assumes destinations will be created in the messaging service configuration and registered in JNDI. The `Session` interface provides methods for creating destinations, however, these are only meant to convert textual addresses into destination objects. The textual address syntax is not standardized.

The standard distinguishes two types of destinations. A queue is a destination for point-to-point communication. A message sent to a queue is stored until it is received and thus consumed by one recipient. A topic is a destination for publish-subscribe communication. A message sent to a topic is distributed to all currently connected recipients.

Temporary queues and temporary topics, with a scope limited to a single connection, are also available.

Destination Creation Example

```
Queue oQueue = oSession.createQueue ("SomeQueueName");
Topic oTopic = oSession.createTopic ("SomeTopicName");

Queue oTemporaryQueue = oSession.createTemporaryQueue ();
Topic oTemporaryTopic = oSession.createTemporaryTopic ();
```

2.10.3. Messages

The messages consist of a header, properties, and a body. The header has a fixed structure with standard fields:

JMSMessageID	A unique message identifier generated by the middleware.
JMSCorrelationID	An optional message identifier of a related message.
JMSDestination	The message destination.
JMSReplyTo	An optional reply destination.
JMSType	The message type, understood only by the sender and the recipient.
JMSTimestamp	The message send time.
JMSExpiration	The message expiration time, computed from the send time and the message lifetime.
JMSDeliveryTime	The earliest delivery time, computed from the send time and the minimum message delivery delay.
JMSPriority	The message priority.
JMSDeliveryMode	The delivery mode, either transient or persistent.
JMSRedelivered	Indicates repeated delivery due to session recovery. The delivery count is reported in an associated property (JMS 2.0 and above).

Message Header

Set Directly By Sender.

JMSCorrelationID	correlated message identifier
JMSReplyTo	suggested reply destination
JMSType	message type understood by recipient

Set Indirectly By Sender.

JMSDestination	message recipient
JMSExpiration	message lifetime
JMSPriority	message priority
JMSDeliveryMode	PERSISTENT or NON_PERSISTENT
JMSDeliveryTime	earliest message delivery time

Set Automatically By Middleware.

JMSMessageID	unique message identifier
JMSTimestamp	message timestamp
JMSRedelivered	repeated delivery indication

Message properties are in fact optional message header fields. Properties are stored as name-value pairs with typed access interface. The standard reserves a unique name prefix for certain typical properties. These include for example user and application identity or current transaction context.

Messages can be filtered based on the value of message properties. The filters are specified using simple conditional expressions called message selectors.

The message body takes one of five shapes derived from the message type, namely `BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage`, `TextMessage`.

Message Body Types

StreamMessage	stream of primitive types
MapMessage	set of named values
TextMessage	java.lang.String
ObjectMessage	serializable object
BytesMessage	byte array

2.10.4. Producers and Consumers

The messages are sent by message producers and received by message consumers. The classic interfaces are `MessageProducer` and `MessageConsumer`, created by calling the appropriate session methods.

Producer And Consumer Creation Example

```
// Uses the classic API.

MessageProducer sender;
MessageConsumer recipient;

sender = session.createProducer (oQueue);
recipient = session.createConsumer (oQueue);
```

The simplified API interfaces to producers and consumers (JMS 2.0 and above) are `JMSProducer` and `JMSConsumer`, created by calling the appropriate context methods.

Producer And Consumer Creation Example

```
// Uses the simplified API.

// Configure sender with method chaining.
// Sender is not bound to destination here.
JMSProducer sender = context.createProducer ().
                     setDeliveryMode (PERSISTENT).
                     setDeliveryDelay (1000).
                     setTimeToLive (10000);

JMSConsumer recipient = context.createConsumer (oQueue);
```

The interfaces to send messages support various degrees of blocking, termed as synchronous and asynchronous (JMS 2.0 and above) message send. The standard does not define any interface that would guarantee non blocking operation.

Synchronous Message Send Example

```
// Uses the classic API.

TextMessage message;

message = session.createTextMessage ();
message.setText ("Hello");

// Always blocks until message is sent.
sender.send (message);
```

Synchronous Message Send Example

```
// Uses the simplified API.
```

```
// By default blocks until message is sent.
// Overloaded versions for all body types exist.
sender.send (oQueue, "Hello");
```

The interface to receive messages supports both blocking and nonblocking operation, termed as synchronous and asynchronous message receive in the standard.

The use of nonblocking communication is strongly related to the session threading model. As soon as a message listener is registered for a session of an active connection, that session becomes reserved for the internal thread implementing that listener, and neither the session nor the producers and consumers of the session can be called from other threads. It is safe to call the session or the associated objects from within the message listener using the listener thread. Registering a completion listener does not reserve the session, however, it is not safe to call the session from within the completion listener if it can be called from other code at the same time.

Message Receive Example

```
// Uses the classic API.

TextMessage oMessage;

oMessage = (TextMessage) recipient.receive ();
oMessage = (TextMessage) recipient.receive (1000);
```

Message Listener Example

```
// Uses the classic API.

public class SomeListener implements MessageListener {
    public void onMessage (Message message) {
        ...
    }
}

SomeListener oListener = new SomeListener ();
recipient.setMessageListener (oListener);
```

Message Receive Example

```
// Uses the simplified API.

// Template versions for all body types exist.
String body = consumer.receiveBody (String.class);
```

Message filters can be associated with message consumers.

Message Filter Example

```
String selector;
MessageConsumer receiver;

selector = new String ("(SomeProperty = 1000)");
receiver = session.createConsumer (oQueue, selector);
```

To guarantee reliable delivery, messages need to be acknowledged. Each session provides a `recover` method that causes unacknowledged messages to be delivered again. The acknowledgment itself can be done either automatically upon message delivery or manually by calling the `acknowledge` method on the message. When transactions are used, acknowledgment is done as a part of commit and recovery as a part of rollback.

A durable subscription to a topic can be requested. The messaging service stores messages for durable subscriptions of temporarily disconnected recipients.

Durable Subscriber Example

```
session.createDurableSubscriber (oTopic,"DurableSubscriberName");
```

A shared subscription to a topic can be requested (JMS 2.0 and above). The messaging service delivers messages for shared subscriptions to one of the connected recipients to provide load balancing.

Shared Subscriber Example

```
MessageConsumer consumer;

consumer = session.createSharedConsumer (oQueue, "SharedSubscriberName");
```

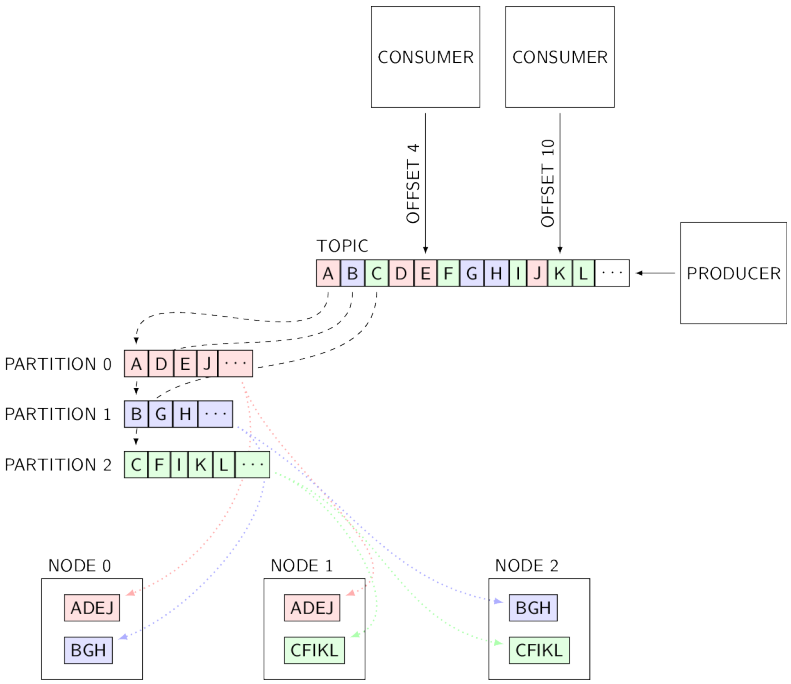
2.10.5. References

1. Java Message Service Specification. https://javaee.github.io/jms-spec
2. JSR 343: Java Message Service 2.0. https://jcp.org/aboutJava/communityprocess/final/jsr343/index.html

2.11. Apache Kafka

Apache Kafka is a distributed stream processing middleware.

Kafka Architecture



Data. Data streamed in topics

- Each data record is a key value pair
- Timestamps and additional headers supported

Topics split into partitions

- Each data record stored in one partition
- Record addressed by offset within partition

- Configurable assignment of records to partitions

Brokers. Replicated broker cluster

- Each broker stores data logs of some topic partitions
- Data log retention period configurable
- Partition replication configurable

Leader follower architecture

- Topic access done on leader broker
- Leader election in case of leader failure
- Producer may require minimum number of in sync replicas

Clients. Producers

- Can batch records when so configured
- Can guarantee exactly once delivery semantics
- Can wait for confirmation from zero, one or all in sync brokers

Consumers

- Each consumer maintains own topic position
- Consumer groups split topic partitions among themselves
- Can update topic position together with output in transaction

Stream processors

Kafka Producer Interface

```
public class KafkaProducer <K,V> implements Producer <K,V> {
    public KafkaProducer (Properties properties) { ... }

    public Future <RecordMetadata> send (ProducerRecord <K,V> record) { ... }
    public Future <RecordMetadata> send (ProducerRecord <K,V> record, Callback callback) { ... }

    public void flush () { ... }
    public void close () { ... }

    public void initTransactions () { ... }
    public void beginTransaction () { ... }
    public void abortTransaction () { ... }
    public void commitTransaction () { ... }

    // Introspection.
    public List <PartitionInfo> partitionsFor (String topic) { ... }

    ...
}

public class ProducerRecord <K,V> {
    public ProducerRecord (String topic, V value) { ... }
    public ProducerRecord (String topic, K key, V value) { ... }
    public ProducerRecord (
        String topic, Integer partition, K key, V value) { ... }
    public ProducerRecord (
        String topic, Integer partition, K key, V value, Iterable <Header> headers) { ... }
    public ProducerRecord (
        String topic, Integer partition, Long timestamp, K key, V value, Iterable <Header> headers)

    public K key () { ... }
    public V value () { ... }
    public Headers headers () { ... }

    ...
}
```

```
public interface Header {
    String key ();
    byte [] value ();
}

public final class RecordMetadata {
    public boolean hasOffset () { ... }
    public long offset () { ... }
    public boolean hasTimestamp () { ... }
    public long timestamp () { ... }

    public String topic () { ... }
    public int partition () { ... }

    public int serializedKeySize () { ... }
    public int serializedValueSize () { ... }
}

public interface Callback {
    void onCompletion (RecordMetadata metadata, Exception exception);
}
```

Kafka Consumer Interface

```
public class KafkaConsumer <K,V> implements Consumer <K,V> {
    public KafkaConsumer (Properties properties) { ... }

    // Statically assigned topics and partitions.
    public void assign (Collection <TopicPartition> partitions) { ... }
    public Set <TopicPartition> assignment () { ... }

    // Specific topics with dynamically assigned partitions.
    public void subscribe (Collection <String> topics) { ... }
    public void subscribe (Collection <String> topics, ConsumerRebalanceListener listener) { ... }

    // Regular expression topics with dynamically assigned partitions.
    public void subscribe (Pattern pattern) { ... }
    public void subscribe (Pattern pattern, ConsumerRebalanceListener listener) { ... }

    public void unsubscribe() { ... }
    public Set <String> subscription () { ... }

    // Poll for records.
    public ConsumerRecords <K,V> poll (final Duration timeout) { ... }

    // Seek and query position in topic partitions.
    public void seek (TopicPartition partition, long offset) { ... }
    public void seek (TopicPartition partition, OffsetAndMetadata offsetAndMetadata) { ... }
    public void seekToEnd (Collection <TopicPartition> partitions) { ... }
    public void seekToBeginning (Collection <TopicPartition> partitions) { ... }

    public long position (TopicPartition partition) { ... }
    public long position (TopicPartition partition, final Duration timeout) { ... }

    // Set and query committed position in topic partitions.
    public void commitSync () { ... }
    public void commitSync (Duration timeout) { ... }
    public void commitSync (final Map <TopicPartition, OffsetAndMetadata> offsets) { ... }
    public void commitSync (final Map <TopicPartition, OffsetAndMetadata> offsets, final Duration t
    public void commitAsync () { ... }
    public void commitAsync (OffsetCommitCallback callback) { ... }

    public OffsetAndMetadata committed (TopicPartition partition) { ... }
    public OffsetAndMetadata committed (TopicPartition partition, final Duration timeout) { ... }

    public void pause (Collection<TopicPartition> partitions) { ... }
    public void resume (Collection<TopicPartition> partitions) { ... }
    public void close () { ... }

    // Introspection.
```

```
    public Map <String, List <PartitionInfo>> listTopics () { ... }
    public List <PartitionInfo> partitionsFor (String topic) { ... }

    ...
}

public class ConsumerRecord <K,V> {
    public String topic () { ... }
    public int partition () { ... }
    public long offset () { ... }
    public long timestamp () { ... }

    public K key () { ... }
    public V value () { ... }
    public Headers headers () { ... }

    public int serializedKeySize () { ... }
    public int serializedValueSize () { ... }

    ...
}
```

Consumer Position Tracking

By default, the consumer position is committed periodically, as directed by the `enable.auto.commit` and `auto.commit.interval.ms` configuration settings. Explicit position commit is also supported.

To achieve atomic message processing, the consumer position information is stored inside an internal system topic. Consumer position updates are thus in fact message publishing operations. A client whose inputs and outputs are messages can wrap the consumer position updates and the message publishing operations in a transaction.

Kafka KStream Interface

```
public interface KStream <K,V> {

    // Filter stream by predicate.
    KStream <K,V> filter (Predicate <? super K, ? super V> predicate);
    KStream <K,V> filterNot (Predicate <? super K, ? super V> predicate);

    // Replace key with new key.
    <KR> KStream <KR,V> selectKey (KeyValueMapper <? super K, ? super V, ? extends KR> mapper);

    // Map entry to new entry.
    <KR,VR> KStream <KR,VR> map (KeyValueMapper <
        ? super K, ? super V,
        ? extends KeyValue <? extends KR, ? extends VR>> mapper);

    // Map value to new value.
    <VR> KStream <K,VR> mapValues (ValueMapper <? super V, ? extends VR> mapper);

    // Map entry to multiple new entries.
    <KR,VR> KStream <KR,VR> flatMap (KeyValueMapper <
        ? super K, ? super V,
        ? extends Iterable <? extends KeyValue <? extends KR, ? extends VR>> mapper);

    // Map value to multiple new values.
    <VR> KStream <K,VR> flatMapValues (ValueMapper <? super V, ? extends Iterable <? extends VR>> m

    // Print entries.
    void print (Printed <K, V> printed);

    // Consume or peek at entries with action.
    void foreach (ForeachAction <? super K, ? super V> action);
    KStream <K,V> peek (ForeachAction <? super K, ? super V> action);

    // Split by predicate or merge a stream.
```

```
    KStream <K,V> [] branch (Predicate <? super K, ? super V> ... predicates);
    KStream <K,V> merge (KStream <K,V> stream);

    // Materialize a stream into a topic.
    KStream <K,V> through (String topic);
    void to (String topic);

    // Process a stream using a stateful processor.
    <KO,VO> KStream <KO,VO> process (
        ProcessorSupplier <? super K, ? super V, KO, VO> processorSupplier,
        String... stateStoreNames);
    <VO> KStream <K,VO> processValues (
        FixedKeyProcessorSupplier <? super K, ? super V, VO> processorSupplier,
        String... stateStoreNames);

    // Group entries in a stream.
    KGroupedStream <K,V> groupByKey ();
    <KR> KGroupedStream <KR,V> groupBy (KeyValueMapper <? super K, ? super V, KR> selector);

    // Join stream with another stream or table on key.
    // Operation on streams limited by join window.
    <VO,VR> KStream <K,VR> join (
        KStream <K, VO> otherStream,
        ValueJoiner <? super V, ? super VO, ? extends VR> joiner,
        JoinWindows windows);
    <VO,VR> KStream <K,VR> leftJoin (
        KStream <K, VO> otherStream,
        ValueJoiner <? super V, ? super VO, ? extends VR> joiner,
        JoinWindows windows);
    <VO,VR> KStream <K,VR> outerJoin (
        KStream <K,VO> otherStream,
        ValueJoiner <? super V, ? super VO, ? extends VR> joiner,
        JoinWindows windows);
    <VT,VR> KStream <K,VR> join (
        KTable <K,VT> table,
        ValueJoiner <? super V, ? super VT, ? extends VR> joiner,
        Joined <K, V, VT> joined);
    <VT,VR> KStream <K,VR> leftJoin (
        KTable <K,VT> table,
        ValueJoiner <? super V, ? super VT, ? extends VR> joiner);

    ...
}
```

Kafka KGroupedStream Interface

```
public interface KGroupedStream <K,V> {

    KTable <K,Long> count ();
    KTable <K,V> reduce (Reducer <V> reducer);

    <VR> KTable <K,VR> aggregate (
        Initializer <VR> initializer,
        Aggregator <? super K, ? super V, VR> aggregator);

    <W extends Window> TimeWindowedKStream <K,V> windowedBy (Windows<W> windows);

    ...
}
```

Load Balancing Protocol

Correct stream processing requires that each partition is assigned to single client only. Client churn is handled by dynamically balancing partitions between clients.

2.11.1. References

1. The Apache Kafka Project Home Page. https://kafka.apache.org

2.12. Memcached

Memcached is a distributed memory caching system for uninterpreted arrays of bytes.

Architecture

- clients access data
  - data as key value pairs
  - key is a string (250 B limit)
  - value is array of bytes (1 MB limit)
  - client side compression supported
- servers cache data
  - standardized protocols
    - can use TCP or UDP
    - transparent binary protocol
    - text protocol limits keys and values
- servers do not know each other
- server selected by client side hashing
- least recently used strategy for eviction

Usage Example

```
// Initialization of memcache client.
mem = new Memcache ()
mem.add_server ("10.0.0.1:12345")
mem.add_server ("10.0.0.2:12345")
mem.add_server ("10.0.0.3:12345")


// Construct key for database query.
sql = "SELECT * FROM user WHERE name = ?"
key = "SQL:" + hash (sql) + name

// Try to fetch value from memcache.
if (defined (result = mem.get (key))) return (result)

// Fetch value from database and populate memcache otherwise.
result = execute_query (sql, name)
mem.set (key, result, lifetime)
return (result)


// Example adjusted from documentation, see references.
```

- getting and setting values of keys
- atomic setting of values for new keys
- incrementing and decrementing of integer values
- invalidating values by keys

2.12.1. References

1. The Memcached Website. http://www.memcached.org

2.13. Message Passing Interface (MPI)

MPI (Message Passing Interface) is a standard interface for high performance computing middleware. MPI uses typed messages and provides a variety of message exchange functions for point-to-point communication, collective communication functions designed for coordinated multiparty message exchange, optionally optimized for particular communication topologies, remote memory access

functions, and additional functions for coordinated I/O. The standard defines the interface in a language agnostic notation, language bindings are defined for C and Fortran.

2.13.1. Architecture

Initialization assumes either a world model or a session model. The world model assumes a flat application architecture where all processes interact. The session model assumes a modular application architecture where communication is restricted to processes executing individual modules. Cluster resources in session model are identified using URI names configured by cluster administrator.

MPI Initialization

World Model. For programs where all processes coordinate together

```
int MPI_Init (int *argc, char ***argv);
int MPI_Init_thread (int *argc, char ***argv, int required, int *provided);
int MPI_Query_thread (int *provided);
int MPI_Is_thread_main (int *flag);
int MPI_Initialized (int *flag);
int MPI_Finalize (void);
```

MPI_THREAD_SIGNLE	single thread in this process
MPI_THREAD_FUNNELED	multiple threads but only main thread calls MPI
MPI_THREAD_SERIALIZED	multiple threads but only one MPI call at a time
MPI_THREAD_MULTIPLE	multiple threads and multiple MPI calls at a time

Session Model. For programs where processes coordinate within components

```
int MPI_Session_init (MPI_Info info, MPI_Errhandler errhandler, MPI_Session *session);
int MPI_Session_get_num_psets (MPI_Session session, MPI_Info info, int *npset_names);
int MPI_Session_get_nth_pset (MPI_Session session, MPI_Info info, int n, int *pset_len, char *pset_
int MPI_Session_finalize (MPI_Session *session);
```

Process sets are implementation defined groups of processes

- arbitrary overlap possible
- intended to express shared resource scopes
- `mpi://SELF` and `mpi://WORLD` always exist

Dynamic Process Model. For explicit control over process lifecycle

```
int MPI_Comm_spawn (
    const char *command, char *argv[], int maxprocs, MPI_Info info,
    int root, MPI_Comm comm, MPI_Comm *intercomm,
    int array_of_errcodes []);

int MPI_Comm_spawn_multiple (
    int count, char *array_of_commands [], char **array_of_argv [],
    const int array_of_maxprocs [], const MPI_Info array_of_info [],
    int root, MPI_Comm comm, MPI_Comm *intercomm,
    int array_of_errcodes [])

int MPI_Comm_get_parent (MPI_Comm *parent);
```

- children have separate `MPI_COMM_WORLD`

Configuration. A set of key value pairs used to provide additional configuration

```
int MPI_Info_create (MPI_Info *info);
int MPI_Info_free (MPI_Info *info);
int MPI_Info_dup (MPI_Info info, MPI_Info *newinfo);
```

```
int MPI_Info_set (MPI_Info info, const char *key, const char *value);
int MPI_Info_get_nkeys (MPI_Info info, int *nkeys);
int MPI_Info_get_nthkey (MPI_Info info, int n, char *key);
int MPI_Info_get_string (MPI_Info info, const char *key, int *buflen, char *value, int *flag);
int MPI_Info_delete (MPI_Info info, const char *key);
```

MPI Addressing

Groups. A group contains processes belonging to an application

```
MPI_GROUP_EMPTY

int MPI_Group_size (MPI_Group group, int *size);
int MPI_Group_rank (MPI_Group group, int *rank);

int MPI_Group_translate_ranks (
    MPI_Group group1, int n, const int ranks1 [],
    MPI_Group group2, int ranks2 []);

int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int *result);

int MPI_Group_union (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
int MPI_Group_intersection (MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);

int MPI_Group_incl (MPI_Group group, int n, const int ranks [], MPI_Group *newgroup);
int MPI_Group_excl (MPI_Group group, int n, const int ranks [], MPI_Group *newgroup);
int MPI_Group_range_incl (MPI_Group group, int n, int ranges [][3], MPI_Group *newgroup);
int MPI_Group_range_excl (MPI_Group group, int n, int ranges [][3], MPI_Group *newgroup);

int MPI_Comm_group (MPI_Comm comm, MPI_Group *group);

int MPI_Group_from_session_pset (MPI_Session session, const char *pset_name, MPI_Group *newgroup);

int MPI_Group_free (MPI_Group *group);
```

- created from other groups in the world model
- created from process sets in the session model
- processes addressed using rank from `0` to `size - 1`

Communicators. A communicator represents one or two groups in communication context

```
MPI_COMM_SELF
MPI_COMM_WORLD

int MPI_Comm_size (MPI_Comm comm, int *size);
int MPI_Comm_rank (MPI_Comm comm, int *rank);

int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int *result);

int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *newcomm);
int MPI_Comm_dup_with_info (MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm);

int MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
int MPI_Comm_create_group (MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm);
int MPI_Comm_create_from_group (MPI_Group group, const char *stringtag, MPI_Info info, MPI_Errhandl
int MPI_Intercomm_create (
    MPI_Comm local_comm, int local_leader,
    MPI_Comm peer_comm, int remote_leader,
    int tag, MPI_Comm *newintercomm);
int MPI_Intercomm_create_from_groups (
    MPI_Group local_group, int local_leader,
    MPI_Group remote_group, int remote_leader,
    const char *stringtag, MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newintercomm);

int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
int MPI_Comm_split_type (MPI_Comm comm, int split_type, int key, MPI_Info info, MPI_Comm *newcomm);

int MPI_Intercomm_merge (MPI_Comm intercomm, int high, MPI_Comm *newintracomm);
```

```
int MPI_Comm_free (MPI_Comm *comm);

int MPI_Comm_set_info (MPI_Comm comm, MPI_Info info);
int MPI_Comm_get_info (MPI_Comm comm, MPI_Info *info_used);

int MPI_Comm_test_inter (MPI_Comm comm, int *flag);
int MPI_Comm_remote_size (MPI_Comm comm, int *size);
int MPI_Comm_remote_group (MPI_Comm comm, MPI_Group *group);
```

inter-communicator	communication within single group
intra-communicator	communication between two groups

2.13.2. Point-To-Point Communication

MPI_Send Function

```
int MPI_Send (
    const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm);
int MPI_Send_c (
    const void *buf, MPI_Count count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm);
```

buf	address of send buffer
count	number of elements in send buffer
datatype	datatype of each send buffer element
dest	rank of destination
tag	message tag
comm	communicator

MPI_Recv Function

```
int MPI_Recv (
    void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm,
    MPI_Status *status);
int MPI_Recv_c (
    void *buf, MPI_Count count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm,
    MPI_Status *status);
```

buf	address of receive buffer
count	maximum number of elements in receive buffer
datatype	datatype of each receive buffer element
source	rank of source or `MPI_ANY_SOURCE`
tag	message tag or `MPI_ANY_TAG`
comm	communicator
status	status object

```
int MPI_Get_count (const MPI_Status *status, MPI_Datatype datatype, int *count);
int MPI_Get_count_c (const MPI_Status *status, MPI_Datatype datatype, MPI_Count *count);
```

MPI_Sendrecv Function

```
int MPI_Sendrecv (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int source, int recvtag, MPI_Comm comm,
    MPI_Status *status);
```

```
int MPI_Sendrecv_c (
    const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype,
    int source, int recvtag, MPI_Comm comm,
    MPI_Status *status);
```

Point-To-Point Communication Modes

```
int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Bsend (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Ssend (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Rsend (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Buffer_attach (void *buffer, int size);
int MPI_Buffer_attach_c (void *buffer, MPI_Count size);

int MPI_Buffer_detach (void *buffer_addr, int *size);
int MPI_Buffer_detach_c (void *buffer_addr, MPI_Count *size);
```

Send	may block, buffer available on return, asynchronous
BSend	does not block, uses supplied buffers, buffer available on return, asynchronous
SSend	may block, target must be receiving otherwise function fails
RSend	may block, target must be receiving otherwise undefined

```
int MPI_Isend (
    const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm,
    MPI_Request *request);
int MPI_Isend_c (
    const void *buf, MPI_Count count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm,
    MPI_Request *request);

int MPI_Ibsend (...);
int MPI_Issend (...);
int MPI_Irsend (...);

int MPI_Irecv (
    void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm,
    MPI_Request *request);
int MPI_Irecv_c (
    void *buf, MPI_Count count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm,
    MPI_Request *request);

int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
int MPI_Improbe (int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *s
int MPI_Imrecv (void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *req
int MPI_Imrecv_c (void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Message *message, MPI_Requ

int MPI_Wait (MPI_Request *request, MPI_Status *status);
int MPI_Waitany (int count, MPI_Request array_of_requests [], int *index, MPI_Status *status);
int MPI_Waitall (int count, MPI_Request array_of_requests [], MPI_Status array_of_statuses []);
int MPI_Waitsome (int incount, MPI_Request array_of_requests [], int *outcount, int array_of_indice

int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Testany (int count, MPI_Request array_of_requests [], int *index, int *flag, MPI_Status *st
int MPI_Testall (int count, MPI_Request array_of_requests [], int *flag, MPI_Status array_of_status
int MPI_Testsome (int incount, MPI_Request array_of_requests [], int *outcount, int array_of_indice

int MPI_Request_free (MPI_Request *request);

int MPI_Request_get_status (MPI_Request request, int *flag, MPI_Status *status);

int MPI_Cancel (MPI_Request *request);
```

```
int MPI_Psend_init (
    const void *buf, int partitions, MPI_Count count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Info info,
    MPI_Request *request);

int MPI_Precv_init (
    void *buf, int partitions, MPI_Count count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Info info,
    MPI_Request *request);

int MPI_Start (MPI_Request *request);

int MPI_Pready (int partition, MPI_Request request);
int MPI_Pready_range (int partition_low, int partition_high, MPI_Request request);
int MPI_Pready_list (int length, const int array_of_partitions [], MPI_Request request);

int MPI_Wait (...);
```

Data Types

Selected Basic Types.

MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG	signed integer types
MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG	unsigned integer types
MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE	floating point types
MPI_CHAR, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_WCHAR	character data types
MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T	exact size signed integer types
MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, MPI_UINT64_T	exact size unsigned integer types
MPI_BYTE	buffer with raw data
MPI_PACKED	buffer with packed data

Local Derived Types.

```
int MPI_Type_commit (MPI_Datatype *datatype);
int MPI_Type_free (MPI_Datatype *datatype);
int MPI_Type_dup (MPI_Datatype oldtype, MPI_Datatype *newtype);

int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *ne

int MPI_Type_indexed (
    int count,
    const int array_of_blocklengths [],
    const int array_of_displacements [],
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);

int MPI_Type_create_struct (
    int count,
    const int array_of_blocklengths [],
    const MPI_Aint array_of_displacements [],
    const MPI_Datatype array_of_types [],
    MPI_Datatype *newtype);

int MPI_Type_create_subarray (
    int ndims,
    const int array_of_sizes [], const int array_of_subsizes [],
    const int array_of_starts [],
    int order,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

- elements of basic types
- offset for each element
- also versions with byte offsets
- also functions for introspecting derived types
- also functions for data import and export in canonical format

MPI_ORDER_C	row major order
MPI_ORDER_FORTRAN	column major order

Distributed Derived Types.

```
int MPI_Type_create_darray (
    int size, int rank,
    int ndims,
    const int array_of_gsizes [],
    const int array_of_distribs [], const int array_of_dargs [],
    const int array_of_psizes [],
    int order,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

MPI_DISTRIBUTE_BLOCK	sequential block distribution (AAABBBCCC...)
MPI_DISTRIBUTE_CYCLIC	cyclic element distribution (ABC...ABC...ABC...)

Conversions.

- MPI types provided by communication parties must be the same
- representation conversion for portability is performed as necessary
- representation of derived types with byte offsets may not be portable

2.13.3. Collective Communication

Collective Communication Primitives

```
int MPI_Gather (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm);
int MPI_Gather_c(
    const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype,
    void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm);

int MPI_Gatherv (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, const int recvcounts [], const int displs [], MPI_Datatype recvtype,
    int root, MPI_Comm comm);
int MPI_Gatherv_c (...);

int MPI_Igather (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm,
    MPI_Request *request);
int MPI_Igather_c (...);

int MPI_Igatherv (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, const int recvcounts [], const int displs [],
    MPI_Datatype recvtype, int root, MPI_Comm comm,
    MPI_Request *request);
int MPI_Igatherv_c (...);

int MPI_Gather_init (
```

```
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm, MPI_Info info,
    MPI_Request *request);
int MPI_Gather_init_c (...);

int MPI_Gatherv_init (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, const int recvcounts [], const int displs [], MPI_Datatype recvtype,
    int root, MPI_Comm comm, MPI_Info info,
    MPI_Request *request);
int MPI_Gatherv_init_c (...);
```

Bcast	sender A, receivers A, A, A
Gather	senders A, B, C, receiver ABC
Scatter	sender ABC, receivers A, B, C
Allgather	senders A, B, C, receivers ABC, ABC, ABC
Alltoall	senders ABC, DEF, GHI, receivers ADG, BEH, CFI
Reduce	senders A, B, C, receiver A+B+C
Allreduce	senders A, B, C, receivers A+B+C, A+B+C, A+B+C
Reduce_scatter	senders ABC, DEF, GHI, receivers A+D+G, B+E+H, C+F+I
Scan	senders A, B, C, receivers A, A+B, A+B+C
Exscan	senders A, B, C, receivers N/A, A, A+B
Barrier	rendez vous

- semantics differ for intra and inter communication
- intra communication can use special argument for single buffer

Reduction Operations

```
MPI_SUM, MPI_PROD
MPI_MIN, MPI_MINLOC
MPI_MAX, MPI_MAXLOC
MPI_LAND, MPI_LOR, MPI_LXOR
MPI_BAND, MPI_BOR, MPI_BXOR

int MPI_Op_create (MPI_User_function *user_fn, int commute, MPI_Op *op);
int MPI_Op_free (MPI_Op *op);

typedef void MPI_User_function (void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);
```

- must be associative
- may be commutative

2.13.4. Virtual Process Topologies

Virtual Topology Creation

```
int MPI_Cart_create (
    MPI_Comm comm_old,
    int ndims, const int dims [], const int periods [],
    int reorder, MPI_Comm *comm_cart);

int MPI_Cart_sub (MPI_Comm comm, const int remain_dims [], MPI_Comm *newcomm);

int MPI_Graph_create(
    MPI_Comm comm_old,
    int nnodes, const int index [], const int edges [],
    int reorder, MPI_Comm *comm_graph);

int MPI_Dist_graph_create_adjacent (
    MPI_Comm comm_old,
    int indegree, const int sources [], const int sourceweights [],
    int outdegree, const int destinations [], const int destweights [],
```

```
    MPI_Info info, int reorder, MPI_Comm *comm_dist_graph);

int MPI_Dist_graph_create (
    MPI_Comm comm_old,
    int n, const int sources [], const int degrees [], const int destinations [], const int weights
    MPI_Info info, int reorder, MPI_Comm *comm_dist_graph);
```

cartesian	cartesian grid with optionally periodic dimensions
centralized graph	centralized graph definition with node degrees and flattened edge list
adjacent distributed graph	distributed graph definition which specifies incoming and outgoing edges at each node
general distributed graph	distributed graph definition which specifies arbitrary subset of edges at each node

Virtual Topology Addressing

```
int MPI_Cart_rank (MPI_Comm comm, const int coords [], int *rank);
int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int coords []);

int MPI_Cart_shift (MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest);

int MPI_Graph_neighbors (MPI_Comm comm, int rank, int maxneighbors, int neighbors []);

int MPI_Dist_graph_neighbors (
    MPI_Comm comm,
    int maxindegree, int sources [], int sourceweights [],
    int maxoutdegree, int destinations [], int destweights []);
```

Virtual Topology Communication

```
int MPI_Neighbor_allgather (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm);
int MPI_Neighbor_allgatherv (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, const int recvcounts [], const int displs [], MPI_Datatype recvtype,
    MPI_Comm comm);

int MPI_Neighbor_alltoall (
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm);
int MPI_Neighbor_alltoallv (
    const void *sendbuf, const int sendcounts [], const int sdispls [], MPI_Datatype sendtype,
    void *recvbuf, const int recvcounts [], const int rdispls [], MPI_Datatype recvtype,
    MPI_Comm comm);

int MPI_Neighbor_alltoallw (
    const void *sendbuf, const int sendcounts [], const MPI_Aint sdispls [], const MPI_Datatype sen
    void *recvbuf, const int recvcounts [], const MPI_Aint rdispls [], const MPI_Datatype recvtypes
    MPI_Comm comm);
```

2.13.5. Remote Memory Access

Memory Window Initialization

```
int MPI_Win_create (
    void *base, MPI_Aint size, int disp_unit,
    MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

```
int MPI_Win_create_c (
    void *base, MPI_Aint size, MPI_Aint disp_unit,
    MPI_Info info, MPI_Comm comm, MPI_Win *win);

int MPI_Win_allocate (
    MPI_Aint size, int disp_unit,
    MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);
int MPI_Win_allocate_c (
    MPI_Aint size, MPI_Aint disp_unit,
    MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);

int MPI_Win_allocate_shared (
    MPI_Aint size, int disp_unit,
    MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);
int MPI_Win_allocate_shared_c (
    MPI_Aint size, MPI_Aint disp_unit,
    MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);

int MPI_Win_create_dynamic (MPI_Info info, MPI_Comm comm, MPI_Win *win);
int MPI_Win_attach (MPI_Win win, void *base, MPI_Aint size);
int MPI_Win_detach (MPI_Win win, const void *base);

int MPI_Win_free (MPI_Win *win);
```

disp_unit unit size used in scaling remote offsets

- shared windows only possible when hardware architecture permits that
- dynamic windows permit dynamic attaching and detaching of memory with given size

Remote memory access requires synchronization. Passive target synchronization assumes the target does not explicitly participate in synchronization and the origin process explicitly delimits access epochs with calls to `MPI_Win_flush` or `MPI_Win_lock` and `MPI_Win_unlock`. Active target synchronization assumes the target makes data available by explicitly delimiting exposure epochs with calls to `MPI_Win_fence` or `MPI_Win_post` and `MPI_Win_wait`, and the origin process explicitly delimits access epochs with calls to `MPI_Win_fence` or `MPI_Win_start` and `MPI_Win_complete`.

Accessing Remote Memory

```
int MPI_Put (
    void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank, int target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win);

int MPI_Get (
    void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Win win);

int MPI_Accumulate (
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Op op,
    MPI_Win win);

int MPI_Get_accumulate (
    const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
    void *result_addr, int result_count, MPI_Datatype result_datatype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Op op,
    MPI_Win win);

int MPI_Fetch_and_op (
    const void *origin_addr, void *result_addr, MPI_Datatype datatype,
    int target_rank, MPI_Aint target_disp,
    MPI_Op op,
    MPI_Win win);

int MPI_Compare_and_swap (
    const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype,
```

```
    int target_rank, MPI_Aint target_disp,
    MPI_Win win);
```

- local access requires explicit synchronization

Passive Target Synchronization.

```
int MPI_Win_flush (int rank, MPI_Win win);
int MPI_Win_flush_all (MPI_Win win);
int MPI_Win_flush_local (int rank, MPI_Win win);
int MPI_Win_flush_local_all (MPI_Win win);

int MPI_Win_lock (int lock_type, int rank, int assert, MPI_Win win);
int MPI_Win_lock_all (int assert, MPI_Win win);
int MPI_Win_unlock (int rank, MPI_Win win);
int MPI_Win_unlock_all (MPI_Win win);
```

Active Target Synchronization.

```
int MPI_Win_fence (int assert, MPI_Win win);

int MPI_Win_start (MPI_Group group, int assert, MPI_Win win);
int MPI_Win_complete (MPI_Win win);

int MPI_Win_post (MPI_Group group, int assert, MPI_Win win);
int MPI_Win_wait (MPI_Win win);
```

- assert parameter specifies application hints for optimization
- starting and completing delineates epoch of remote window access
- posting and waiting delineates epoch of exposure to remote window access

2.13.6. References

1. The MPI Forum Home Page. https://www.mpi-forum.org

2. The MPICH Project Home Page. https://www.mpich.org

3. The OpenMPI Project Home Page. https://www.open-mpi.org

2.14. MessagePack (msgpack)

MessagePack is a binary serialization format with data model based on JSON. Individual language bindings are provided for multiple languages including C++, Java, Python, JavaScript.

2.14.1. References

1. The MessagePack Project Home Page. https://msgpack.org

2.15. Open Services Gateway Initiative (OSGi)

OSGi (Open Services Gateway Initiative) is a platform for component applications. An OSGi application consists of components called bundles that export and import packages and potentially also services. The OSGi platform provides means for resolving dependencies between bundles and controlling the lifecycle of bundles, and also implements some standard services.

2.15.1. Bundles

An OSGi bundle is a JAR file that contains a manifest file describing the bundle and the class files implementing the bundle, possibly with other resources the implementation might require. The

manifest file identifies the bundle, describes the interface of the bundle in terms of its exported and imported packages, and specifies the dependencies on other bundles for situations where package dependencies are not suitable.

For dependency specification purposes, bundles are identified by their names and versions. Names are unique strings that follow the common reverse domain naming conventions. Versions are triplet of integers with the usual major.minor.micro semantics. When installed, bundles are also assigned a unique numerical identifier.

Exported and imported packages are connected using class loader hierarchy. A bundle can only use code that it implements or imports. Other bundles can only use code that a bundle exports.

Once a bundle is installed and its dependencies resolved, it can be started and stopped. The framework starts a bundle before use and stops a bundle after use by calls to its activator interface. The bundle is provided with a bundle context that allows it to access the various framework functions as required.

OSGi Bundle States

```
interface Bundle {
    // Bundle state constants
    int UNINSTALLED = 0x00000001;
    int INSTALLED = 0x00000002;
    int RESOLVED = 0x00000004;
    int STARTING = 0x00000008;
    int STOPPING = 0x00000010;
    int ACTIVE = 0x00000020;

    int getState ();

    ...
}
```

OSGi Bundle Activator Interface

```
interface BundleActivator {
    void start (BundleContext context) throws Exception;
    void stop (BundleContext context) throws Exception;
}
```

OSGi Bundle Context Interface (Bundle Related)

```
interface BundleContext {
    // Access to framework properties
    String getProperty (String key);

    // Access to objects representing bundles
    Bundle getBundle ();
    Bundle getBundle (long id);
    Bundle getBundle (String location);
    Bundle [] getBundles ();

    // Support for bundle management
    Bundle installBundle (String location, InputStream input) throws BundleException;
    Bundle installBundle (String location) throws BundleException;

    // Support for bundle lifecycle notifications
    void addBundleListener (BundleListener listener);
    void removeBundleListener (BundleListener listener);

    // Support for framework event notifications
    void addFrameworkListener (FrameworkListener listener);
    void removeFrameworkListener (FrameworkListener listener);

    // Support for persistent storage
    File getDataFile (String filename);
```

```
    ...
}
```

2.15.2. Services

A bundle can dynamically register and unregister services. A service is identified by its interface and by arbitrary additional properties specified during service registration. The framework keeps track of available services and distributes events whenever service availability or service properties change.

OSGi Bundle Context Interface (Service Related)

```
interface BundleContext {

    ...

    // Support for service management
    ServiceRegistration registerService (String [] clazzes, Object service, Dictionary properties);
    ServiceRegistration registerService (String clazz, Object service, Dictionary properties);

    Filter createFilter (String filter) throws InvalidSyntaxException;
    ServiceReference [] getServiceReferences (String clazz, String filter) throws InvalidSyntaxExce
    ServiceReference [] getAllServiceReferences (String clazz, String filter) throws InvalidSyntaxE

    ServiceReference getServiceReference (String clazz);
    Object getService (ServiceReference reference);
    boolean ungetService (ServiceReference reference);

    // Support for service lifecycle notifications
    void addServiceListener (ServiceListener listener, String filter) throws InvalidSyntaxException
    void addServiceListener (ServiceListener listener);
    void removeServiceListener (ServiceListener listener);
}
```

Some services are standardized. Among framework related services are the Package Admin Service, Start Level Service, Permission Admin Service. Among general purpose services are the Log Service, HTTP Service, XML Parser Service.

2.16. Protocol Buffers (protobuf)

Protocol Buffers is a framework that generates serialization code for messages described in platform independent message description language. Individual language bindings are provided for multiple languages including C++, Java, Python, JavaScript.

Protocol Buffers exists in two major versions, 2 and 3, which differ in how messages are structured. Version 2 supports messages with both required and optional fields. Fields can have custom default values, and missing fields are distinguished from fields with default values. Version 3 considers all fields optional. Fields have standard default value of zero or null, fields with default values are serialized only with explicit presence tracking.

2.16.1. Message Description Language

The message description language defines each message as a set of fields. Each field has a type, a name, and a key, which identifies the field inside serialized messages. Standard fields are present in a message at most once, with explicit presence tracking when defined as optional. Repeated fields can be present an arbitrary number of times.

Protocol Buffers Message Specification Example

```
syntax = "proto3";
```

```
// File level options supported.
option optimize_for = SPEED;

message SomeMessage {

    // Field identifiers reserved after message changes.
    reserved 8, 100;

    // Many integer types with specific encodings.
    int32 aMostlyPositiveInteger = 1;
    sint64 aSignedInteger = 2;
    uint64 anUnsignedInteger = 3;
    fixed32 anOftenBigUnsignedInteger = 4;
    sfixed32 anOftenBigSignedInteger = 5;

    // String always with UTF 8 encoding.
    string aString = 10;

    // Another message type.
    AnotherMessage aMessage = 111;

    // Explicit presence tracking is optional.
    optional float aFloatWithPresenceTracking = 222;

    // Variable length content supported.
    repeated string aStringList = 333;
    map <int32, string> aMap = 444;

    // Field level options supported.
    int32 aDeprecatedInteger = 666 [deprecated = true];

    // Extension field range.
    extensions 1234 to 5678;
}

extend SomeMessage {
    // Extension field in extension field range.
    int32 anExtensionField = 1234;
}
```

- A spectrum of basic types
- Packages and nested types
- Fields can be repeated
- Fields can have presence tracked
- Explicit field identifiers for versioning
- Options tune code generation
- Extensions reserve fields

Historically, the `optional` modifier is somewhat misnamed. In version 2, fields were either required or optional. In version 3, fields are always optional, and the modifier merely indicates that the field presence is tracked.

Protocol Buffers Primitive Field Types

Integer Types.

(s)fixed(32|64)	Integers with fixed length encoding
(u)int(32|64)	Integers with variable length encoding
sint(32|64)	Integers with sign optimized variable length encoding

Floating Poing Types.

float	IEEE 754 32 bit float
double	IEEE 754 64 bit float

Additional Primitive Types.

bool	Boolean
bytes	Arbitrary sequence of bytes
string	Arbitrary sequence of UTF-8 characters

Protocol Buffers More Field Types

Oneof Type.

```
message AnExampleMessage {
    oneof some_oneof_field {
        int32 some_integer = 1;
        string some_string = 2;
    }
}
```

- Assigning one field clears others

Enum Type.

```
enum AnEnum {
    INITIAL = 0;
    RED = 1;
    BLUE = 2;
    GREEN = 3;
    WHATEVER = 8;
}
```

- Must include zero

Any Type.

```
import "google/protobuf/any.proto";
message AnExampleMessage {
    repeated google.protobuf.Any whatever = 8;
}
```

- Internally a type identifier and a value
- Type identifier is URI string
- Value is byte buffer

Map Type.

```
message AnExampleMessage {
    map<int32, string> keywords = 8;
}
```

See https://github.com/protocolbuffers/protobuf/blob/main/src/google/protobuf/descriptor.proto for the full list of available options that tune code generation. Options can be associated with a file, a type, or a field.

2.16.2. C++ Generated Code Basics

C++ Message Manipulation

Construction.

```
AnExampleMessage message;
AnExampleMessage message (another_message);
message.CopyFrom (another_message);
```

Singular Fields.

```
cout << message.some_integer ();
message.set_some_integer (1234);
if (message.has_optional_integer ()) {
    message.clear_optional_integer ();
}
```

Repeated Fields.

```
int size = messages.messages_size ();
const AnExampleMessage &message = messages.messages (1234);
AnExampleMessage *message = messages.mutable_messages (1234);
AnExampleMessage *message = messages.add_messages ();
```

Byte Array Serialization.

```
char buffer [BUFFER_SIZE];
message.SerializeToArray (buffer, sizeof (buffer));
message.ParseFromArray (buffer, sizeof (buffer));
```

Standard Stream Serialization.

```
message.SerializeToOstream (&stream);
message.ParseFromIstream (&stream);
```

2.16.3. Java Generated Code Basics

Java Message Manipulation

Construction.

```
AnExampleMessage.Builder messageBuilder;
messageBuilder = AnExampleMessage.newBuilder ();
messageBuilder = AnExampleMessage.newBuilder (another_message);
AnExampleMessage message = messageBulder.build ();
```

Singular Fields.

```
System.out.println (message.getSomeInteger ());
messageBuilder.setSomeInteger (1234);
if (message.hasOptionalInteger ()) {
    messageBuilder = message.toBuilder ();
    messageBuilder.clearOptionalInteger ();
}
```

Repeated Fields.

```
int size = messages.getMessagesCount ();
AnExampleMessage message = messages.getMessages (1234);
List<AnExampleMessage> messageList = messages.getMessagesList ();
messagesBuilder.addMessages (messageBuilder);
messagesBuilder.addMessages (message);
```

Byte Array Serialization.

```
byte [] buffer = message.toByteArray ();
try {
    AnExampleMessage message = AnExampleMessage.parseFrom (buffer);
} catch (InvalidProtocolBufferException e) {
    System.out.println (e);
}
```

Standard Stream Serialization.

```
message.writeTo (stream);
AnExampleMessage message = AnExampleMessage.parseFrom (stream);
```

2.16.4. Python Generated Code Basics

Python Message Manipulation

Construction.

```
message = AnExampleMessage ()
message.CopyFrom (another_message)
```

Singular Fields.

```
print (message.some_integer)
message.some_integer = 1234
if message.HasField ('optional_integer'):
    message.ClearField ('optional_integer')
```

Repeated Fields.

```
size = len (messages.messages)
message = messages.messages [1234]
message = messages.messages.add ()
```

Byte Array Serialization.

```
buffer = message.SerializeToString ()
message.ParseFromString (buffer)
message = AnExampleMessage.FromString (buffer)
```

Standard Stream Serialization.

```
file.write (message.SerializeToString ())
message.ParseFromString (file.read ())
AnExampleMessage.FromString (file.read ())
```

2.16.5. References

1. The protobuf Project Home Page. https://protobuf.dev

2.17. Redis

Redis is an in memory datastore with optional snapshot persistence and replication.

Interactive Experiments

Use the redis-cli command to interact with a local redis server.

```
> redis-cli
127.0.0.1:6379> help
...
```

2.17.1. Data Model

Redis server provides access to multiple databases identified by sequential numbers starting from zero. Each database is a key value store whose keys are binary safe strings and values can be one of strings, lists, sets, sorted sets, maps, streams, and hyper log log estimators.

Redis Data Model

Databases. A server can host multiple databases

- Identified by sequential numbers starting from zero
- Each database is a key value store
- Keys are binary safe strings
- Keys can have expiration time
- Values are typed

Value Types. Binary safe string

- Can also be interpreted as an integer or a float or a bitmap

List

- Ordered list of binary safe strings
- Atomic element removal from both ends

Set

- Set and sorted set of binary safe strings
- Sorted set keeps float score with each element

Hash

- A key value map of binary safe strings

Stream

- A stream of key value maps of binary safe strings
- Individual consumers query entries by timestamp
- Consumer groups can cooperate in processing

Hyper Log Log estimator

- Opaque type for estimating set cardinality

2.17.2. Database

A database can have an eviction policy, default is no eviction. Alternatives include random and LRU and TTL based policies working across either all keys or keys with expiration set.

The database content snapshot can be periodically written to disk, or an operation log can be written to disk and periodically compacted, or both.

Database Selection Commands

SELECT	select database to use
SWAPDB	swap databases
DBSIZE	get number of keys in database
MOVE	move existing key to another database
FLUSHALL	delete all keys from all databases
FLUSHDB	delete all keys from current database

General Data Access Commands

SCAN	iterate over existign keys
KEYS	list keys matching glob
RANDOMKEY	get random existing key (but not value)
EXISTS	test existence of a key

COPY	copy existing value to another key
RENAME	move existing value to another key
RENAMENX	... if target does not exist
DEL	delete a key
UNLINK	... with asynchronous memory reclaim

EXPIRE	set relative key expiration time
PEXPIRE	... in milliseconds
EXPIREAT	set absolute key expiration time
PEXPIREAT	... in milliseconds
TTL	get key expiration time
PTTL	... in milliseconds
PERSIST	remove key expiration time

DUMP	serialize value with checksum
RESTORE	... and restore serialized value
TOUCH	set last access time for eviction policies
TYPE	get key type (string, list, hash, zset, set, stream)

String Type Access Commands

GET	get the value associated with a key
GETRANGE	... or only some characters
GETDEL	... and delete the key
GETEX	... and set key expiration time

SET	set the value associated with a key
SETRANGE	... or only some characters
SETNX	... if target does not exist
SETEX	... and set the key expiration time
PSETEX	... in milliseconds
GETSET	... and return the previous value

MGET	get values for multiple keys
MSET	set values for multiple keys
MSETNX	... if targets do not exist

APPEND	append new value to the existing value
STRLEN	get the length of the existing value

GETBIT	get single bit
SETBIT	set single bit
BITCOUNT	count bits that are set
BITPOS	find first bit that is set or reset
BITOP	perform logical operation between multiple keys
BITFIELD	perform get or set or inc on subset of bits

INCR	interpret string as integer and increment
INCRBY	... by arbitrary value
DECR	interpret string as integer and decrement
DECRBY	... by arbitrary value
INCRBYFLOAT	interpret string as float and increment or decrement

Experiments With Strings

```
> redis-cli
127.0.0.1:6379> help @string
...
127.0.0.1:6379> SET somekey somevalue
OK
```

```
127.0.0.1:6379> GET somekey
"somevalue"
127.0.0.1:6379> GETRANGE somekey 3 6
"eval"
127.0.0.1:6379> GETSET somekey anothervalue
"somevalue"
127.0.0.1:6379> GETBIT somekey 15
(integer) 0
127.0.0.1:6379> SETBIT somekey 15 1
(integer) 0
127.0.0.1:6379> GET somekey
"aoothervalue"
127.0.0.1:6379> INCR somekey
(error) ERR value is not an integer or out of range
127.0.0.1:6379> SET somekey 123
OK
127.0.0.1:6379> INCR somekey
(integer) 124
127.0.0.1:6379> GET somekey
"124"
127.0.0.1:6379> SET somekey ephemeral PX 1
OK
127.0.0.1:6379> GET somekey
(nil)
...
```

Hash Type Access Commands

HSCAN	iterate over existing fields
HKEYS	list existing fields
HVALS	get existing values
HRANDFIELD	get random existing field (and value)
HEXISTS	test existence of a field
HLEN	get the number of existing fields
HDEL	delete a field

HGET	get the value of a field
HSET	set the value of a field
HSETNX	... if target does not exist
HGETALL	get all fields and values
HMGET	get multiple fields
HMSET	set multiple fields

HSTRLEN	get the length of a field value
HINCRBY	interpret field as integer and increment or decrement
HINCRBYFLOAT	interpret field as float and increment or decrement

Experiments With Hashes

```
> redis-cli
127.0.0.1:6379> help @hash
...
127.0.0.1:6379> HSET somekey somefield somevalue
(integer) 1
127.0.0.1:6379> HSET somekey anotherfield anothervalue
(integer) 1
127.0.0.1:6379> GET somekey
(error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> HGET somekey somefield
"somevalue"
127.0.0.1:6379> HGETALL somekey
1) "somefield"
2) "somevalue"
3) "anotherfield"
4) "anothervalue"
127.0.0.1:6379> HRANDFIELD somekey 1 WITHVALUES
```

```
1) "somefield"
2) "somevalue"
...
```

2.17.3. Publish Subscribe

Publish Subscribe Commands

SUBSCRIBE	subscribe to messages from given channels
PSUBSCRIBE	... with channels given by glob
UNSUBSCRIBE	unsubscribe from given channels
PUNSUBSCRIBE	... with channels given by glob

PUBLISH publish a message on a given channel

PUBSUB CHANNELS list channels with subscribers

- subscription restricts connection commands to pubsub
- publishing does not require prior channel connection
- pattern subscription therefore matches continuously
- key update notifications available on channels derived from key names

Experiments With Publish Subscribe

```
> redis-cli
127.0.0.1:6379> help @pubsub
...
127.0.0.1:6379> SUBSCRIBE somechannel
Reading messages... (press Ctrl-C to quit)
...
```

```
> redis-cli
127.0.0.1:6379> PUBSUB CHANNELS
1) "somechannel"
127.0.0.1:6379> PUBLISH somechannel somemessage
(integer) 1
...
```

2.17.4. Transactional Command Execution

Transactional Command Execution

MULTI	begin a transaction
EXEC	commit a transaction
DISCARD	abort a transaction

WATCH	changes to watched keys label transaction fail only
UNWATCH	discard watched keys

- results of all operations are collected during the transaction
- individual operation failures do not terminate the transaction

Experiments With Transactions

```
> redis-cli
127.0.0.1:6379> help @transactions
...
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> SET somekey somevalue
QUEUED
```

```
127.0.0.1:6379(TX)> SET anotherkey anothervalue
QUEUED
127.0.0.1:6379(TX)> EXEC
1) OK
2) OK
127.0.0.1:6379> WATCH somekey
OK
127.0.0.1:6379> GET somekey
"somevalue"
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> SET anotherkey somevalue
QUEUED
127.0.0.1:6379(TX)> EXEC
(nil)
127.0.0.1:6379> GET somekey
"interference"
...
```

2.17.5. Scripting

Scripting

EVAL	execute LUA code with arguments
EVALSHA	... or execute LUA code from cache
SCRIPT LOAD	explicitly load LUA code to cache
SCRIPT EXISTS	check presence of LUA code in cache
SCRIPT FLUSH	drop LUA code cache content

- script execution is atomic
- long running read only scripts can be terminated after timeout
- long running read write scripts cannot be terminated other than by shutdown

Experiments With Scripting

```
> redis-cli
127.0.0.1:6379> help @scripting
...
127.0.0.1:6379> SET somekey 123
OK
127.0.0.1:6379> SET anotherkey 456
OK
127.0.0.1:6379> EVAL "return tonumber (redis.call ('GET', KEYS [1])) + tonumber (redis.call ('GE
(integer) 579
...
```

2.17.6. References

1. The redis Website. http://www.redis.io

2.18. Java Remote Method Invocation (RMI)

Java RMI (Remote Method Invocation) is a remote procedure call mechanism integrated within the Java programming environment. Standard features of the Java environment are used both to define the remotely accessible interfaces and to serialize the invocation arguments.

2.18.1. Interface

The interface of a remotely accessible object is a standard Java interface, with several simple restrictions. A remotely accessible object must implement the Remote interface, which marks it as an object that can receive remote invocations. All remotely accessible methods must be able to throw the `RemoteException` exception. Only serializable types can be passed by value.

Remotely Accessible Type Example

```
public interface Example extends Remote {
    void printString (String text) throws RemoteException;
}
```

- Inheritance used to request passing by reference
- Serializable arguments
- Remote exception

The stubs are instances of the generic `Proxy` class that appear to implement all Remote interfaces of the server implementation. It is possible to cast between multiple remote interfaces of the same remote object. The stubs implement the `equals` and `hashCode` methods, making proxies that refer to the same remote object appear equal.

2.18.2. Implementation

To receive invocations, an instance of a remote object must first be exported. Exporting associates the object with a unique object identifier, which is registered by the RMI infrastructure and used by the stubs. A remote object can simply inherit from the `UnicastRemoteObject` class, which exports the object in the inherited constructor. Alternatively, a remote object can be registered by calling the `exportObject` method and unregistered by calling the `unexportObject` method, both provided by the `UnicastRemoteObject` class.

Remotely Accessible Object Example

```
public class ExampleImpl extends UnicastRemoteObject implements Example {
    public ExampleImpl () throws RemoteException { }
    public void printString (String text) { System.out.println (text); }
}
```

- Interface used to mark remotely accessible object
- Inheritance used to export the instance
- Constructor can return exception

exportObject Methods

```
static Remote exportObject (Remote obj, int port)
static boolean unexportObject (Remote obj, boolean force)
```

Passing an object by reference is done by passing its proxy by value. An object is passed by reference only when it is exported, objects that are not exported are passed by value even when they implement the `Remote` interface.

Serialization Exploits

The use of the default language serialization mechanism for remotely accessible interfaces poses security risks. A class can implement proprietary variants of the `readObject` and `writeObject` methods, used by the serialization mechanism to transfer state between object instances and serialization streams. See https://github.com/frohoff/ysoserial for examples of how these methods in common libraries can be tricked to perform remote code execution, and https://mogwailabs.de/en/blog/2019/03/attacking-java-rmi-services-after-jep-290 for more work after input stream filtering was introduced.

2.18.3. Threading

The specification explicitly makes no guarantees regarding the threading model. A common implementation requires exclusive use of both a thread and a connection during invocation, threads and connections are created on demand and potentially reused until collected.

2.18.4. Lifecycle

The lifecycle of instances is directed by a system of leases. A client that holds a remote reference must possess a lease from the server, the lease is renewed by the client periodically and returned by the client after the object reference is garbage collected locally. To avoid races when passing around references, the reference recipient will first request the lease from the target server, and then acknowledge to the reference sender, who keeps the reference locally alive at least until that moment. The lease duration is configured in system properties, default is 10 minutes. To prevent errors due to network delays, clients renew leases when half expired.

The Unreferenced interface can be used to receive a notification when all leases to an object expire. Exporting an object creates a weak reference. Leasing an object creates a strong reference. A remotely accessible object is therefore subject to garbage collection when it is neither leased remotely nor referenced locally.

Unreferenced Interface

```
public interface Unreferenced {
    void unreferenced ();
}
```

- Called some time after no client holds reference to remote object

2.18.5. Naming

Naming uses the rmiregistry server to register object references under string names and to look up the references using the names.

Naming Interface

```
class java.rmi.Naming {
    static void bind (String name, Remote obj);
    static void rebind (String name, Remote obj);
    static void unbind (String name);
    static Remote lookup (String name);
    static String [] list (String name);
}
```

Naturally, both the client and the server have to use the same instance of the rmiregistry server.

Naming Example

Server Side Registration.

```
ExampleImpl obj = new ExampleImpl ();
Naming.rebind ("//localhost/Example", obj);
```

Client Side Lookup.

```
Example object = (Example) Naming.lookup ("//localhost/Example");
object.printString ("Hello RMI !");
```

2.18.6. References

1. Oracle: The Java RMI Specification. https://docs.oracle.com/en/java/javase/13/docs/specs/rmi/index.html

2.19. Sun RPC

Sun RPC is a remote procedure call mechanism originally defined to support NFS. The mechanism defines an interface definition language and data encoding, together called External Data Representation (XDR).

Interface Definition Example

```
const MNTPATHLEN = 1024;    /* maximum bytes in a pathname argument */
const MNTNAMLEN = 255;      /* maximum bytes in a name argument */
const FHSIZE = 32;         /* size in bytes of a file handle */

typedef opaque fhandle [FHSIZE];
typedef string name <MNTNAMLEN>;
typedef string dirpath <MNTPATHLEN>;

union fhstatus switch (unsigned fhs_status) {
    case 0:
        fhandle fhs_fhandle;
    default:
        void;
};

typedef struct mountbody *mountlist;
struct mountbody {
    name ml_hostname;
    dirpath ml_directory;
    mountlist ml_next;
};

typedef struct groupnode *groups;
struct groupnode {
    name gr_name;
    groups gr_next;
};

typedef struct exportnode *exports;
struct exportnode {
    dirpath ex_dir;
    groups ex_groups;
    exports ex_next;
};

program MOUNTPROG {
    version MOUNTVERS {
        void  MOUNTPROC_NULL (void) = 0;
        fhstatus  MOUNTPROC_MNT (dirpath) = 1;
        mountlist  MOUNTPROC_DUMP (void) = 2;
        void MOUNTPROC_UMNT (dirpath) = 3;
        void MOUNTPROC_UMNTALL (void) = 4;
        exports MOUNTPROC_EXPORT (void) = 5;
        exports MOUNTPROC_EXPORTALL (void) = 6;
    } = 1;
} = 100005;
```

Each function is identified by a unique combination of service ID, version ID and function ID. Servers are registered in the RPC port mapper service, which provides standard registry features.

Portmapper Services Example

```
> rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100011    1   udp    892  rquotad
    100011    2   udp    892  rquotad
```

```
    100011    1   tcp    895  rquotad
    100011    2   tcp    895  rquotad
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100021    1   udp  39968  nlockmgr
    100021    3   udp  39968  nlockmgr
    100021    4   udp  39968  nlockmgr
    100005    1   udp  39969  mountd
    100005    1   tcp  45529  mountd
    100005    2   udp  39969  mountd
    100005    2   tcp  45529  mountd
    100005    3   udp  39969  mountd
    100005    3   tcp  45529  mountd
    100024    1   udp  39970  status
    100024    1   tcp  45530  status
    391002    2   tcp  45533  sgi_fam
```

2.19.1. References

1. RFC 5531: Remote Procedure Call Protocol Specification Version 2. https://tools.ietf.org/html/rfc5531

2.20. Apache Thrift

Apache Thrift is a remote procedure call mechanism for heterogeneous environments. A platform independent interface description language is used to describe the remotely accessible interfaces. The runtime environment supports multiple encodings over multiple transports.

2.20.1. Interface Description Language

Thrift Interface Specification Example

```
namespace cpp org.example
namespace java org.example

enum AnEnum {
    ONE = 1,
    TWO = 2,
    THREE = 3
}

struct SomeMessage {
    1: bool aBooleanField,
    2: i8 aByteField,
    3: i16 aShortIntField,
    4: i32 aNormalIntField,
    5: i64 aLongIntField,

    10: double aDoubleField,

    20: string aStringField,
    22: binary aBinaryField,

    // Fields can have default values.
    100: AnEnum anEnumFieldWithDefault = AnEnum.THREE,

    // Fields can be optional.
    200: optional i16 anOptionalIntField
}

// Exceptions are structures too.
exception SomeException {
    1: list<string> aStringList,
    2: set<i8> aByteSet,
    3: map<i16, string> aMap
}
```

```
service AnInterface {
    void ping (),
    bool aMethod (1: i16 argShort, 2: i64 argLong),
    oneway void aOnewayMethod (1: SomeMessage message)
}

service AnotherInterface extends AnInterface {
    void yetAnotherMethod (1: SomeMessage message) throws (1: SomeException ex)
}
```

- Namespaces per language
- A spectrum of basic types
- Containers for other types
- Fields can be optional
- Explicit field and argument identifiers for versioning
- Methods can be oneway
- Methods can throw exceptions

2.20.2. C++ Server Code Basics

C++ Server Implementation

Method Implementation.

```
class ExampleHandler : virtual public ExampleIf {
    void printString (const std::string &text) override {

        // Method implementation goes here ...

    }
    ...
}
```

Server Initialization.

```
// Handler is the user defined implementation.
std::shared_ptr<ExampleHandler> handler (new ExampleHandler ());

// Processor is responsible for decoding function arguments and invoking the handler.
std::shared_ptr<ExampleProcessor> processor (new ExampleProcessor (handler));

// Transport provides reading and writing of byte buffers.
std::shared_ptr<TServerTransport> transport (new TServerSocket (SERVER_PORT));

// Buffered transport is a wrapper for another transport object.
std::shared_ptr<TTransportFactory> transport_factory (new TBufferedTransportFactory ());

// Protocol provides reading and writing of individual types on top of transport.
std::shared_ptr<TProtocolFactory> protocol_factory (new TBinaryProtocolFactory ());

// New connections use their own transport and protocol instances hence the factories.
std::shared_ptr<TServer> server (new TSimpleServer (processor, transport, transport_factory, protoc

server->serve ();
```

- Public `read` and `write` methods on generated transport types
- Multiple transports available
  - Plain socket
  - Socket with SSL
  - Socket with HTTP
  - Socket with WebSocket
  - Wrapper for zlib compression
  - File and pipe
  - Memory buffer

- Multiple protocols available
  - JSON
  - Simple binary encoding
  - Compact binary encoding
  - Wrapper for serving multiple services
- Multiple servers available
  - Single main thread
  - Thread per connection
  - Thread pool with fixed size
  - Thread pool with fixed size and dedicated dispatcher

The file and memory buffer transports can be used for simple serialization without performing remote calls.

The simple binary encoding protocol stores primitive types in fixed length format with configurable byte ordering. Structure fields are stored with type and identifier also in fixed length format. Containers are similarly straightforward. See https://github.com/apache/thrift/blob/master/doc/specs/thrift-binary-protocol.md for details on the simple binary encoding protocol.

The compact binary encoding protocol stores primitive integer types in variable length format. Structure fields store identifiers as deltas where possible and use variable length format where not. Containers store type and size together for small enough sizes and use variable length format otherwise. See https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md for details on the compact binary encoding protocol.

Example Server Internals

This note looks at the example from https://github.com/d-iii-s/teaching-middleware/tree/master/src/thrift-basic-server in more detail, the code snippets were generated with Thrift 0.14. On the server side, the `ExampleHandler` class inherits from `ExampleIf`, a generated abstract class with the `printString` method that reflects the interface definition:

```
class ExampleIf {
    public:
        virtual ~ExampleIf () {}
        virtual void printString (const std::string& text) = 0;
};
```

The `ExampleIf` class is what the generated `ExampleProcessor` class calls to deliver method invocations:

```
void ExampleProcessor::process_printString (int32_t seqid, TProtocol* iprot, TProtocol* oprot, v

    ...

    Example_printString_args args;
    args.read (iprot);
    iprot->readMessageEnd ();
    uint32_t bytes = iprot->getTransport ()->readEnd ();

    ...

    Example_printString_result result;
    try {
        iface_->printString (args.text);
    } catch (const std::exception& e) {

        ...

        TApplicationException x (e.what ());
        oprot->writeMessageBegin ("printString", T_EXCEPTION, seqid);
        x.write (oprot);
        oprot->writeMessageEnd ();
        oprot->getTransport ()->writeEnd ();
```

```
            oprot->getTransport ()->flush ();
            return;
        }

        ...

        oprot->writeMessageBegin ("printString", T_REPLY, seqid);
        result.write (oprot);
        oprot->writeMessageEnd ();
        bytes = oprot->getTransport ()->writeEnd ();
        oprot->getTransport ()->flush ();

        ...
    }


    uint32_t Example_printString_args::read (TProtocol* iprot) {

        ...

        xfer += iprot->readStructBegin(fname);

        bool isset_text = false;

        while (true) {
            xfer += iprot->readFieldBegin (fname, ftype, fid);
            if (ftype == T_STOP) {
                break;
            }
            switch (fid) {
                case 1:
                    if (ftype == T_STRING) {
                        xfer += iprot->readString (this->text);
                        isset_text = true;
                    } else {
                        xfer += iprot->skip (ftype);
                    }
                    break;
                default:
                    xfer += iprot->skip(ftype);
                    break;
            }
            xfer += iprot->readFieldEnd ();
        }

        xfer += iprot->readStructEnd ();

        if (!isset_text) throw TProtocolException (INVALID_DATA);

        return xfer;
    }
```

2.20.3. C++ Client Code Basics

C++ Client Implementation

Client Initialization.

```
// Transport provides reading and writing of byte buffers.
std::shared_ptr<TTransport> socket (new TSocket (SERVER_ADDR, SERVER_PORT));

// Buffered transport is a wrapper for another transport object.
std::shared_ptr<TTransport> transport (new TBufferedTransport (socket));

// Protocol provides reading and writing of individual types on top of transport.
std::shared_ptr<TProtocol> protocol (new TBinaryProtocol (transport));
```

Method Call.

```
std::shared_ptr<ExampleClient> client (new ExampleClient (protocol));
client->printString ("Hello from Thrift in C++ !");
```

Example Client Internals

This note looks at the example from https://github.com/d-iii-s/teaching-middleware/tree/master/src/thrift-basic-server in more detail, the code snippets were generated with Thrift 0.14. On the client side, the `ExampleClient` class is a generated stub class with the `printString` method responsible for the remote method invocation:

```
class ExampleClient : virtual public ExampleIf {
    public:
        void printString (const std::string& text);
        void send_printString (const std::string& text);
        void recv_printString ();

    ...
};


void ExampleClient::send_printString (const std::string& text) {

    int32_t cseqid = 0;
    oprot_->writeMessageBegin ("printString", T_CALL, cseqid);

    Example_printString_pargs args;
    args.text = &text;
    args.write (oprot_);

    oprot_->writeMessageEnd ();
    oprot_->getTransport ()->writeEnd ();
    oprot_->getTransport ()->flush ();
}


void ExampleClient::recv_printString () {

    ...

    iprot_->readMessageBegin (fname, mtype, rseqid);

    if (mtype == T_EXCEPTION) {
        TApplicationException x;
        x.read (iprot_);
        iprot_->readMessageEnd ();
        iprot_->getTransport ()->readEnd ();
        throw x;
    }
    if (mtype != T_REPLY) {
        iprot_->skip (T_STRUCT);
        iprot_->readMessageEnd ();
        iprot_->getTransport ()->readEnd ();
    }
    if (fname.compare ("printString") != 0) {
        iprot_->skip (T_STRUCT);
        iprot_->readMessageEnd ();
        iprot_->getTransport ()->readEnd ();
    }

    Example_printString_presult result;
    result.read (iprot_);
    iprot_->readMessageEnd ();
    iprot_->getTransport ()->readEnd ();

    return;
}


int32_t Example_printString_args::write (TProtocol* oprot) const {
```

```
        ...

        xfer += oprot->writeStructBegin ("Example_printString_args");

        xfer += oprot->writeFieldBegin ("text", T_STRING, 1);
        xfer += oprot->writeString (this->text);
        xfer += oprot->writeFieldEnd ();

        xfer += oprot->writeFieldStop ();
        xfer += oprot->writeStructEnd ();

        return xfer;
    }
```

2.20.4. References

1. The Apache Thrift Project Home Page. https://thrift.apache.org

2.21. Web Services

Web services standardize an infrastructure for integrating information systems in the environment of the Internet. The two major web services standards are the Simple Object Access Protocol (SOAP) and the Web Service Description Language (WSDL).

2.21.1. SOAP

The SOAP standard by W3C defines a communication protocol based on a textual form of message encoding in XML. Each message consists of a series of optional headers and a body, with the headers carrying information intended for systems that route the message and the body intended for the final recipient of the message. The messages are extensible and easy to transport via HTTP.

Message Example

```
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
               SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- Header with additional information -->
    <SOAP:Header>
        <wscoor:CoordinationContext
            xmlns:wscoor="http://schemas.xmlsoap.org/ws/2003/09/wscoor"
            SOAP:mustUnderstand="true">
            <wscoor:Identifier>
                http://example.com/context/1234
            </wscoor:Identifier>
        </wscoor:CoordinationContext>
    </SOAP:Header>

    <!-- Body with message content -->
    <SOAP:Body>
        <m:aMethodRequest xmlns:m="http://example.com/soap.wsdl">
            <aNumber xsi:type="xsd:int">42</aNumber>
        </m:aMethodRequest>
    </SOAP:Body>

</SOAP:Envelope>
```

The SOAP standard also introduces a data model, which allows describing simple and compound types, as well as encoding rules, which allow encoding graphs of typed data. Unfortunately, the data model is not explicitly related to XML Schema, which is used to describe simple and compound types in WSDL. Encoding of types described using XML Schema therefore does not necessarily pass validation using the same XML Schema. This discrepancy makes it difficult to validate a SOAP encoded message given the WSDL description of the service for which the message is intended.

Many technologies prefer literal to encoded messages, with the language binding defined directly between the XML Schema in WSDL and the implementation language, rather than between the SOAP data model and the implementation language. This is the case of JAX-RPC and JAX-WS with JAXB.

2.21.2. WSDL

The WSDL standard by W3C defines a web service description in XML. For each service, the description specifies all the data types and message formats used by the service, the message encoding and the communication protocol supported by the service, and the network addresses of the service. The description thus provides all the information that is required to set up communication with the service.

Service Example

```
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:xsd="http://example.com/stockquote.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

    <!-- Types used in communication -->
    <types>
        <schema targetNamespace="http://example.com/stockquote.xsd"
                xmlns="http://www.w3.org/2000/10/XMLSchema">
            <element name="TradePriceRequest">
                <complexType>
                    <all>
                        <element name="tickerSymbol" type="string"/>
                    </all>
                </complexType>
            </element>
            <element name="TradePriceReply">
                 <complexType>
                     <all>
                         <element name="price" type="float"/>
                     </all>
                 </complexType>
              </element>
       </schema>
  </types>

    <!-- Messages exchanged in communication -->
    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd:TradePriceReply"/>
    </message>

    <!-- Ports available in communication -->
    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
            <input message="tns:GetLastTradePriceInput"/>
            <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>

    <!-- Bindings used in communication -->
    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding
            style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
            <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
            <input><soap:body use="literal"/></input>
            <output><soap:body use="literal"/></output>
```

```
        </operation>
    </binding>

    <!-- Service -->
    <service name="StockQuoteService">
        <documentation>Stock quoter service.</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
            <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>

</definitions>

<!-- Example adjusted from WSDL 1.1 Specification. -->
```

The example specifies that the service is accessible using SOAP over HTTP with the document style binding. The document style binding imposes no restriction on the SOAP message structure. An alternative RPC style binding requires that the SOAP message body contains a single procedure element with multiple argument elements as children, the element names are the procedure and argument names.

2.21.3. BPEL

The BPEL standard by OASIS defines a language for service orchestration. Orchestration is described by connecting primitive activities through structured activities. Primitive activities are wait for request, send reply, invoke service, assign variable, throw exception, delay. Structured activities are synchronous sequence, parallel flow, switch, while.

BPEL Example

```
<process name="anExampleProcess">

    <!-- Partners of the example process -->
    <partnerLinks>
        <partnerLink name="client"
            partnerLinkType="aClientPort"
            myRole="aProviderRole"/>
        <partnerLink name="serverOne"
            partnerLinkType="aServerPort"
            myRole="aClientRole"
            partnerRole="aServerRole"/>
        <partnerLink name="serverTwo"
            partnerLinkType="aServerPort"
            myRole="aClientRole"
            partnerRole="aServerRole"/>
    </partnerLinks>

    <!-- Internal variables -->
    <variables>
        <variable name="clientRequest" messageType="RequestMessage"/>
        <variable name="serverOneResponse" messageType="ResponseMessage"/>
        <variable name="serverTwoResponse" messageType="ResponseMessage"/>
        <variable name="providerResponse" messageType="ResponseMessage"/>
    </variables>

    <!-- Process definition -->
    <sequence>
        <!-- Get the request from the client -->
        <receive partnerLink="client"
            portType="aClientPort"
            operation="GetOffer"
            variable="clientRequest"
            createInstance="yes"/>

        <!-- Forward the request to both servers -->
        <flow>
            <invoke partnerLink="serverOne"
                portType="aServerPort"
```

```
                operation="GetOffer"
                inputVariable="clientRequest"
                outputVariable="serverOneResponse"/>
            <invoke partnerLink="serverTwo"
                ... />
        </flow>

        <!-- Create response from cheapest offer -->
        <switch>
            <case condition="bpws:getVariableData ('serverOneResponse','price')
                            <
                            bpws:getVariableData ('serverTwoResponse','price')">
                <assign>
                    <copy>
                        <from variable="serverOneResponse"/>
                        <to variable="providerResponse"/>
                    </copy>
                </assign>
            </case>
            <otherwise>
                ...
            </otherwise>
        </switch>

        <!-- Return the response to the client -->
        <reply partnerLink="client"
            portType="aClientPort"
            operation="GetOffer"
            variable="providerResponse"/>
    </sequence>
</process>
```

2.22. 0MQ

0MQ is a middleware for versatile messaging communication. 0MQ implements enhanced sockets that provide an interface to communication patterns such as publish subscribe or pipeline. 0MQ exports a low level interface in C, intended for use by language binding implementations, and a multitude of high level interfaces in languages ranging from C and C++ to Haskell, intended for application use.

2.22.1. Sockets

0MQ sockets represent the communication endpoints. Each socket has a type, specified at the socket creation time, which describes the role of the socket in a particular communication pattern.

Socket Creation Functions

Low Level Functions.

```
void *zmq_socket (void *context, int type);
int zmq_close (void *socket);
```

context	initialized library handle
type	role in communication pattern

High Level Functions.

```
// Generic socket creation functions.
zsock_t *zsock_new (int type);
void zsock_destroy (zsock_t **self_p);

// Type specific socket creation functions.
zsock_t *zsock_new_pub (const char *endpoint);
zsock_t *zsock_new_sub (const char *endpoint, const char *subscribe);
zsock_t *zsock_new_req (const char *endpoint);
zsock_t *zsock_new_rep (const char *endpoint);
```

A socket can use multiple supported transports. Basic transport types are local communication within a process (inproc), communication over local pipe (ipc), communication over hypervisor transport (vmci), communication over TCP sockets (tcp), communication over IP multicast (pgm) and UDP multicast (epgm).

A socket is connected with a transport by calling `zmq_bind` for incoming connections and by calling `zmq_connect` for outgoing connections. A single socket can be connected multiple times to multiple transports.

Socket Connection Functions

Low Level Functions.

```
int zmq_bind (void *socket, const char *endpoint);
int zmq_connect (void *socket, const char *endpoint);
```

inproc	communication within process • address is an arbitrary string
ipc	communication over local pipe • address is a local pipe file name
vmci	communication over hypervisor transport • address is a virtual machine identifier or hypervisor • address includes port number with function similar to TCP or UDP
tcp	TCP socket opened on demand
pgm	IP multicast to destination
epgm	UDP multicast to destination

High Level Functions.

```
int zsock_bind (zsock_t *self, const char *format, ...);
int zsock_connect (zsock_t *self, const char *format, ...);
```

Actual transport operations are asynchronous. In particular, nodes can call `zmq_bind` and `zmq_connect` in any order, connections are established transparently including across network outages (except in process communication where binding must come before connecting).

Asynchronous transport operation is carried out by dedicated threads. The design expects the thread count to reflect the traffic volume, with one thread handling roughly a gigabyte of traffic per second, regardless of the connection count.

In contrast to the background transport operation, where a thread can handle multiple connections, each socket should be used by a single thread only. Multiple threads can communicate through the process local sockets.

Sockets are configured through socket options. These include low level connection options such as system buffer sizes, message size limits, high water marks (which determine when excess messages are discarded or threads blocked), transport reconnection intervals, thread affinity settings, message filters and more.

Socket Configuration Functions

Low Level Functions.

```
int zmq_setsockopt (void *socket, int option_name, const void *option_value, size_t option_len);
int zmq_getsockopt (void *socket, int option_name, void *option_value, size_t *option_len);
```

ZMQ_SUBSCRIBE	subscription filter
ZMQ_SNDHWM, ZMQ_RCVHWM	high water mark
ZMQ_SNDBUF, ZMQ_RCVBUF	system buffer size

ZMQ_RECONNECT_IVL	transport reconnect interval
ZMQ_RECOVERY_IVL	multicast absence tolerance
ZMQ_AFFINITY	transport thread affinity
ZMQ_RATE	multicast data rate

High Level Functions.

```
// Convert from high level to low level socket reference.
void *zsock_resolve (void *self);
```

Messages are byte arrays even in heterogeneous environments. Messages are delivered atomically, regardless of size. Multipart messages are supported. Regardless of transport, message delivery is not guaranteed.

Message Transport Functions

Low Level Functions.

```
int zmq_msg_send (zmq_msg_t *msg, void *socket, int flags);
int zmq_msg_recv (zmq_msg_t *msg, void *socket, int flags);
```

- messages are byte arrays
- message delivery is atomic
- multipart messages are supported

High Level Functions.

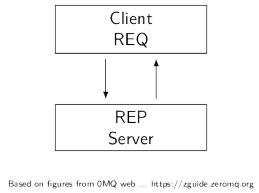
```
zmsg_t *zmsg_recv (void *source);
int zmsg_send (zmsg_t **self_p, void *dest);

// Multipart message functions.
size_t zmsg_size (zmsg_t *self);
int zmsg_prepend (zmsg_t *self, zframe_t **frame_p);
int zmsg_append (zmsg_t *self, zframe_t **frame_p);
zframe_t *zmsg_pop (zmsg_t *self);
zframe_t *zmsg_first (zmsg_t *self);
zframe_t *zmsg_next (zmsg_t *self);
```

2.22.2. Patterns

0MQ patterns represent the communication topology. Each socket has a role in a patern.

The request reply pattern connects clients with the ZMQ_REQ socket type to servers with the ZMQ_REP socket type. The pattern expects alternating request and reply messages and provides round robin to multiple servers and fair queueing to multiple clients. An intermediary made from the ZMQ_ROUTER and ZMQ_DEALER socket types can be used in an extended variant of the pattern.

Synchronous Request Reply Pattern



Connects multiple clients to multiple servers.

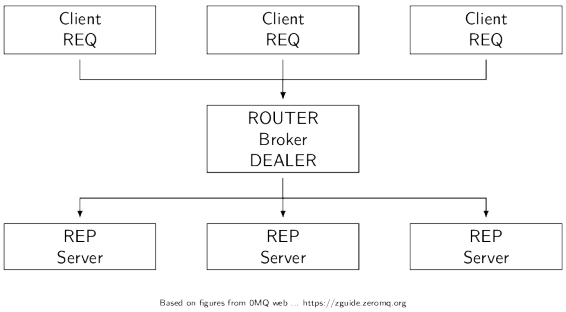
ZMQ_REQ.

- used by client to send requests and receive replies
- allows only alternating sequence of send and recv
- round robin when multiple servers connected
- blocks when no service available

ZMQ_REP.

- used by service to receive requests and send replies
- allows only alternating sequence of recv and send
- fair queueing among clients

Asynchronous Request Reply Pattern



Connects multiple clients to multiple servers through an intermediary.

ZMQ_ROUTER.

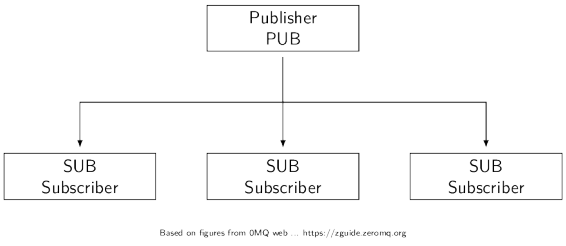
- receives requests from clients with fair queueing
- prefixes requests with client identifier
- delivers replies to identified client

ZMQ_DEALER.

- receives replies from servers with fair queueing
- delivers requests to servers with round robin

The publish subscribe pattern connects publishers with the ZMQ_PUB socket type to subscribers with the ZMQ_SUB socket type. The pattern distributes messages from a publisher to all connected subscribers. An intermediary made from the ZMQ_XSUB and ZMQ_XPUB socket types can be used in an extended variant of the pattern.

Static Publish Subscribe Pattern



Connects multiple publishers to multiple subscribers.

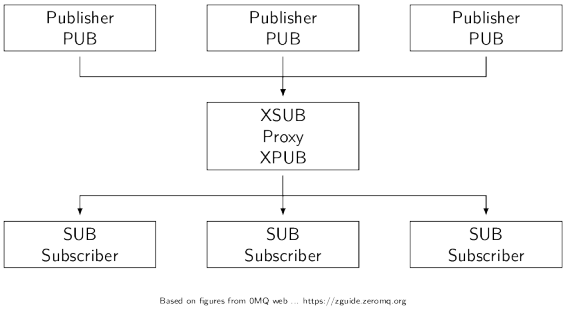
ZMQ_PUB.

- delivers messages to connected subscribers
- never blocks

ZMQ_SUB.

- receives messages from connected publishers
- fair queueing among publishers

Dynamic Publish Subscribe Pattern



Connects multiple publishers to multiple subscribers through an intermediary.

ZMQ_XSUB.

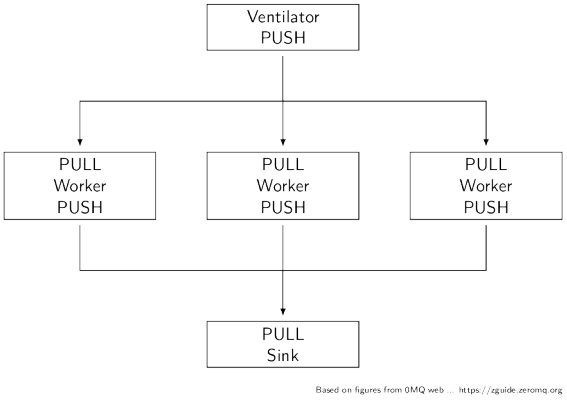
- delivers subscription requests to connected publishers
- receives messages from connected publishers
- fair queueing among publishers

ZMQ_XPUB.

- receives subscription requests from connected subscribers
- delivers messages to connected subscribers
- fair queueing among subscribers
- never blocks

The pipeline pattern connects task generators with the ZMQ_PUSH socket type to task processors with the ZMQ_PULL socket type. The pattern delivers messages from a task generator to a task processor.

Pipeline Pattern



Connects task generators to task processors.

ZMQ_PUSH.

- delivers messages to connected task processors
- round robin among processors

ZMQ_PULL.

- receives messages from connected task generators
- fair queueing among generators

The exclusive pair pattern connects two peers with the ZMQ_PAIR socket type. The pattern is designed for communication within process.

More draft socket patterns exist, such as client server (a thread safe alternative to request reply), radio dish (a thread safe alternative to publish subscribe), channel (a thread safe alternative to exclusive pair), or peer to peer.

2.22.3. References

1. The 0MQ Guide. https://zguide.zeromq.org
2. The 0MQ RFC Specifications. https://rfc.zeromq.org

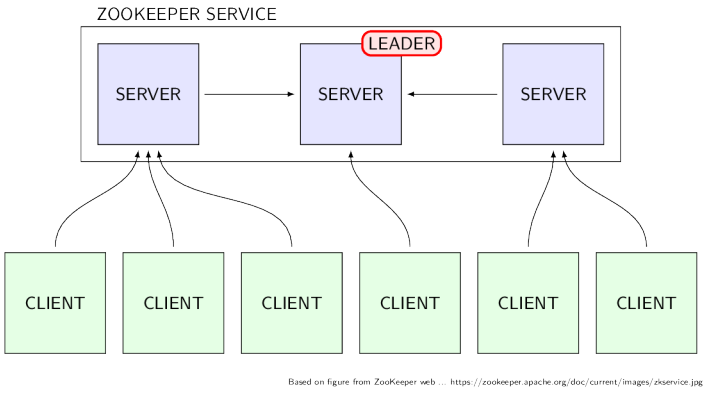
2.23. Apache ZooKeeper

Apache ZooKeeper is a distributed service configuration repository. Java and C bindings are available as part of the project, multiple other bindings are provided by community.

2.23.1. Architecture

ZooKeeper servers maintain synchronized memory state with persistent journal and snapshots. Clients specify a server list and connect to a single server at a time with fail over.

ZooKeeper Architecture



Servers. Replicated server cluster

- Each server stores complete state in memory
- Updates are also stored in persistent log
- Persistent snapshot done when updates accumulate

Atomic communication protocol

- All updates pass through leader server
- Leader collects majority quorum for each update
- Leader election triggered in case of cluster failure

Clients.

- Provided with a list of servers to use
- Connected to a single server at a time
- Connection failure handled by switching to another server

2.23.2. Interface

ZooKeeper Data Model

Data.

- Tree of named nodes navigated by string paths
- Support for unique node naming
- Node data is array of bytes
- Updates increment version

Some data objects in the interface are generated from platform independent specification.

ZooKeeper Data Objects

```
module org.apache.zookeeper.data {
    ...
    class Stat {
        long czxid;             // ZXID of transaction that created this node
        long mzxid;             // ZXID of transaction that last modified this node
        long pzxid;             // ZXID of transaction that last modifined node children
        long ctime;             // Node creation time
        long mtime;             // Node last modification time
        int version;            // Node version
        int aversion;           // Node ACL version
        int cversion;           // Node child version
        int dataLength;         // Node data length
        int numChildren;        // Node child count
        long ephemeralOwner;    // Owner identifier for ephemeral nodes
    }
    ...
}
```

ZooKeeper Blocking Interface

```
public class ZooKeeper {
    public ZooKeeper (String connectString, int sessionTimeout, Watcher watcher) { ... }
    public ZooKeeper (String connectString, int sessionTimeout, Watcher watcher, boolean canBeReadO
    ...

    public String create (String path, byte data [], List<ACL> acl, CreateMode createMode) { ... }
    public void delete (String path, int version) { ... }

    public Stat exists (String path, boolean watch) { ... }
    public Stat exists (String path, Watcher watcher) { ... }

    public byte [] getData (String path, boolean watch, Stat stat) { ... }
    public byte [] getData (String path, Watcher watcher, Stat stat) { ... }

    public Stat setData (String path, byte data [], int version) { ... }

    public List<String> getChildren (String path, boolean watch) { ... }
    public List<String> getChildren (String path, boolean watch, Stat stat) { ... }
    public List<String> getChildren (String path, Watcher watcher) { ... }
```

```
    public List<String> getChildren (String path, Watcher watcher, Stat stat) { ... }

    // Make sure the server is current with the leader.
    public void sync (String path, VoidCallback cb, Object ctx) { ... }

    public synchronized void close () { ... }
}
```

ZooKeeper Non Blocking Interface

```
public class ZooKeeper {
    ...

    public void create (
        String path, byte data [],
        List<ACL> acl, CreateMode createMode,
        StringCallback cb, Object ctx) { ... }

    public void delete(String path, int version, VoidCallback cb, Object ctx) { ... }

    public void exists (String path, boolean watch, StatCallback cb, Object ctx) { ... }
    public void exists (String path, Watcher watcher, StatCallback cb, Object ctx) { ... }

    public void getData (String path, boolean watch, DataCallback cb, Object ctx) { ... }
    public void getData (String path, Watcher watcher, DataCallback cb, Object ctx) { ... }

    public void setData (String path, byte data [], int version, StatCallback cb, Object ctx) { ...

    public void getChildren (String path, boolean watch, ChildrenCallback cb, Object ctx) { ... }
    public void getChildren (String path, boolean watch, Children2Callback cb, Object ctx) { ... }
    public void getChildren (String path, Watcher watcher, ChildrenCallback cb, Object ctx) { ... }
    public void getChildren (String path, Watcher watcher, Children2Callback cb, Object ctx) { ...

    ...
}

public interface StatCallback extends AsyncCallback {
    public void processResult (int rc, String path, Object ctx, Stat stat);
}

public interface DataCallback extends AsyncCallback {
    public void processResult (int rc, String path, Object ctx, byte data [], Stat stat);
}

public interface ChildrenCallback extends AsyncCallback {
    public void processResult (int rc, String path, Object ctx, List<String> children);
}

public interface Children2Callback extends AsyncCallback {
    public void processResult (int rc, String path, Object ctx, List<String> children, Stat stat);
}

...
```

ZooKeeper Multiple Operations Interface

```
public class ZooKeeper {
    ...

    // Execute multiple operations atomically.
    public List<OpResult> multi (Iterable<Op> ops) { ... }
    public void multi (Iterable<Op> ops, MultiCallback cb, Object ctx) { ... }

    ...
}

public abstract class Op {
    private int type;
    private String path;
```

```
    private Op (int type, String path) {
        this.type = type;
        this.path = path;
    }

    public static Op create (String path, byte [] data, List<ACL> acl, int flags) {
        return new Create (path, data, acl, flags);
    }

    public static class Create extends Op {
        private byte [] data;
        private List<ACL> acl;
        private int flags;

        private Create (String path, byte [] data, List<ACL> acl, int flags) {
            super (ZooDefs.OpCode.create, path);
            this.data = data;
            this.acl = acl;
            this.flags = flags;
        }

        ...
    }

    ...
}


public abstract class OpResult {
    private int type;

    private OpResult (int type) {
        this.type = type;
    }

    public static class CreateResult extends OpResult {
        private String path;

        public CreateResult (String path) {
            super (ZooDefs.OpCode.create);
            this.path = path;
        }

        ...
    }

    ...
}
```

ZooKeeper Watcher Interface

```
public class ZooKeeper {
    ...

    // Manage watches with explicit mode.
    void addWatch (String basePath, AddWatchMode mode);
    void removeWatches (String path, Watcher watcher, Watcher.WatcherType watcherType, boolean loca

    ...
}


public enum AddWatchMode {
    PERSISTENT (0),
    PERSISTENT_RECURSIVE (1);
}


public interface Watcher {
```

```
    abstract public void process (WatchedEvent event);

    public interface Event {
        public enum EventType {
            None (-1),
            NodeCreated (1),
            NodeDeleted (2),
            NodeDataChanged (3),
            NodeChildrenChanged (4);

            ...
        }
    }
}


public class WatchedEvent {
    ...

    public KeeperState getState () { ... }
    public EventType getType () { ... }
    public String getPath () { ... }
}
```

- One shot watches are removed after every event
- Persistent watches stay until removed explicitly
- Recursive watches also report events on children

Watchers will receive notification on connection failures but non delivered events are considered lost afterwards.

2.23.3. Recipes

The atomicity and consistency guarantees provided by Apache ZooKeeper can be used to implement multiple high level recipes. Such implementations are provided by the Apache Curator project.

Curator Recipes

Agreement.

GroupMember	group membership tracking
LeaderLatch	leader election with polling interface
LeaderSelector	leader election with callback interface

Synchronization.

DistributedBarrier	barrier with explicit state setting calls
DistributedDoubleBarrier	barrier with node count condition
InterProcessMutex	recursive lock
InterProcessSemaphoreMutex	non recursive lock
InterProcessReadWriteLock	recursive read write lock
InterProcessSemaphore	semaphore
InterProcessMultilock	wrapper for acquiring multiple locks atomically

Communication.

SimpleDistributedQueue	backwards compatible queue
DistributedQueue	ordered queue with optional item identities
DistributedDelayQueue	queue with delayed delivery
DistributedPriorityQueue	queue with priorities
SharedCount	shared integer counter

DistributedAtomicLong	shared long integer counter

Resiliency.

CuratorCache	generic local path cache
PersistentNode	connection loss resistant node interface
PersistentTTLNode	connection loss resistant node interface with keepalive
PersistentWatcher	connection loss resistant watch interface

2.23.4. References

1. The Apache ZooKeeper Project Home Page. https://zookeeper.apache.org

2. The Apache Curator Project Home Page. https://curator.apache.org