Middleware

Petr Tůma

Middleware
by Petr Tůma

This is version 179M generated on 2016-03-09 14:06:17.
For the latest version, check http://d3s.mff.cuni.cz/~ceres.

Table of Contents

Chapter 1. Introduction

The origins of the term "middleware" can be traced back to late 1960s, when it was used to denote software technologies situated between an application and the underlying software platform, with the goals of masking platform dependence and providing specialized services. In the late 1980s and especially throughout 1990s, the use of the term shifted. Rather than denoting any software technology interposed between an application and the underlying software platform, the term is now used more strictly in the context of distributed applications, where the issues of platform dependence and specialized services stand out. Several more or less formal definitions of the term "middleware" illustrate the concept:

> The term "middleware" is defined by one's point of view. Many interesting categorizations exist, all centered around sets of tools and data that help applications use networked resources and services. Some tools, such as authentication and directories, are in all definitions of middleware. Other services, such as coscheduling of networked resources, secure multicast, object brokering, and messaging, are the particular interests of certain communities, such as scientific researchers or business systems vendors. This breadth of meaning is reflected in the following working definition: Middleware is "the intersection of the stuff that network engineers don't want to do with the stuff that applications developers don't want to do."
>
> —Klingenstein K. J.: Middleware: The Second Level of IT Infractructure.

> Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware is essential to migrating mainframe applications to client-server applications and to providing for communication across heterogeneous platforms. This technology has evolved during the 1990s to provide for interoperability in support of the move to client-server architectures. The most widely-publicized middleware initiatives are the Open Software Foundation's Distributed Computing Environment (DCE), Object Management Group's Common Object Request Broker Architecture (CORBA), and Microsoft's COM/DCOM (COM, DCOM).
>
> —Bray M.: Middleware.

> The role of middleware is to ease the task of designing, programming and managing distributed applications by providing a simple, consistent and integrated distributed programming environment. Essentially, middleware is a distributed software layer, or platform, which abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages.
>
> —Coulson G.: Middleware.

Developing and maintaining distributed applications requires considerable effort. The evolution of middleware reflects more general trends to reduce this effort, especially through reducing diversity and reducing coupling. One example of the drive to reduce diversity is the introduction of *architectural styles* that impose reasonable constraints on distributed application design, standardized platforms then support those architectural styles to make development easier. One example of the drive to reduce coupling is the tendency to strictly separate interfaces from implementation, which limits change propagation, and, when interfaces are based on the problem domain rather than the application implementation, also has the potential to reduce change frequency.

Examples of simple architectural styles include a distributed application built around a central server that implements the entire application and clients that act as remote terminals, or a distributed application built around a central server that implements a network filesystem and clients that implement the entire application. Other architectural styles include tiered architectures, distributed object architectures, event based architectures, data centric architectures, service-oriented architectures, peer-to-peer architectures. Obviously, the styles are not necessarily exclusive.

References

1. Bray M.: Middleware. Software Technology Review, Carnegie Mellon Software Engineering Institute, 1997, http://www.sei.cmu.edu/str/descriptions/middleware.html.
2. Coulson G.: Middleware. Distributed Systems Online, IEEE, 2000, http://computer.org/dsonline/middleware/index.htm.
3. Klingenstein K. J.: Middleware: The Second Level of IT Infrastructure. Cause And Effect Journal Vol. 22 No. 4, EduCause, 1999, http://www.educause.edu/ir/library/html/cem/cem99/cem9942.html.

Tiered Architectures

A tiered architecture emphasizes structuring of an application into *tiers* that implement distinct features such as the presentation logic or the application logic. The architecture is typically supported by middleware frameworks that provide functions tailored to particular tiers. An example of a tiered architecture is illustrated on Figure 1-1.

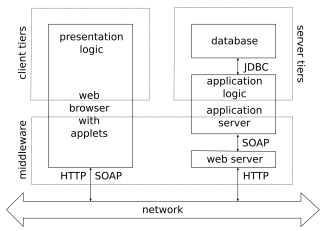

Figure 1-1. Tiered Architecture Example

A distinct case of a tiered architecture is the *two-tier* architecture, which distinguishes the client tier and the server tier. The client tier implements the presentation logic, the server tier stores the application data. The application logic is split between the client and the server tier, with the case where the client tier contains most of the logic denoted as a thick-client architecture, and the case where the server tier contains most of the logic denoted as a thin-client architecture. Thick-client architectures have the potential of minimizing network communication and server workload, whereas thin-client architectures have the potential of simplifying client maintenance.

The *three-tier* architecture introduces a middle tier between the client tier and the server tier. The middle tier implements the application logic, which was split between the client tier and the server tier in the two-tier architecture.

Rather than being limited to two or three tiers, the architecture of a distributed application can distinguish more tiers. To give a few examples, a user tier can handle the authentication and authorization, a workspace tier can establish sessions, a resource tier can manage resource sharing resources, etc. The benefits of introducing more tiers, such as having explicit application architecture and providing increased reuse potential, are weighted against the drawbacks, most notably the architecture complexity and the associated overhead.

Service-Oriented Architectures

A service-oriented architecture emphasizes structuring of an application into largely independent *services* that are composed to form various applications. An example of a service-oriented architecture is illustrated on Figure 1-2.

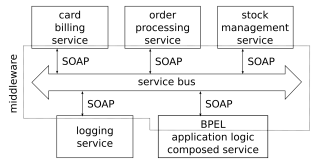

Figure 1-2. Service-Oriented Architecture Example

Service-oriented middleware emphasizes support for service integration, which is carried out by independent integrators rather than by the service developers. Independent integration requires support for service deployment, service location, declarative rather than imperative service composition, and other features that lead to loosely coupled architectures.

Rehearsal

At this point, you should have a rough appreciation for the position of middleware in distributed applications and for the design concerns shaping the architecture of distributed applications.

Questions

1. Explain the term *client-server architecture*.
2. Explain the term *two-tier architecture*.
3. Explain the term *three-tier architecture*.
4. Explain the term *middleware*.

Chapter 2. Communication

The support for communication in a distributed application is perhaps the most frequently offered middleware feature. The support for communication includes building network protocols on top of the functions provided by the network hardware, and building application interfaces on top of the functions provided by the network protocols.

The role of network protocols in middleware is efficiently supporting various forms of communication that the applications may require. The protocols are built in layers that form a protocol stack. At the very bottom of the stack are the functions provided by the network hardware, which allow sending and receiving packets on the network segment connected directly to the network hardware. Further up the protocol stack, other layers add necessary functions such as routing, fragmentation and reassembly, acknowledgement and retransmission, etc.

The role of application interfaces in middleware is seamlessly integrating various forms of communication into the applications. The interfaces take care of formatting the data to be communicated, synchronizing the communicating parties, reporting the communication errors, etc.

The following sections describe selected groups of network protocols, roughly in the order of increasing complexity, and selected groups of application interfaces.

Unreliable Unicast

The term *unicast* describes communication where data is sent from a single sender to a single recipient. The term *unreliable* denotes an absence of guarantees that data sent by the sender are received by the recipient. To avoid the extreme of no communication being a special case of unreliable communication, *best effort* guarantees are assumed implicitly.

Unreliable unicast communication is provided by the network hardware. Typically, however, the network hardware can only communicate directly with other nodes connected to the same network segment. Additionally, the size of the data packets can be limited. To circumvent these limitations, addressing, routing, fragmentation and reassembly mechanisms are added by the unreliable unicast layer.

Example: IP Addressing And Routing

The IP addressing mechanism assigns a distinct address to each network interface. The address is 4 bytes long in IP protocol version 4 and 16 bytes long in IP protocol version 6. In both versions, the address is structured into parts of variable length, typically including the address prefix, the subnet identification and the interface identification. The *address prefix* determines the class of the address, with each class of addresses having a specific use. The *subnet identification* is a part of the address that is common for all hosts connected to the same network segment. The *interface identification* is a part of the address that distinguishes hosts within the network segment.

The IP routing mechanism distinguishes two basic situations when sending a data packet depending on whether both the sender and the recipient share the same network segment. When this is the case, the IP routing mechanism uses the network hardware to send the packet directly to the recipient, otherwise the packet is sent to a router on the same network segment rather than to the recipient.

The IP addresses are not related to the addresses used by the network hardware. When sending a data packet, ARP (Address Resolution Protocol) is used to query the address used by the network hardware. The result of the query is cached to avoid the overhead of sending the ARP query and receiving the ARP reply whenever sending a data packet.

References

1. Postel J.: RFC 0791 - Internet Protocol Version 4 (IPv4) Specification
2. Postel J.: RFC 0792 - Internet Control Message Protocol (ICMPv4) Specification
3. Plummer D. C.: RFC 0826 - Ethernet Address Resolution Protocol
4. Hinden R., Deering S.: RFC 2373 - Internet Protocol Version 6 Addressing Architecture
5. Deering S., Hinden R.: RFC 2460 - Internet Protocol Version 6 (IPv6) Specification
6. Conta A., Deering S.: RFC 2463 - Internet Control Message Protocol (ICMPv6) Specification
7. Hinden R., Deering S.: RFC 3513 - Internet Protocol Version 6 (IPv6) Addressing Architecture

Example: IP Fragmentation And Reassembly

When sending a data packet, the length of the packet can exceed MTU (Maximum Transmission Unit) of the network interface. When that is the case, the packet is split into fragments no larger than MTU, identified by their offset within the data packet. The fragments are sent separately and reassembled at the recipient.

Different network hardware can use different MTU. Fragmentation can therefore occur at each router that is connected to network segments with different MTU values. The sender of a data packet can discover the minimum of the MTU values on the path the packet travels, denoted as PMTU (Path Maximum Transmission Unit), and split the packet into fragments no larger than PMTU to minimize fragmentation.

References

1. Mogul J. C., Deering S. E.: RFC 1191 - Path MTU Discovery for IPv4
2. McCann J., Deering S., Mogul J.: RFC 1981 - Path MTU Discovery for IPv6

Reliability In Unicast

The term *reliability* generally denotes a presence of guarantees that data sent by the sender are received by the recipient. Ideally, data that were sent would always be received and no data would be received that were not sent. This ideal is referred to as *exactly once* delivery semantics.

To guarantee the exactly once delivery semantics, the protocol stack has to cope with various network failures. The most frequently considered network failures are damage to a data packet, loss of a data packet, duplication of a data packet.

Packet Damage

To detect damage to a data packet, the sender supplements the data with an additional information that is calculated from the data as it is being sent. The recipient can check whether the same calculation from the data as it is being received yields the same additional information. When that is not the case, either the data or the additional information has been damaged. Obviously, it is still possible that both the

data and the additional information have been damaged in a way that makes the calculation from the damaged data yield the damaged additional information, escaping the damage detection. Compromise is therefore sought between the size of the additional information and the ability to detect damage.

Detection Using Data Duplication

A trivial example of the additional information that can be used to detect damage is a copy of the data. For N bytes of data, N bytes of additional information are needed.

As far as the ability to detect damage is concerned, data duplication can detect 100% of errors that damage a single bit within a packet and 100% of errors that damage up to N adjacent bytes within a packet. Data duplication can, however, miss some errors that damage two bits within a packet, namely errors that damage the same bit in the data and in the additional information.

Data duplication is an example of an additional information that has a relatively low ability to detect damage in spite of its size.

Detection Using Checksum

Adding a modulo sum of all bytes of data requires 1 byte of additional information for N bytes of data. Checksum can detect 100% of errors that damage a single bit within a packet. Checksum can, however, miss some errors that damage two bits within a packet.

Checksum is an example of an additional information that is simple to calculate and reasonably reliable given its size.

Detection Using Parity

Adding an odd or even parity per each bit of all bytes of data requires 1 byte of additional information for N bytes of data. Parity can detect 100% of errors that damage a single bit within a packet. Parity can, however, miss some errors that damage two bits within a packet.

Parity is an example of an additional information that is simple to calculate and reasonably reliable given its size.

Detection Using Cross Parity

Cross parity is calculated by arranging the data in an imaginary square grid and adding an odd or even parity of all rows and all columns of data. Adding a cross parity requires 2*SQRT(N) bits of additional information for N bits of data.

Cross parity can detect 100% of errors that damage a single bit within a packet and 100% of errors that damage two bits within a packet. Cross parity can, however, miss some errors that damage three bits within a packet.

Alternatively, cross parity can be used to correct errors that damage a single bit, looking the bit up by the row and the column of the imaginary square grid that have an incorrect parity. When used to correct errors, cross parity can correct 100% of errors that damage a single bit within a packet. Cross parity can, however, mistake some errors that damage two bits for an error that damages another single bit and cause further damage rather than correct the error.

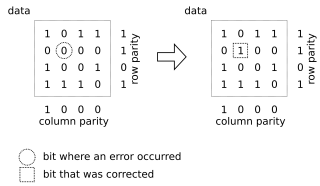

Figure 2-1. Cross Parity Correction Example

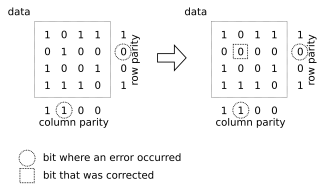

Figure 2-2. Cross Parity Miscorrection Example

Cross parity is an example of an additional information that is simple to calculate and can be used both to detect damage and correct errors.

Detection Using Hamming Code

Hamming code interleaves parity with data. Assume bits in a packet are numbered from 1. A bit at position I is a parity bit when I is a power of two, a data bit otherwise. A parity bit at position I protects groups of I bits spaced I bits apart, starting at position I.

In combination with an extra parity bit, the code can correct 100% of errors that damage a single bit within a packet and detect 100% of errors that damage two bits within a packet. Interestingly, when correcting errors, the position of a damaged bit can be calculated as a sum of positions of the parity bits that indicate the error.

Detection Using Cyclic Redundancy Check

Cyclic redundancy check is calculated in a modulo 2 integer arithmetic. Formally, data is interpreted as coefficients of polynomials in a commutative ring of polynomials over integers modulo 2.

When sending a data packet, the data in form of a polynomial G(x) is shifted by the width of the cyclic redundancy check and divided by a generating polynomial P(x) to form a remainder R(x) = (G(x) * x ^ DEG(P(x))) MOD P(x). The remainder R(x) is added to the data, forming the data packet F(x) = (G(x) * x ^ DEG(P(x))) + R(x). Note that in the modulo 2 integer arithmetic, both adding and subtracting the remainder yields the same result, hence the generating polynomial P(x) divides F(x).

When receiving a data packet, the packet in form of the polynomial F(x) can be damaged by an error in form of a polynomial E(x). The damaged packet F(x) + E(x) is divided by the generating polynomial P(x), yielding a remainder (F(x) + E(x)) MOD P(x) = F(x) MOD P(x) + E(x) MOD P(x) = E(x) MOD P(x). Note that the remainder depends on the error and not on the data.

The ability of the cyclic redundancy check to detect errors depends on the choice of the generating polynomial P(x). A frequent choice for the generating polynomial is x^16 + x^12 + x^5 + 1, a standard generating polynomial denoted as CCITT CRC 16. When considering errors that damage a single bit of data, E(x) has the form of

x^k. Since no polynomial of the form x^k divides CCITT CRC 16 P(x), 100% of single bit errors can be detected. Similarly, it can be shown that CCITT CRC 16 can detect 100% of errors that damage any number of bits not farther apart than 16 bits and 100% of errors that damage any two bits except two bits apart exactly 2^16-1 bits. For arbitrary errors, all but one in 2^16 errors can be detected.

Consider 8 bits of data and 2 bits of redundant information created using P(x) = x^2 + 1.

```
G = 10011101
P = 101

1001110100 DIV 101 = 10110001 (result thrown away)
 011
  111
   101
    000
     001
      010
       100
        01 = R (appended to data to form data packet)
```

Verify that P(x) divides the data packet.

```
F = 1001110101

1001110101 DIV 101 = 10110001
 011
  111
   101
    000
     001
      010
       101
        00 (indicates undamaged data packet)
```

See that P(x) does not divide the data packet after it has been damaged by a random 1 bit error.

```
E = 0001000000
F = 1001110101

1000110101 DIV 101 = 10100100
 010
  101
   001
    010
     101
      000
       001
        01 (indicates damaged data packet)
```

See that (F(x) + E(x)) MOD P(x) = E(x) MOD P(x).

```
E = 0001000000

0001000000 DIV 101 = 00010101
 001
  010
   100
    010
     100
```

```
010
 100
  01 = E MOD P = (F + E) MOD P
```

Figure 2-3. CRC Calculation Example

Cyclic redundancy check is an example of an additional information that is simple to calculate in hardware and reliable for errors that have a character of short bursts of damaged bits.

When Damage Is Detected

Obviously, when a correctable damage to a packet is detected, it is corrected. When an uncorrectable damage to a packet is detected, the packet is discarded and the situation is further handled as a loss of the packet. When the delays involved in handling the loss of the packet are more of a problem than size of the additional information, an additional information with high error correction ability is used so that uncorrectable damage is rare. This is called *forward error correction*.

Packet Loss

The loss of a data packet is handled by introducing acknowledgements. The recipient is expected to acknowledge the delivery of a data packet by sending an acknowledgement packet to the sender of the data packet. When the recipient does not acknowledge the data packet within a timeout, the sender concludes that the packet has been lost and sends it again.

An important issue is the choice of timeout. With a timeout too short, the sender will needlessly resend data packets even when no loss occurs. With a timeout too long, the recipient will needlessly wait for data packets when a loss occurs. The issue is resolved by having an adaptive timeout that grows exponentially when a loss occurs to avoid causing network congestion.

To minimize waiting when damage to a packet is detected, positive and negative acknowledgement packets, denoted as ACK and NAK, can be distinguished. The recipient uses ACK to indicate a correct packet and NAK to indicate a damaged packet. With the timeout necessarily longer than the roundtrip time of the data and acknowledgement packets, sender will sooner get NAK than timeout waiting for ACK.

Given that an acknowledgement packet can be lost same as a data packet, the introduction of acknowledgements can lead to a situation where the data packet is delivered but the corresponding acknowledgement packet is lost. To the sender, this looks the same as if the data packet was lost and the corresponding acknowledgement packet was never sent. Eventually, this leads to a duplication of the data packet.

Packet Duplication

The duplication of a data packet is handled by introducing unique identifiers. The sender tags each data packet with a unique identifier. The recipient records the identifiers of the data packets and discards packets whose identifiers have already been recorded. When packets are not reordered, the mechanism boils down to tagging packets with sequential numbers at the sender side and remembering the last number in each sequence at the recipient side.

Note that when the duplication of a data packet occurs due to the loss of an acknowledgement packet, the acknowledgement packet has to be sent even when the data packet is discarded, otherwise the sender would keep resending the data packet.

Other Failures

Depending on circumstances, the protocol stack might have to cope with other failures beyond damage, loss and duplication of a data packet. One important class of failures, termed *amnesia failures*, involves loss of information on the sender or the recipient. An amnesia failure can cause the sender to forget the unique identifiers of the packets that were sent, or the recipient to forget the unique identifiers of the packets that were delivered. Generally, this makes it impossible to guarantee the exactly once delivery semantics.

As an alternative to the exactly once delivery semantics, the *at most once* and *at least once* variants are considered in practice. The at most once delivery semantics guarantees reliable delivery except for failures that make reliable delivery impossible. When such a failure occurs, it is guaranteed that no packets were duplicated, but some packets could be lost. The at least once delivery semantics also guarantees reliable delivery except for failures that make reliable delivery impossible. When such a failure occurs, it is guaranteed that no packets were lost, but some packets could be duplicated.

Example: TCP Reliability

TCP is used to transport continuous streams of data split into multiple data packets, with each data packet carrying a fragment of the continuous stream. A packet is identified by its position within the stream and protected by a two byte checksum.

The recipient acknowledges the delivery of a data packet by notifying the sender of the position within the stream that the recipient is at. The sender interprets the first acknowledgement of a certain position as a positive acknowledgement of a data packet at that position, and any consecutive acknowledgements of a certain position as a negative acknowledgement of a data packet after that position. For sake of efficiency, traffic in both directions is supported and data packets sent in one direction also serve as acknowledgement packets in the other direction.

Modifications of TCP that allow selective acknowledgement and retransmission exist.

Flow And Congestion

From the application perspective, an individual data packet is not the most suitable unit of communication. An application may need to exchange large messages or continuous streams, both of which need to be split into multiple data packets. While the splitting is simple by itself, exchanging large messages or continuous streams emphasizes need for both flow control and congestion control.

The role of *flow control* is to guarantee that the sender will not send data packets faster than the recipient is able to process them. A simple solution to flow control is defining *a flow control window* as a number or size of data packets that the recipient is able to process. The size of the flow control window is communicated to the sender in acknowledgement messages.

The role of *congestion control* is to guarantee that the sender will not send data packets faster than the network is able to transport them. A simple solution to congestion control is defining *a congestion control window* as a number or size of data packets that the network is able to transport. The size of the congestion control window is adjusted by the sender in response to events that indicate presence or absence of congestion, such as packet delivery or packet loss.

Example: TCP Flow And Congestion

In TCP, flow control and congestion control both rely on dynamically adjusted windows, but the mechanisms used to adjust the windows rely on principally different inputs. In flow control, the main input for adjusting the window is how much data the receiver can accept, and since this information is known to the receiver, the sender can obtain it directly from the receiver. In congestion control, the main input for adjusting the window is how much data the network can accept, and since this information is not directly known to either the sender or the receiver, the sender has to estimate it indirectly.

In more detail for the flow control mechanism, the receiver advertises how many bytes it is still willing to receive in each acknowledgment message, and the receiver pauses when the messages that are in flight would exceed this limit.

In more detail for the congestion control mechanism, the sender estimates the state of the network from four distinct events that it can observe.

- An *in order acknowledgment* is interpreted as an evidence that the network handles the current traffic well, and the congestion control window is slowly grown in response.
- One or two *duplicit acknowledgments*, which occur when the receiver observes out of order messages, are interpreted as an evidence of acceptable message reordering, of no interest to congestion control.
- *Triple duplicit acknowledgments* are interpreted as an evidence of a single message loss and therefore slight network overload. The congestion control window is halved in response.
- *Timeout* is interpreted as an evidence of a significant message loss and therefore significant network overload. The congestion control window is minimized in response.

The congestion window growth is directed by either *Slow Start* or *Congestion Control* algorithms. Roughly, Slow Start increases the congestion window size by one segment every ACK of a single segment, while Congestion Control increases the congestion window size by one segment every ACK of the whole window. The switch from Slow Start to Congestion Control happens when an adaptive congestion window size threshold is reached.

The reaction to the slight network overload is somewhat complicated by the fact that the first indication of recovery in the form of the first in order acknowledgment comes only after the lost message is delivered. Until then, the last acknowledged position in the stream does not change, and if the congestion window were simply halved, no new message would be sent until the first in order acknowledgment arrives. After resending the lost message using the *Fast Retransmit* mechanism, the protocol therefore enters the *Fast Recovery* mode, where the congestion window size is increased on every duplicit ACK. As a result, the artificially inflated congestion window will reach the size that allows sending additional messages when just about half of the previously sent messages has reached the receiver. When the resent message reaches the receiver, the in order acknowledgment arrives and the artificially inflated congestion window is deflated back to the original size.

Timing Guarantees

An application may need guarantees on the timing of communication. Such guarantees are provided by *real time* communication middleware. The *soft* and *hard* real time guarantees are distinguished. Soft real time guarantees mostly hold, hard real time guarantees always hold.

The guarantees on the timing of communication can concern various aspects of timing such as throughput and latency or jitter.

- *Throughput* is defined as the amount of data delivered within a unit of time, guarantees of minimum throughput are useful for estimating encoding quality in streaming multimedia applications.
- *Latency* is defined either as the duration of a *one way trip* of data from the sender to the receiver, or as a duration of a *roundtrip* of data from the sender to the receiver and back. Guarantees of maximum latency are useful for interactive distributed applications.
- *Jitter* is defined as the fluctuation in the communication latency, guarantees of maximum jitter are useful for estimating buffer sizes in streaming multimedia applications.

Obviously, the timing guarantees must be provided by the network hardware in the first place. Achieving higher throughput might require switching from twisted pair links to fiber optic links, achieving lower latency might require switching from queueing routers on Ethernet to cut through routers on Myrinet. The communication protocols build on the network hardware by resource reservation.

Example: Resource Reservation Protocol

RSVP provides support for resource reservation in situations where data is transported from one sender to one or more recipients within a communication session. The sender uses periodic Path messages to notify all nodes along the paths that the data packets take about the existence of the session. The recipients use Resv messages to request that nodes along the paths reserve resources for the session.

Example: Real Time Protocol

RTP je protokol framework pro prenos dat. Kazdy RTP paket obsahuje hlavne payload type (identifikator co veze), sequence number (cislo inkrementovane s kazdym paketem, nahodne na zacatku session), timestamp (casove razitko payloadu, nahodne na zacatku session), SSRC (synchronization source ID), CSRC list (contributing source ID list). Krom RTP paketu se posilaji jeste RTCP pakety, RTCP paket je bud sender report (statistiky vysilacu), receiver report (statistiky prijimacu), source description (identifikace a popis session), nebo ridici paket. Vsechny maji hlavicku podobnou RTP paketum, reporty pak obsahuji casove razitko (v RTP i NTP formatu, aby se dal merit roundtrip a jitter), objem prenesenych dat, objem a procento ztracenych dat, jitter. No a to je v podstate vsechno, pak se uz jen rekne, ze vysilaci a prijimaci mohou RTP framework pouzit ke komunikaci, ze mohou definovat mixery (uzly, ktere spojuji vic vstupnich streamu pod vlastnim SSRC) a translatory (uzly, ktere modifikuji vstupni stream, ale nechavaji SSRC). Reakce na RTCP se nechava na aplikacich, pravdepodobne bude v podobe flow control a upravy sitovych parametru pomoci RSVP.

References

1. Schulzrinne H., Casner S., Frederick R., Jacobson V.: RFC 1889, RTP, A Transport Protocol for Real-Time Applications
2. Schulzrinne H., Casner S., Frederick R., Jacobson V.: RFC 3550, RTP, A Transport Protocol for Real-Time Applications

Example: Real Time Streaming Protocol

RTSP je protokol framework pro prenos streamu. Funkci RTSP pripomina HTTP, ma zpravy s metodami SETUP, PLAY, RECORD, PAUSE, TEARDOWN (zjevna funkce), v nich se pouzivaji bezne URL streamu, kterych se tyto metody maji tykat. Jediny rozdil je v tom, ze data uz se pomoci RTSP neprenaseji, misto toho RTSP ovlada nejaky jiny, napriklad RTP, stream.

RTSP (Real Time Streaming Protocol, RFC 2326).

Rehearsal

The purpose of this section is to sketch the principles and issues related to implementing the unicast communication protocols and relate those principles and issues to the middleware that relies on the protocols.

At this point, you should be able to explain how local unicast communication provided by hardware is extended to global scale through addressing and routing.

You should understand the principal character of the different types of communication failures and the approaches used to cope with the failures. You should be able to explain the limits of these approaches in providing the ideally reliable communication.

You should be able to characterize the communication performance in terms of both quantitative parameters and qualitative behavior. You should be able to explain the practical function of flow and congestion control mechanisms in face of typical situations requiring flow or congestion control.

Questions

1. Define the properties of an ideally reliable communication mechanism in terms of messages sent and received by the participants.

   Hint: What exactly does it mean that communication is reliable ? What is guaranteed when a participant sends a message ? What is guaranteed when a participant receives a message ?

2. Describe the circumstances under which packet damage cannot be masked by the approaches used to provide reliable communication.
3. Describe the circumstances under which packet loss and packet duplication cannot be masked by the approaches used to provide reliable communication.
4. The TCP flow control mechanism is based on the receiver informing the sender of the number of bytes that it can still accept. Explain why this approach is used instead of the receiver simply telling the sender whether it can accept any data or not.

Exercises

1. Navrhněte přenosový protokol, který bude zaručovat spolehlivé doručování zpráv od jednoho odesílatele jednomu příjemci. Váš návrh by měl definovat tyto funkce:

```
void ReliableSend (tMsg *pMessage, tAddr *pTarget);
  // Odeslání zprávy, blokuje do přijetí zprávy
```

```
void ReliableReceive (tMsg &*pMessage, tAddr &*pSource);
  // Příjem zprávy, blokuje do přijetí zprávy, zaručuje
  // právě jeden příjem nepoškozené odeslané zprávy
```

Váš návrh by měl používat tyto funkce:

```
void UnreliableSend (tMsg *pMessage, tAddr *pTarget);
  // Odeslání zprávy, neblokuje, nemusí odeslat

void UnreliableReceive (tMsg &*pMessage, tAddr &*pSource, int iTimeout);
  // Příjem zprávy, blokuje do timeoutu, může přijmout
  // poškozenou zprávu nebo tutéž zprávu vícekrát
```

Dále předpokládejte existenci rozumných funkcí pro manipulaci se zprávami jako je jejich vytváření a rušení, nastavování a dotazování atributů přenášených spolu s obsahem zprávy a podobně.

> **Hint:** The point of this exercise is to cover the entire spectrum of failures possible with the `UnreliableSend` and `UnreliableReceive` functions. For every packet exchanged by the protocol, what happens when it gets damaged, lost, duplicated ?
>
> For those unfamiliar with the particular programming language, note that the sending functions expect the message and the address as their inputs, but the receiving functions as their outputs.

Unreliable Multicast

The terms *multicast* and *broadcast* describe communication where data is sent from a single sender to multiple recipients. Multicast emphasizes delivering to a subset of available recipients, whereas broadcast emphasizes delivering to all available recipients.

The advantage of multicast when compared to a series of unicasts to the same recipients is in that the network hardware can send a single data packet that is received by multiple nodes connected to the same network segment. Using multicast instead of unicast to deliver the same data to multiple recipients can therefore save network bandwidth.

Given that the network hardware can only communicate directly with other nodes connected to the same network segment, addressing and routing mechanisms must be added by the unreliable unicast layer.

Example: IP Multicast Addressing And Routing

The IP multicast uses special IP addresses. In IP protocol version 4, these are class D addresses that begin with an address prefix of 1110b. In IP protocol version 6, these are addresses that begin with an address prefix of 11111111b. Each multicast IP address denotes a group of nodes.

In IPv4, the membership of nodes in groups is managed by IGMP (Internet Group Management Protocol), which allows routers to keep track of which groups are represented on which network segments. IGMP relies on a combination of querying by routers and reporting by nodes.

On each network segment, the router with the numerically smallest IP address periodically multicasts a General Membership Query, which is a variant of the Membership Query with Group Address set to 0. A node that is a member of a group

responds to the query by multicasting a Membership Report. To avoid flooding the network segment, the report is only sent after a random delay during which no other node sent a report for the same group.

A node that joins or leaves a group multicasts a State Change Report, which is a variant of the Membership Report. To find out whether a group is still represented on a network segment, the router with the numerically smallest IP address responds to the report by multicasting a Group Specific Membership Query for each group that the node left. The Group Specific Membership Query is a variant of the Membership Query and nodes that are members of the group respond to the query similarly as to the General Membership Query.

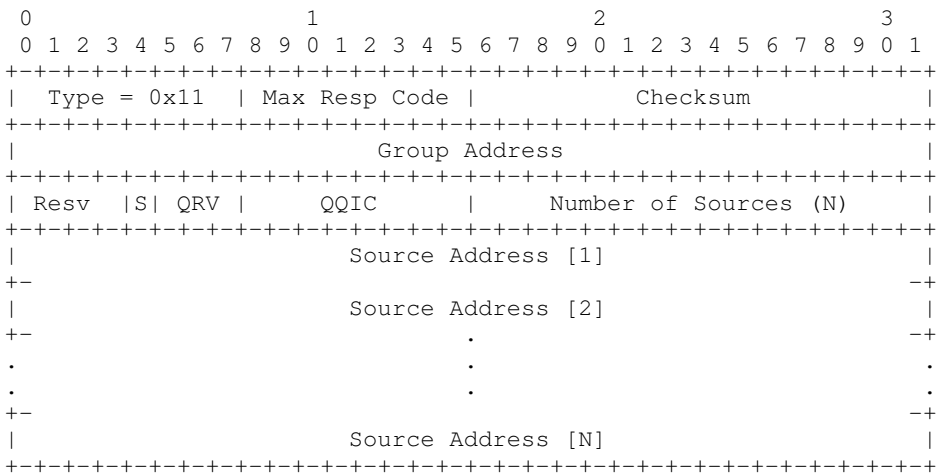

Figure 2-4. IGMP Membership Query

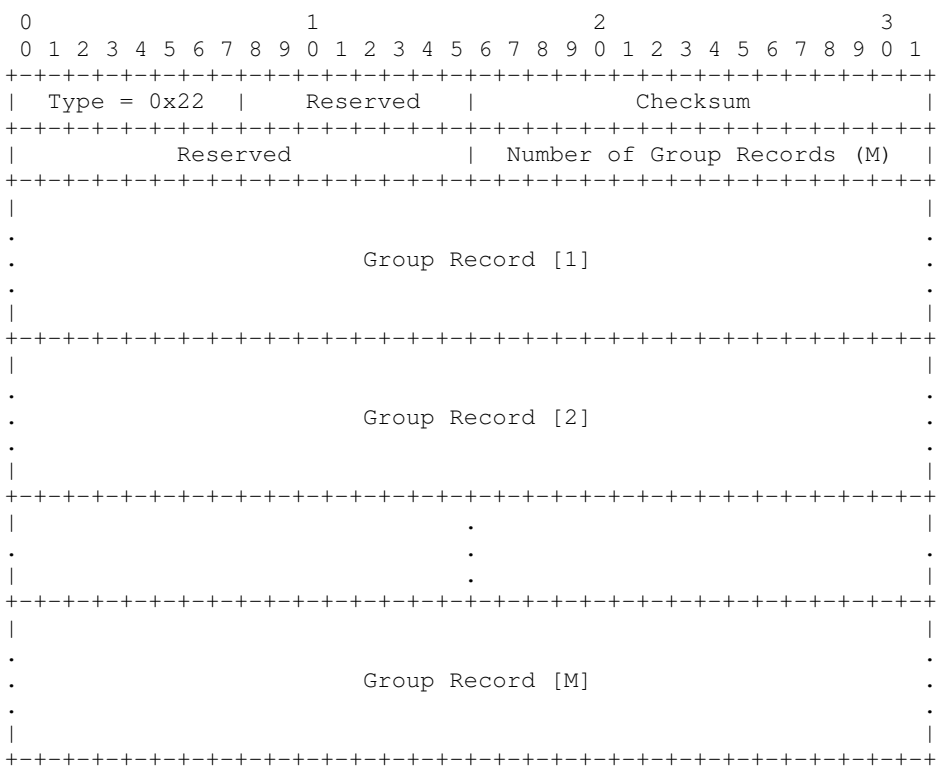

Figure 2-5. IGMP Membership Report

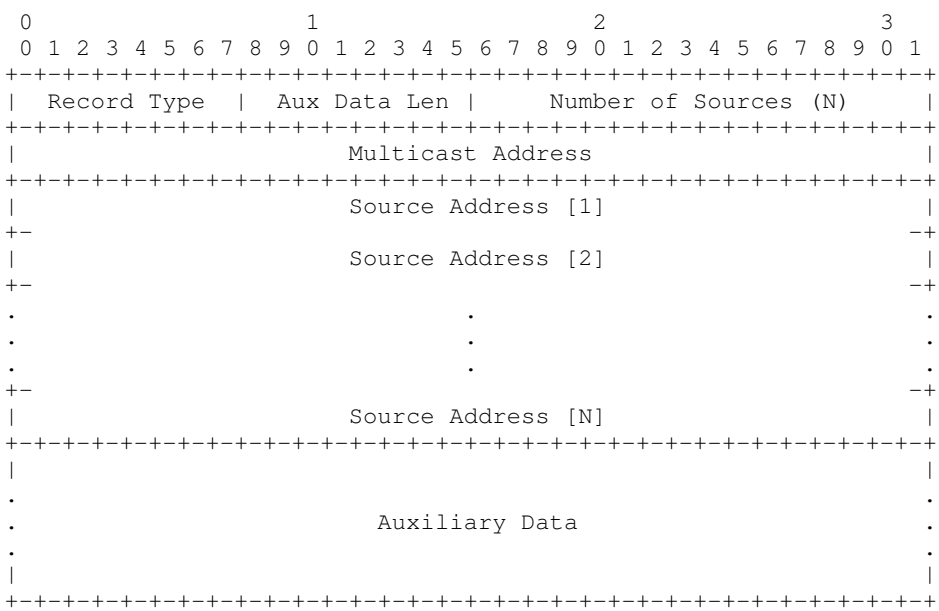

Figure 2-6. IGMP Group Record

In IPv6, IGMP is replaced by technically similar MLD (Multicast Listener Discovery).

Additional protocols are used to configure multicast routing. Examples include DVMRP (Distance Vector Multicast Routing Protocol), MOSPF (Multicast Open Shortest Path First), CBT (Core Based Trees), PIM (Protocol Independent Multicast).

Multicast addresses are either assigned administratively or allocated using MADCAP (Multicast Address Dynamic Client Allocation Protocol) and announced using SAP (Session Announcement Protocol).

References

1. Deering S. E.: RFC 988 - Host Extensions for IP Multicasting
2. Waitzman D., Partridge C., Deering S.: RFC 1075 - Distance Vector Multicast Routing Protocol
3. Moy J.: RFC 1584 - Multicast Extensions to OSPF
4. Ballardie A.: RFC 2189 - Core Based Trees Version 2 Multicast Routing Protocol Specification
5. Estrin D., Farinacci D., Helmy A., Thaler D., Deering S., Handley M., Jacobson V., Liu C., Sharma P., Wei L.: RFC 2362 - Protocol Independent Multicast Sparse Mode Specification
6. Meyer D.: RFC 2365 - Administratively Scoped IP Multicast
7. Hanna S., Patel B., Shah M.: RFC 2730 - Multicast Address Dynamic Client Allocation Protocol
8. Handley M., Perkins C., Whelan E.: RFC 2974 - Session Announcement Protocol
9. Cain B., Deering S., Kouvelas I., Fenner B., Thyagarajan A.: RFC 3376 - Internet Group Management Protocol Version 3

10. Adams A., Nicholas J., Siadak W.: RFC 3973 - Protocol Independent Multicast Dense Mode Specification

Reliability In Multicast

Reliable multicast faces the challenge of extending reliability in form of the exactly once delivery semantics or its less strict cousins to communication where data is sent from a single sender to multiple recipients. A straightforward approach is to use the acknowledgements as in reliable unicast.

References

1. Levine B. N., Garcia-Luna-Aceves J. J.: A Comparison of Known Classes of Reliable Multicast Protocols
2. Pingali S., Towsley D., Kurose J. F.: A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols

Sender Initiated Error Recovery

A simple solution to the problem of packet loss in reliable multicast is by having recipients acknowledge data packets through positive acknowledgements sent to the sender of the data packet. Because it is the sender who is responsible for tracking whether packets need to be resent, the term *sender initiated protocol* is sometimes used.

In sender initiated protocols, the sender must keep track of all the recipients. This might pose a scalability problem in large networks.

Another scalability problem becomes evident when a large number of recipients sends acknowledgements to the sender of a data packet, the network close to the sender can become congested by all the acknowledgements. This is denoted as *ACK implosion problem*.

Receiver Initiated Error Recovery

The scalability problems of the sender initiated protocols can be resolved by using negative rather than positive acknowledgements. While positive acknowledgements were sent by the recipients when data packets arrived, negative acknowledgements are sent by the recipients when data packets do not arrive. Because it is the receiver who is responsible for tracking whether packets need to be resent, the term *receiver initiated protocol* is sometimes used.

The receiver must know what data packets it should receive. In applications that send a steady flow of data packets, timeouts coupled with sequential numbering of data packets can be used to detect packet loss. In applications that do not send a steady flow of data packets, keepalive packets can be introduced.

In receiver initiated protocols, the sender does not know how long to remember a data packet in case it needs to be resent. Timeouts can be used to forget data packets.

The network close to the sender of a data packet can become congested when a large number of recipients detects packet loss and sends negative acknowledgements to the sender. This is denoted as *NAK implosion problem*. The problem can be alleviated by having the recipients multicast rather than unicast the negative acknowledgements, and by introducing random delays before sending the negative acknowledgement. When a recipient sees an acknowledgement of a data packet during the

random delay before sending its own acknowledgement of the same packet, it does not need to send the acknowledgement.

Tree Topology

Reliable multicast protocols exploit the tree topology of the underlying network by arranging the recipients into a logical tree that roughly copies the network topology. The sender of data packets is the root of the tree. Each node is responsible for delivering the data packets to its direct descendants.

Each node sends a *local acknowledgement* when it has received a data packet and an *aggregate acknowledgement* when it has received aggregate acknowledgements from all its direct descendants. The local acknowledgement is used for flow control. The aggregate acknowledgement is used for forgetting the data packet.

Ring Topology

Reliable multicast protocols arrange the recipients into a logical ring with a token. The node with the token sends a positive acknowledgement to the sender when it has received a data packet. The nodes without the token send a negative acknowledgement to the node with the token, rather than to the sender, when they have missed a data packet.

Ordering Guarantees

In the context of communication, ordering refers to the order in which data are received by multiple recipients, relative to the order in which the data were sent by multiple senders. The three notable ordering options are source ordering, causal ordering and total ordering.

- *Source ordering* or *sender ordering* denotes delivery of data from a single sender in the same order as it was sent.
- *Causal ordering* denotes delivery of data that is causally related in the order corresponding to the causal relation. Causal ordering guarantees that information about causes is delivered before information about their effects.
- *Total ordering* denotes delivery of data in the same order to all recipients.

The principle of providing ordering guarantees is roughly the same regardless of the particular guarantees provided. Recipients distinguish data *reception* and data *delivery*, reception being the arrival of the data at the recipient node and delivery being the arrival of the data at the recipient application. Between reception and delivery, the ordering guarantees can be enforced.

When a message carrying the data is received, it can be delivered only if there is no other message that would precede it with respect to the ordering guarantees. Message numbering can be used to make this decision.

- A sequential numbering assigned locally by the sender is sufficient to enforce source ordering. The recipient can deliver a message when its number immediately succeeds the number of the last delivered message from the same sender.
- A vector clock numbering assigned locally by the sender within the group of communicating nodes is sufficient to enforce causal ordering. The recipient can deliver a message when its number immediately succeeds the number of the last delivered message from the same group.

- A sequential numbering assigned by a central authority within the group of communicating nodes is sufficient to enforce total ordering. Again, the recipient can deliver a message when its number immediately succeeds the number of the last delivered message from the same group.

Causal Relation

Generally speaking, causal relation orders events so that causes come before effects. When considered in the context of a group of communicating processes, the causal relation is defined as follows:

- Every action performed by a process is causally related to the previous action performed by the process.
- Every action that constitutes receiving a message is causally related to the action that constituted sending the message.
- The causal relation is transitive.

Lamport Clock

Lamport clock is the precursor of vector clock. Lamport clock timestamp is an integer number maintained at each process in a group of communicating processes. The timestamp is updated using three rules:

- Whenever a process observes a significant event, it increments its timestamp.
- Whenever a process sends a message to another process, it also adds its timestamp to the message.
- Whenever a process receives a message from another process, it also adjusts its timestamp so that it is larger than the timestamp in the message but not smaller than its previous value.

The update rules are based on the definition of the causal relation. Obviously, when an event A causally precedes an event B, the Lamport clock timestamp associated with the event A is numerically smaller than the Lamport clock timestamp associated with the event B. The implication, however, does not hold in the opposite direction.

Vector Clock

Vector clock timestamp is a vector of integer numbers maintained at each process in a group of communicating processes. The size of the vector is equal to the number of processes in the group. The timestamp is updated using three rules:

- Whenever a process i observes a significant event, it increments the i-th element of its timestamp.
- Whenever a process sends a message to another process, it also adds its timestamp to the message.
- Whenever a process receives a message from another process, it also adjusts its timestamp so that all its elements are maxima from the corresponding elements of the timestamp in the message and their previous values.

Again, the update rules are based on the definition of the causal relation. Intuitively, the element i of the timestamp of process j corresponds to the latest event at process i that process j can know of.

Timestamps can be compared to each other. Timestamp A is smaller (or larger) than timestamp B if each element of A is numerically not larger (or not smaller) than the

corresponding element of B, and at least one element of A is numerically smaller (or larger) than the corresponding element of B. Timestamp A is unrelated to timestamp B if it is neither smaller nor larger.

If and only if an event A causally precedes an event B, the vector clock timestamp associated with the event A is smaller than the vector clock timestamp associated with the event B. If and only if an event A is causally unrelated to an event B, the vector clock timestamp associated with the event A is unrelated to the vector clock timestamp associated with the event B.

When used to enforce ordering guarantees on messages exchanged in a group of communicating processes, the only significant event is sending of a message. Then, the element i of the timestamp of process j corresponds to the number of messages sent by process i that process j has received. The node hosting a process can deliver a received message to the process when at most one of the elements of the message timestamp is larger than the corresponding element of the process timestamp, and larger exactly by one.

Example: Reliable Multicast Protocol

Příjemci jsou uspořádáni do logického kruhu, jeden z příjemců má token. Odesílatel posílá zprávu všem normálním multicastem a čeká na ACK od příjemce s token. Příjemce s token posílá ACK všem normálním multicastem, ostatní příjemci posílají normálním multicastem NAK s označením přeferovaného příjemce s token. Zprávy obsahují identifikaci odesílatele a lokální pořadové číslo, ACK obsahuje globální pořadové číslo. Příjemce s token jej předá svému následníkovi po odeslání ACK, následník může přijmout token pouze pokud přijal všechny zprávy s nižším globálním pořadovým číslem.

Popsaný mechanismus dovoluje source ordering pomocí lokálního pořadového čísla a total ordering pomocí globálního pořadového čísla. V kruhu délky N také omezuje potřebnou vyrovnávací paměť na N zpráv, protože všechny starší zprávy byly již nutně doručeny.

Vlastnosti se dají nastavovat u každé zprávy zvlášť. Toto nastavení ovlivňuje, kdy jsou přijaté zprávy doručeny na straně příjemce:

- Unreliable. Zpráva je doručena v okamžiku přijetí.
- Reliable Unordered. Zpráva je doručena v okamžiku přijetí. Pokud příjemce detekuje výpadek lokálního pořadového čísla, pomocí NAK požádá o zprávu příjemce s token.
- Reliable Source Ordered. Zpráva je doručena v okamžiku přijetí všech zpráv s nižším lokálním pořadovým číslem od stejného odesílatele.
- Reliable Totally Ordered. Zpráva je doručena v okamžiku přijetí všech zpráv s nižším globálním pořadovým číslem.
- K Resilient. Zpráva je doručena ve chvíli, kdy byly přijaty všechny zprávy s nižším globálním pořadovým číslem a token byl předán K-krát. Zaručuje, že nejméně K+1 uzlů přijalo paket, a tedy výpadek nejvýše K uzlů nezpůsobí ztrátu dat.
- Agreed Delivery. Jako K resilient s K = (MaxN+1)/2, MaxN je odhad nejvyššího počtu uzlů ve skupině, skupina s méně než (MaxN+1)/2 uzly nesmí pokračovat v provozu. Zaručuje atomický provoz i při rozpadu skupiny.
- Safe Delivery. Jako K resilient s K = N, N je počet uzlů ve skupině. Zaručuje atomický provoz.

Je dobře poznamenat, že protokol nedovoluje uzlu vrátit se do multicast skupiny pod stejnou identitou. Také je důležité, že všechny volby vlastností doručení ovlivňují pouze latency, nikoliv thruput.

Prikladem je TRP (Token Ring Protocol), RMP (Reliable Multicast Protocol). RMP rozsiruje TRP pro velke site, kde se pouziva hierarchie logickych kruhu (tedy, udajne, nasel jsem to v jednom prehledovem clanku, ale v samotnem popisu RMP ne). RMP mapuje multicast groups na IP multicast addresses pomoci hashovani, pokud dve skupiny dostanou stejnou IP multicast address, oddeluji svoje pakety pomoci multicast group ID.

Na detailnejsi urovni je RMP slozeny z peti algoritmu. Prvni, delivery algoritmus, je primocary podle toho, co bylo dosud popsano.

Druhy algoritmus je membership change, zajemce o pripojeni multicastuje pozadavek dokud nedostane odpoved, tu posila token site, ktere zaroven preda token novemu zajemci o pripojeni. Odpojeni vyzaduje, aby uzel zustal clenem skupiny dokud schovava pakety pro resend. Protokol zarucuje virtual synchrony, ktera rika, ze pokud nekdo joins or leaves a multicast group, vsichni se shodnou na tom, ktere zpravy byly pred a ktere po group membership change.

Treti algoritmus je fault recovery, vyvolany kdykoliv se zjisti, ze nejaky uzel multicast group selhal. Iniciator fault recovery nejprve zkousi kontaktovat vsechny uzly multicast group a vyzvedet od nich, ktere pakety naposledy prijali. Pokud zjisti, ze nekomu chybi paket, ktery nekdo jiny ma, oznami to, tohle opakuje tak dlouho, nez se vytvori nova skupina uzlu, ktere prijaly stejne pakety. Tehle skupine pak pomoci dvoufazoveho commitu oznami jeji slozeni.

Ctvrty algoritmus dovoluje uzlum mimo multicast group posilat zpravu do multicast group a vybranemu uzlu odpovedet, vynechavam :-) ...

Paty algoritmus ma na starost flow and congestion control. Je podobny Van Jacobsonove algoritmu z TCP, tedy sliding window, ktere se dynamicky meni podle toho, zda se dari nebo nedari odesilat.

Rehearsal

The purpose of this section is to sketch the principles and issues related to implementing the multicast communication protocols and relate those principles and issues to the middleware that relies on the protocols.

At this point, you should be able to explain how local multicast communication provided by hardware is extended to global scale through addressing and routing and explain how the use of multicast differs from the use of multiple unicasts in this context.

You should understand how the approaches used to cope with communication failures are extended from unicast to multicast.

You should be able to define the properties of various message ordering guarantees and outline both the approaches used to provide such guarantees and the applications likely to require such guarantees.

Questions

1. The sender initiated and receiver initiated error recovery schemes differ in which of the two communicating sides is responsible for keeping track of delivered packets. Explain how this difference impacts the management of memory used to store packets.
2. The use of positive acknowledgements in multicast can lead to network congestion problems due to excessive acknowledgement traffic, also termed as ACK implosion. Outline scenarios in which this problem occurs and approaches used to remedy the problem while preserving positive acknowledgements.

3. The use of negative acknowledgements in multicast can lead to network congestion problems due to excessive acknowledgement traffic, also termed as NAK implosion. Outline scenarios in which this problem occurs and approaches used to remedy the problem while preserving negative acknowledgements.
4. Define the source ordering guarantees in the context of multicast communication and explain in what situations and why this ordering is useful.
5. Define the causal ordering guarantees in the context of multicast communication and explain in what situations and why this ordering is useful.
6. Define the total ordering guarantees in the context of multicast communication and explain in what situations and why this ordering is useful.
7. In a form of algorithms used by the senders and the receivers, sketch the approach used to achieve source ordering in multicast communication.
8. In a form of algorithms used by the senders and the receivers, sketch the approach used to achieve causal ordering in multicast communication. Use of a distributed algorithm is considered better than use of a centralized one.
9. In a form of algorithms used by the senders and the receivers, sketch the approach used to achieve total ordering in multicast communication. Use of a distributed algorithm is considered better than use of a centralized one.

   Hint: Stronger ordering does not necessarily require more complex logical clock.

10. The Lamport Clock is a logical clock used in some distributed algorithms. Outline the algorithm used to calculate the Lamport Clock timestamp and explain what are the useful properties of the timestamp.
11. The Vector Clock is a logical clock used in some distributed algorithms. Outline the algorithm used to calculate the Vector Clock timestamp and explain what are the useful properties of the timestamp.
12. Present a suitable example of a multicast communication in which sender ordering is violated. Explain why the ordering is violated. Use a notation in which S(A,X->B,C) denotes node A sending message X to nodes B and C and R(A,X) denotes node A receiving message X.
13. Present a suitable example of a multicast communication in which causal ordering is violated, but less strict ordering guarantees are preserved. Explain why the ordering is violated. Use a notation in which S(A,X->B,C) denotes node A sending message X to nodes B and C and R(A,X) denotes node A receiving message X.
14. Present a suitable example of a multicast communication in which total ordering is violated, but less strict ordering guarantees are preserved. Explain why the ordering is violated. Use a notation in which S(A,X->B,C) denotes node A sending message X to nodes B and C and R(A,X) denotes node A receiving message X.

Exercises

1. Navrhněte přenosový protokol, který bude zaručovat spolehlivé doručování zpráv od jednoho odesílatele více příjemcům. Váš návrh by měl definovat tyto funkce:

```
void ReliableSend (tMsg *pMessage, tAddrList *pTargetList);
  // Odeslání zprávy, blokuje do přijetí zprávy všemi příjemci
```

```
void ReliableReceive (tMsg &*pMessage, tAddr &*pSource);
  // Příjem zprávy, blokuje do přijetí zprávy, zaručuje
  // právě jeden příjem nepoškozené odeslané zprávy
```

Váš návrh by měl používat tyto funkce:

```
void UnreliableSend (tMsg *pMessage, tAddr *pTarget);
  // Odeslání zprávy, neblokuje, nemusí odeslat

void UnreliableReceive (tMsg &*pMessage, tAddr &*pSource, int iTimeout);
  // Příjem zprávy, blokuje do timeoutu, může přijmout
  // poškozenou zprávu nebo tutéž zprávu vícekrát
```

Dále předpokládejte existenci rozumných funkcí pro manipulaci se zprávami jako je jejich vytváření a rušení, nastavování a dotazování atributů přenášených spolu s obsahem zprávy a podobně a existenci rozumných funkcí pro manipulaci se seznamem adres jako je jeho procházení.

Zadání vyžaduje návrh protokolu pro spolehlivý multicast nad nespolehlivým unicastem, na rozdíl od obvyklejšího návrhu spolehlivého multicastu nad nespolehlivým multicastem. Vysvětlete, jaký má toto omezení vliv na vlastnosti vašeho návrhu.

> **Hint:** For those unfamiliar with the particular programming language, note that the sending functions expect the message and the address as their inputs, but the receiving functions as their outputs.

Addressing By Content

To be done.

References

1. Sameh El-Ansary, Seif Haridi: An Overview of Structured P2P Overlay Networks. http://eprints.sics.se/237

Rehearsal

At this point, you should be able to outline common approaches used to achieve addressing by content, such as constrained flooding or distributed hashing. You should understand the limits of those approaches in terms of scalability and its impact on parameters such as network load, routing information size, routing path length, resiliency to failures.

You should be able to explain how common applications of addressing by content work.

Questions

1. Explain how distributed hashing can be employed to implement a simple distributed file sharing service. Discuss the limitations of such an implementation in terms of the search query complexity.

Messaging Interfaces

The basic functions of a messaging interface allow sending and receiving of messages and typically take the form of a pair of functions, one for sending and one for receiving of messages. In its minimum form, the arguments of the send function typically consist of the message to be sent and the address the message is sent to, while the arguments of the receive function typically consist of the buffer for the message to be received.

Messages

The message itself can be structured to a different degree. On one end of the spectrum is an unstructured message that looks like an array of bytes to the middleware. An example of such an interface can be the UNIX IPC API. On the other end of the spectrum is a structured message where the middleware requires that every item of the message is both syntactically and semantically typed. An example of such an interface can be the MACH IPC API or MPI or MQSeries.

Addresses

The address that a message is sent to can either denote a process or a queue. Sometimes, the address that a message is to be received from can also be specified.

Blocking

The functions of a messaging interface can differ in their blocking properties. Formally, these are related to the guarantee that a function returns in a finite number of its own steps regardless of the actions of other communication participants:

- A *blocking* function can resort to waiting during the course of its execution. The send operation typically blocks when the buffers at the sender side become full. The receive operation typically blocks when the data buffers at the receiver side become empty.
- A *nonblocking* function never resorts to waiting. In a situation where a blocking operation would block, a nonblocking operation either returns with an indication of failure or returns and uses other means to indicate completion, such as *callback* or *polling*.

In simple communication scenarios, the use of blocking operations typically yields shorter and more readable code than the use of nonblocking operations. In complex communication scenarios, where parallel execution of multiple actions is required, the use of nonblocking operations might be necessary.

Synchronization

The functions of a messaging interface can differ in the degree of synchronization between the communication participants:

- The completion of a send operation in *synchronous* communication implies that the data has been received by the receiver. The term emphasizes that communication synchronizes the execution of the sender with the execution of the receiver.

- The completion of a send operation in *asynchronous* communication does not imply that the data has been received by the receiver.

There is a distinction between blocking and synchronization. While blocking is typically understood to refer to programming language functions, synchronization is often understood to refer to communication operations which may involve calling several programming language functions. Thus, a nonblocking synchronous communication is not an oxymoron, but a legal combination of properties which can be achieved by employing a nonblocking send function that uses callback or polling to notify of message reception.

Underlying Protocol Issues

To be done.

Rehearsal

Based on your knowledge of how message communication is used, you should be able to design an API suitable for a specific application scenario, and to explain how your choice of the various API properties fits the scenario.

Exercises

1. Design an interface of a messaging middleware that is suitable for a highly efficient transport of messages between processes of a tightly coupled homogeneous multiprocessor cluster, to be used for scientific calculations. Explain your design choices and advocate the suitability of your design.

   Hint: By definition, messaging certainly is sending and receiving of messages, but the intended application influences what form these functions take. What functions could be useful for scientific calculations ?

   Where details are concerned, think about the issues related to highly efficient transport. Does the fact that the interface runs on a tightly coupled homogeneous cluster help in any way ?

2. Design an interface of a messaging middleware that is suitable for an internet wide transport of messages between heterogeneous desktop computers, to be used for thick client information system running on multiple client platforms. Explain your design choices and advocate the suitability of your design.

   Hint: By definition, messaging certainly is sending and receiving of messages, but the intended application influences what form these functions take. What functions could be useful for thick clients in information systems ?

   Where details are concerned, think about the issues related to heterogeneity. Does the fact that the interface should support heterogeneous network and heterogeneous clients matter in any way ?

Streaming Interfaces

To be done.

Rehearsal

To be done.

Remote Procedure Call

Assume an architecture where a server performs functions as requested by a client, and where messages are used for communication. The communication will follow a simple pattern where the client constructs a message describing what function it wants performed and sends it to the server, who parses the message and performs the function as requested. If a result is to be returned, the server then constructs a message describing what result it returns and sends it to the client, who parses the message and uses the result.

If a messaging interface were used to implement this pattern, constructing and parsing the messages would be a responsibility of the application, with middleware only taking care of sending and receiving the messages. A middleware with a remote procedure call interface takes advantage of the fact that the format of the messages is entirely determined by the signatures of the functions to be performed. This makes it possible to delegate the responsibility of constructing and parsing the messages to the middleware, provided that the middleware knows the signatures.

For sake of simplicity and elegance, the interface between the application and the middleware is made to match the signature of the functions to be performed. The implementation of the interface inside the middleware, called *stub*, does the necessary constructing and parsing of the messages, termed *marshaling* and *unmarshaling*.

To be done.

The idea of masking remote communication behind an automatically generated facade of a procedure call has also been extended to support objects. Client proxies and server servants.

Stub Generation

To be done.

Generation at compile time or run time. Signature in implementation language, additional information. Signature in interface definition language, language mapping.

Argument Passing

To be done.

Transparency in argument passing. Formatting. References. Aliasing.

Parallelism

To be done.

Threading models on server. Main issues: parallelism in server implementation code, scalability with high number of requests and high number of clients, real time guarantees, integration with other frameworks. Single threaded. Thread per connection. Thread per request. Thread pool (naive implementation, leader-follower implementation).

Threading models on client. Main issues: callback programming model, colocated programming model.

Lifecycle

To be done.

Underlying Protocol Issues

To be done.

Choice of transport: datagram vs connection (reliability, buffering, blocking), multiplexed and bidirectional connections (impact on threading model). Message types: requests, replies, exceptions and control. Message format: headers (size and number of reads and fragmentation), body per type. Encoding: canonical vs flexible vs negotiation, usable buffer vs copying extraction, self descriptive and type description (impact on performance and firewalling), basic types, constructed types, method identification and object identity (dispatching efficiency). Specialized protocols (shared memory, weird transports, pluggability and reasonable common base).

Rehearsal

At this point, you should understand how remote procedure calls work both in procedure oriented and in object oriented environments. You should be able to list the steps taken to generate and instantiate the client and server stubs and to execute a remote procedure call.

You should be able to tell what information is required for generating the client and server stubs and explain where this information comes from. You should be able to tell how specific types of arguments are marshalled and explain how marshalling can influence the argument passing semantics.

Based on your knowledge of how remote procedure calls are used, you should be able to design an API suitable for a specific application scenario, and to explain how your choice of the various API properties fits the scenario.

Questions

1. When invoking a procedure that passes an argument by value, RPC middleware has to handle situations where binary representation of the values differs between the client and the server. Outline and discuss the approaches used to do this.
2. When invoking a procedure that passes an argument by reference, RPC middleware has to handle situations where the reference has no meaning outside the address space of the client or the server. Outline and discuss the approaches used to do this.
3. Napište kód stubu na straně klienta a na straně serveru tak, aby zprostředkoval vzdálené volání funkce `int read (int iFile, void *pBuffer, int iSize)`, která prečte data z otevřeného souboru. Váš návrh by měl používat tyto funkce:

```
void ReliableSend (tMsg *pMessage, tAddr *pTarget);
  // Odeslání zprávy, blokuje do přijetí zprávy

void ReliableReceive (tMsg &*pMessage, tAddr &*pSource);
  // Příjem zprávy, blokuje do přijetí zprávy, zaručuje
  // právě jeden příjem nepoškozené odeslané zprávy
```

Dále předpokládejte existenci rozumných funkcí pro manipulaci se zprávami jako je jejich vytváření a rušení, přístup k obsahu zprávy a podobně.

Distributed Shared Memory

To be done.

Rehearsal

To be done.

Chapter 3. Persistence

To be done.

Chapter 4. Replication

To be done.

Chapter 5. Deployment

To be done.

Clouds

To be done.

Components

To be done.

Chapter 6. Mobility

Protocols

[Perkins C. E.: RFC2002 IP Mobility Support, 10/1996, read 7/2004] Existuje ve dvou verzích, standard pro IPv4, draft pro IPv6. Terminologie, definuje MOBILE NODE jako uzel, který má pevnou IP ve své domácí síti, ale pohybuje se v cizích sítích, CORRESPONDENT NODE jako uzel, který komunikuje s MN, HOME AGENT jako uzel, který zůstává na domácí síti MN a zprostředkovává komunikaci, FOREIGN AGENT jako uzel, který je na cizí síti MN a zprostředkovává komunikaci. HA a FA jsou MOBILE AGENTS.

Scénář komunikace je jednoduchý. Když se MN připojí k síti, buď počká na AGENT ADVERTISEMENT zprávu nebo o ní požádá AGENT SOLICITATION zprávou. Odpoví buď HA nebo FA. Pokud odpověděl HA, MN je ve své domácí síti a nic se neděje. Pokud odpověděl FA, MN je v cizí síti, podle konfigurace buď FA pošle svou CARE OF IP adresu v AGENT ADVERTISEMENT nebo MN získá COLOCATED CARE OF IP adresu z DHCP. MN pak pošle svému HA zprávu REGISTRATION REQUEST, buď přímo nebo prostřednictvím FA, HA odpoví zprávou REGISTRATION REPLY a otevře tunel, kterým bude dále posílat všechny zprávy pro MN na FA, který je doručí MN.

Zatímco komunikace ve směru z MN na CN se neliší od přímé komunikace z CARE OF IP adresy, nepříjemná je režie při komunikaci ve směru z CN na MN. Proto se definuje mechanismus [Perkins C. E.: Route Optimization in Mobile IP, 2/1999, read 7/2004]. Základem je udržování BINDING CACHES na CN, ve kterých je mapování z IP adresy MN na CARE OF IP adresu MN. BINDING CACHES jsou aktualizovány zprávami BINDING UPDATE od HA.

Aby se zabránilo ztrátě zpráv při pohybu MN, může MN ve zprávě REGISTRATION REQUEST požádat nový FA, aby uvědomil starý FA pomocí BINDING UPDATE o nové CARE OF IP adrese MN. Starý FA pak bude novému FA nějakou dobu doručovat zprávy pro MN, které používaly starou místo nové CARE OF IP adresy MN.

Aby se celý mechanismus nedal snadno zneužít, je většina zpráv autentizována s použitím keyed MD5 [Bellare M., Canetti R., Krawczyk H.: Keyed Hash Functions for Message Authentication, proceedings of CRYPTO'96, LNCS 1109, 1996]. V principu jde o doplnění zpráv variantou MD5 obsahu, která je inicializována tajným klíčem místo standardního 0123456789abcdeffedcba9876543210H.

Novější je [Perkins C. E.: RFC3220 IP Mobility Support].

Novější je [Perkins C. E.: RFC3344 IP Mobility Support].

Chapter 7. Systems

GM

GM je knihovna pro přenos zpráv po síti Myrinet. Procesy v GM vlastní send a receive tokens, které reprezentují buffer na jednu zprávu. Adresace v GM používá ID uzlů, která přiděluje daemon současně vyhledávající routes, programy pak k odesílání používají porty, což jsou struktury v podstatě reprezentující otevřené spojení na daném rozhraní.

K odeslání zprávy se volá gm_send_with_callback (data, size, priority, callback, context, port, target ...), které odebere jeden send token a asynchronně odešle zprávu, pak gm_receive (), které čeká na GM událost, při příjmu správné události pak gm_unknown (), které zpracovává default události, z něj GM zavolá callback (context, status ...) a vrátí jeden send token.

```
void gm_send_with_callback (
  struct gm_port *p,
  void *message,
  unsigned int size,
  gm_size_t len,
  unsigned int priority,
  unsigned int target_node_id,
  unsigned int target_port_id,
  gm_send_completion_callback_t callback,
  void *context
);

gm_recv_event_t *gm_receive (
  gm_port_t *p);

void gm_unknown (
  gm_port_t *p,
  gm_recv_event_t *e
);
```

K přijetí zprávy konkrétní délky a priority se volá gm_provide_receive_buffer (port, buffer, size, priority ...), které odebere jeden receive token, pak gm_receive (), které čeká na GM událost, při příjmu správné události GM vyplní buffer a vrátí jeden receive token.

```
void gm_provide_receive_buffer_with_tag (
  gm_port_t *p,
  void *ptr,
  unsigned size,
  unsigned priority,
  unsigned int tag
);
```

Od aplikací se očekává, že dají GM vždy nejméně jeden, lépe dva buffery, pro každou délku a prioritu zprávy, kterou mohou přijmout. To dovoluje aplikaci vždy jeden buffer zpracovávat, zatímco GM druhý plní. Všechny buffery musí být alokovány pomocí gm_dma_alloc (), aby je GM mohla efektivně použít.

Hlavní myšlenkou je rozhraní, kde se nemusí během komunikace nic dynamicky alokovat, což dovoluje dosáhnout minimální režie knihovny.

http://www.myri.com/scs/GM/doc/html

http://www.myri.com/scs/GM-2/doc/html

Rehearsal

Questions

1. To achieve maximum efficiency, the GM library is designed to minimize the need for copying data while communicating. Explain how the design minimizes data copying.

   Hint: Think about situations that necessitate copying of data.

IBM MQ

IBM MQ (Message Queueing) is an IBM enterprise messaging middleware. The middleware puts emphasis on robustness and portability, supporting multiple platforms and languages including many legacy systems. The middleware is marketed under multiple names that usually include reference to message queueing.

Among the interfaces provided by IBM MQ is MQI (Message Queue Interface), which defines basic functions for connecting to the middleware and sending and receiving messages, and basic structures passed as arguments.

Queues And Messages

Implementace je zalozena na posilani zprav mezi procesy skrz fronty. V systemu jsou krom procesu spravci front, k tem se proces musi nejdriv pripojit volanim MQCONN (in QMName, out CHandle), na konci odpojit volanim MQDISC (in Handle). Pak se otevre prislusna fronta zprav volanim MQOPEN (in CHandle, inout Descriptor, out OHandle), deskriptor obsahuje hlavne jmeno fronty, na konci se zavre volanim MQCLOSE (in CHandle, in OHandle).

Do otevrene fronty je mozne zapisovat zpravy volanim MQPUT (in CHandle, in OHandle, inout Descriptor, inout Options, in Buffer, in Length).

Z otevrene fronty je mozne cist zpravy volanim MQGET (in CHandle, in OHandle, inout Descriptor, inout Options, in Length, inout Buffer).

Volani prijme prvni zpravu podle priority pouze pokud deskriptor obsahuje unique ID MQMI_NONE a correlation ID MQCI_NONE. Jinak se daji vyplnit a pak se prijme prvni zprava, ktera takove ID ma. Podle options je volani blokujici nebo s timeoutem, orezavajici prilis dlouhe zpravy nebo vracejici chybu. Uvnitr options se da take pozadat o asynchronni prijem, potom se MQGET vrati hned a program dostane pozdeji signal.

Deskriptor obsahuje typ zpravy (datagram, request, reply, report), lifetime, kodovani (numeric, charset), format (string kteremu rozumi prijemce), prioritu (cislo), persistenci (flag), unique ID a correlation ID, frontu kam odpovedet, identifikaci odesilajici aplikace a uzivatele.

Message Encoding

Pokud je zprava nektereho ze systemovych formatu a specifikuje se option MQGMO_CONVERT, prekoduje ji middleware. Pokud je zprava uzivatelskeho formatu, muze aplikace nainstalovat svuj vlastni konvertor do middleware.

Tomuhle konvertoru se vtipne rika data conversion exit, MQSeries na ne maji generator.

Generatoru se predhodi struktura zpravy v C syntaxi, jen datove typy jsou omezene na MQBYTE, MQCHAR, MQSHORT, MQLONG. Vystup generatoru se vlozi do dodaneho skeleton kodu, vysledek se prelozi jako shared library a nainstaluje se do middleware.

Zpravy zpravidla prekodovava posledni prijemce, kvuli efektivite. V pripade potreby je mozne pozadat o prekodovani i odesilatele nebo frontu.

Miscellanea

Middleware nabizi jeste transakce nad zpravami. Zahajeni MQBEGIN (in CHandle), ukonceni MQCMIT (in CHandle), vraceni MQBACK (in CHandle).

Aby aplikace, ktere cekaji na zpravy, nemusely bezet stale, nabizi MQSeries mechanismus pro spousteni on demand, nazyvany triggers. Uzivatel muze specifikovat podminku (minimalni priorita, minimalni pocet cekajicich zprav), jeji splneni pak vyvola zaslani trigger message do lokalni fronty. Ceka se, ze na lokalni fronte bude cekat trigger monitor, ktery jako odezvu na trigger message spusti prislusnou aplikaci. Pokud je trigger monitor vyrazne mensi nez aplikace, usetri se.

Web Services

Web services standardize an infrastructure for integrating information systems in the environment of the Internet. The web services are based on the Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) standards.

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.

```
<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.

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.

```
<?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. -->
```

UDDI

The UDDI standard [6] by the UDDI Consortium defines a web service capable of registering and locating other web services. UDDI relies on the industry classification standards, such as NAICS or UNSPSC, to provide a hierarchy to sort the services by. For each service, UDDI records its position in the hierarchy together with an information about its provider and its WSDL description.

Service Composition

Service orchestration (with central coordinator) and service choreography (without central coordinator).

BPEL for orchestration. Primitive activities - wait for request, send reply, invoke service, assign variable, throw exception, delay. Structured activities - synchronous sequence, parallel flow, switch, while.

```
<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>
```

Rehearsal

Questions

1. Explain how the SOAP, WSDL and UDDI standards cooperate in the web services environment.
2. Describe the reasons for choosing XML as the transport encoding in web services.

CAN

CAN is a middleware that provides basic support for distributed hashing. CAN assigns random unique coordinates and rectangular zones of responsibility within a coordinate space to nodes. Given key coordinates, CAN can deliver a message to a node responsible for the zone with the key coordinates. For N nodes, it takes O(sqrt N) steps to deliver the message, provided each node maintains a routing table of constant size. CAN configuration allows the exponent of the root to be set to an arbitrary number.

Identifiers as coordinates, zones of responsibility. Routing along a straight line. Periodic neighbor checking, taking over zones.

References

1. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker: A Scalable Content-Addressable Network.

DCE

DCE (Distributed Computing Environment) je standard OSF (The Open Group, původně asi Open Systems Foundation). V roce 1989 vydalo OSF dokument Request For Technology, ve kterém vyzvalo k příspěvkům pro distributed computing standard. Kolem roku 1991 byly přijaty příspěvky pro threads (DEC Concert Multithread Architecture), remote procedure call (Apollo/HP/NEC NCS RPC), security (MIT Kerberos), directory (DEC DECdns), time (DEC Distributed Time Synchronization), file (Andrew File System).

Threads

Threads jsou založené na pthreads a podle toho vypadají, asi nemá smysl se o nich o moc víc zmiňovat. Nemají vlastně nic společného s distribucí, součástí DCE jsou jen proto, že nějaké thready jsou v implementaci potřeba.

Remote Procedure Call

RPC je běžné, proxy na klientovi i dispatcheru na serveru se říká stub, servery mají interface definovaný v IDL, každý interface ma UUID a seznam metod, syntaxe zhruba:

```
[uuid (12345678-9ABC-DEF0-1234-56789ABCDEF0), version (1.0)]
interface Foo
{
```

```
  [context_handle] void *Bar ([in] long X, [out] long *Y);
  [idempotent] void Rab ([in, ref, string] char *A);
};
```

UUID is an identifier that is universally unique (128 bits, standardized by X.667 or RFC 4122, generated with limited coordination by relying on combinations of network address and time, on random numbers, or on hashes of other unique identifiers).

Zajimave jsou snad atributy idempotent (rika ze je bezpecne pouzit at least once misto at most once semantics), broadcast (rika ze operaci maji vykonat vsechny viditelne servery, ktere ji umi, vraci se prvni vysledek), maybe (rika ze se nemusi zarucovat vykonani operace). U pointeru se da uvest, jestli je reference (nesmi byt NULL, nesmi se menit uvnitr volani) nebo pointer (muze byt NULL, muze se zmenit a budou spravne vracena nova data). Jako typ se da uvest pipe pro prenos streamu dat, potom smer argumentu urcuje push nebo pull pipe, argument se namapuje jako struktura s metodami pull (buffer), push (buffer), alloc (size), klient si je pak implementuje jak potrebuje, server podle typu pipe vola bud pull nebo push, ktere se doruci zpet na klienta. Jen nejde kombinovat vice pipes, tedy jde, ale musi se vyprazdnovat jedna po druhe :-).

Krom kontaktovani serveru pomoci stringove formy object reference je k dispozici directory service s automatickym, implicitnim a explicitnim binding. Automaticke binding spoji klienta s nejblizsim kompatibilnim serverem, implicitni binding dovoli klientovi uvest binding handle v globalni promenne a pouzivat jej pro vsechna volani daneho interface, explicitni binding dovoli klientovi uvadet handle pro kazde volani daneho interface explicitne.

Server vypada normalne, pri startu musi registrovat kazdy podporovany interface volanim rpc_server_register_if (iface_handle ...), registrovat se u directory service pomoci rpc_ns_binding_export (name ...), registrovat svuj endpoint pomoci rpc_ep_register (...), zavolat smycku dispatcheru pomoci rpc_server_listen (...). Pokud chceme uvazovat objektove RPC, je potreba kazdeu objektu priradit UUID a registrovat o neco sloziteji, skipped.

Directory

Directory definuje Cell Directory Service a Global Directory Service. Format jmen je podobny Unixu, forward slashes, /.:/ znaci root lokalni Cell Directory Service, jinak nic zvlastniho.

Rehearsal

The notable features of DCE whose purpose and design you should understand include:

- The interface definition language with its use of annotated C types.
- The mechanism for distinguishing interfaces based on UUID.
- The mechanism for opening a stream between the client and the server based on pipes.

Questions

1. DCE relies on UUID (Universally Unique Identifier) as a unique identifier to distinguish interfaces. Explain how an UUID can be generated so that its uniqueness is guaranteed.
2. Using an argument with the pipe annotation, DCE allows a stream between the client and the server to be created within the context of a remote call. The

argument with the pipe annotation is mapped to a pair of push and pull functions. Explain how these functions are used.

DDS

To be done.

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 model between the 2.x and 3.x revisions due to the introduction of language annotations. The text in this section therefore deals separately with the 2.x and 3.x versions where necessary.

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, persistency, transactions, and makes the beans accessible to the clients.

Beans come in four distinct classes, namely *stateful session beans*, *stateless session beans*, *message driven beans* and *entities*. Each class is tailored to fill a specific role in a component application. For maximum simplicity, method invocations on all types of beans are serialized.

Stateful Session Beans

Stateful session beans are intended to represent stateful conversations with clients of a component application. A stateful session bean looks like an object with a business interface (EJB 3.0 and above) or an object with a remote interface and a factory with a home interface (EJB 2.1 and below).

```
public interface ASessionBeanHome extends javax.ejb.EJBHome
{
  public ASessionBean createOneWay (int iArgument)
    throws RemoteException, CreateException;
  public ASessionBean createAnotherWay (int iArgument, String sArgument)
    throws RemoteException, CreateException;
}

public interface EJBHome extends Remote
{
  public void remove (Handle handle) throws RemoteException, RemoveException;
  ...
}

public interface ARemoteInterface extends javax.ejb.EJBObject
{
  public void myMethodOne (int iArgument) throws RemoteException { ... }
  public int myMethodTwo (Object oArgument) throws RemoteException { ... }
}

public class ASessionBean implements javax.ejb.SessionBean
{
  // Method that provides reference to standard session context object
  public void setSessionContext (SessionContext sessionContext) { ... };
```

```
  // Method that is called after construction
  public void ejbCreateOneWay (int iArgument)
  throws RemoteException, CreateException
  { ... }
  public void ejbCreateAnotherWay (int iArgument, String sArgument)
  throws RemoteException, BadAccountException, CreateException
  { ... }

  // Method that is called before destruction
  public void ejbRemove () { ... }

  // Methods that are called after activation and before passivation
  public void ejbActivate () { ... }
  public void ejbPassivate () { ... };

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

@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 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);
  }
}
```

Lifecycle of a stateful session bean from client point of view. (EJB 3.0 and above) Created when a reference is obtained, a business method to initialize the state, a method designated as a Remove method to discard the state. (EJB 2.1 and below) Created when a createXxx method is called on home interface, delivered as an ejbCreateXxx method to initialize the state, a remove method to discard the state.

Lifecycle of a stateful session bean from container point of view. Activation and passivation, preserves conversational state as transitive closure of field values using serialization.

Stateless Session Beans

Stateless session beans for stateless services. Looks like a stateful session bean. Lifecycle from client point of view, no need for explicit discarding of the state. Lifecycle from container point of view, no need for activation and passivation.

```
// Business interface dependency injection
// Instance per session
@EJB Cart cart;

// Business interface naming service lookup
// Instance per session
@Resource SessionContext ctx;
Cart cart = (Cart) ctx.lookup ("cart");

// Home interface dependency injection
// Instance created explicitly
@EJB CartHome cartHome;
Cart cart = cartHome.createLargeCart (...);

// Home interface naming service lookup
// Instance created explicitly
@Resource SessionContext ctx;
CartHome cartHome = (CartHome) ctx.lookup ("cartHome");
Cart cart = cartHome.createLargeCart (...);
```

Message Driven Beans

Message beans for stateless message consumers. Looks like a JMS destination and implements a JMS listener. Lifecycle trivial since it is stateless.

```
// Destination dependency injection
@Resource Queue stockInfoQueue;

// Destination naming service lookup
Context initialContext = new InitialContext ();
Queue stockInfoQueue =
  (javax.jms.Queue) initialContext.lookup
    ("java:comp/env/jms/stockInfoQueue");
```

Entities

Entity beans for database entities. Looks like a class designated as an Entity class (EJB 3.0 and above) or an object with a remote interface and a factory with a home interface (EJB 2.1 and below).

```
public interface AccountHome extends javax.ejb.EJBHome
{
  public Account create (String firstName, String lastName, double initialBalance)
    throws RemoteException, CreateException;
  public Account create (String accountNumber, double initialBalance)
    throws RemoteException, CreateException, LowInitialBalanceException;
  public Account createLargeAccount (String firstname, String lastname, double initialB
    throws RemoteException, CreateException;
  ...
  public Account findByPrimaryKey (String AccountNumber)
    throws RemoteException, FinderException;
  ...
}

public interface EJBHome extends Remote
{
```

```
  public void remove (Object primaryKey) throws RemoteException, RemoveException;
}

@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;

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

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

@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) { ... }
}

// Home interface naming service lookup
Context initialContext = new InitialContext ();
AccountHome accountHome =
  (AccountHome) initialContext.lookup
    ("java:comp/env/ejb/accounts");

// Creation
accountHome.createLargeAccount (...);

// Location
```

```
accountHome.findByPrimaryKey (...);

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);
  ...
}

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);
}
```

Persistence of an entity bean. (EJB 3.0 and above) Instance variables are made persistent, can be fields or properties, types are limited roughly to primitive types, serializable types, collections. Primary key variable annotated as an Id. Entity manager

provides finder methods. (EJB 2.1 and below) Container managed persistence generates accessor methods for fields described by abstract persistence schema in the deployment descriptor. Bean managed persistence requires implementation of manual database access and ejbLoad and ejbStore methods. Home interface provides finder methods.

Transactions

The flat transaction model is supported. Depending on the configuration of the component application, 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 use the UserTransaction interface of JTA to 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:

- The *not supported* transaction attribute instructs the container to suspend the calling transaction, if any, while executing the bean method.
- The *required* transaction attribute instructs the container to use the calling transaction while executing the bean method, and to create a new transaction for the execution of the bean method if there is no calling transaction.
- The *supports* transaction attribute instructs the container to use the calling transaction while executing the bean method, and to execute the bean method outside transaction if there is no calling transaction.
- The *requires new* transaction attribute instructs the container to suspend the calling transaction, if any, and to create a new transaction for the execution of the bean method.
- The *mandatory* transaction attribute instructs the container to execute the bean method inside the calling transaction, and to throw an exception if there is no calling transaction.
- The *never* transaction attribute instructs the container to execute the bean method outside transaction, and to throw an exception if there is a calling transaction.

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 of JTA to receive the afterBegin, beforeCompletion, afterCompletion notifications. These can be used to commit or rollback the state of the session bean explicitly.

Some limitations exist. (EJB 2.1 and below) Entity beans must use container demarcated transactions. (EJB 3.0 and above) Entity beans must use the calling transaction.

Deployment

Deployment. Descriptor in bean package. Bean provider specifies bean name, type, class, business interface and possibly home and remote interfaces, transaction demarcation, persistence management, abstract persistence schema, external references, configuration properties. Application assembler adds reference bindings.

Rehearsal

The notable features of EJB whose purpose and design you should understand include:

- The notion of beans as independently deployable components and containers as standardized execution environments.
- The lifecycle support for individual bean types from the client and server points of view.
- The support for persistency through declaration of persistent entities.
- The support for transactions through declaration of transactional attributes.

Questions

1. Explain what a *bean* and a *container* is in EJB.
2. The EJB standard defines stateful session beans, stateless session beans, message driven beans and entities. Describe the basic properties and the intended application of these four types of beans.
3. Describe the lifecycle of a stateful session bean in EJB, that is, when instances of the bean are, or appear to be, created and destructed, from both the client and the server points of view.
4. Describe the lifecycle of a stateless session bean in EJB, that is, when instances of the bean are, or appear to be, created and destructed, both from the client and the server points of view.
5. Describe the lifecycle of an entity bean in EJB, that is, when instances of the bean are, or appear to be, created and destructed, both from the client and the server points of view.

JMS

JMS (Java Message Service) is a Sun standard interface for accessing 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.

Connections and Sessions and Contexts

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.

```
// 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.

```
// 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.

```
// 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) { ... }
```

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.

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

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

Messages

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

- A unique message identifier is stored in the `JMSMessageID` fields by the middleware. The sender can relate a message to another message by filling the `JMSCorrelationID` field.
- The destination address is stored in the `JMSDestination` field by the middleware. The sender can specify a destination address of an expected reply by filling the `JMSReplyTo` field.
- The `JMSType` field specifies the message type, understood only by the sender and the recipient.
- The middleware fills the `JMSTimestamp` field with the send timestamp. The `JMSDeliveryTime` field provides the earliest delivery time, computed from the send timestamp and the minimum message delivery delay. The `JMSExpiration` field provides the expiration timestamp, computed from the send timestamp and the message lifetime.
- The `JMSPriority` field specifies message priority.
- The `JMSDeliveryMode` filed determines whether the message is persistent.
- The middleware indicates repeated delivery, due to session recovery, using the `JMSRedelivered` field. The delivery count is reported in an associated property (JMS 2.0 and above).

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`.

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.

```
// 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.

```
// 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.

```
// Uses the classic API.

TextMessage message;

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

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

// 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.

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.

```
// Uses the classic API.

TextMessage oMessage;

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

// Uses the classic API.

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

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

// 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.

```
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.

```
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.

```
MessageConsumer consumer;

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

Rehearsal

Questions

1. Explain how the point-to-point message delivery model of JMS works.
2. Explain how the publish-subscribe message delivery model of JMS works.

MPI

MPI (Message Passing Interface) je rozhraní pro psaní paralelních aplikací komunikujících pomocí zasílání zpráv. Obsahuje mapovani do Fortranu, C, C++. Zarucuje prenositelnost zprav.

Cela knihovna se inicializuje pres MPI::Init (int argc, char *argv[]), zavira pres MPI::Finalize (). Procesy jsou rozdeleny do skupin reprezentovanych komunikatory, ty se daji dynamicky vytvaret, vzdy existuje skupina MPI_COMM_WORLD pro vsechny procesy v ramci aplikace. Proces je identifikovan pomoci ranku, coz neni nic jineho nez jeho poradove cislo od 0 uvnitr skupiny. Pro komunikaci jsou pak k dispozici dva mechanismy, message passing a remote memory access.

Peer To Peer Communication

Message passing obsahuje radu funkci na vsechno mozne, ktere obsahuji popis datovych typu, ktere prenaseji. Zakladni jsou unicast funkce MPI::Send (void *buffer, int count, datatype &type, int dst, int tag, int comm) a MPI::Recv (void *buffer, int count, datatype &type, int src, int tag, int comm, status *status). Zprava je prijata pokud souhlasi pozadovany source a tag, da se dat MPI_ANY_TAG a MPI_ANY_SOURCE, pocet prijatych polozek se pozna z argumentu status. Zprava v Recv musi byt typove stejna jako v Send a musi se vejit do bufferu.

Zasilani je blokujici v tom smyslu, ze po navratu ze Send je mozne prepsat buffer, a asynchronni v tom smyslu, ze Send nemusi cekat na Recv. K dispozici jsou jeste volani BSend (zarucuje buffered send, kdy se zprava uklada do fronty), SSend

(zajistuje synchronous send, kdy se ceka na prijemce) a RSend (zajistuje ready send, kdy prijemce musi byt v Recv nez odesilatel udela Send). Zarucuje se sender ordering. Pro buffered send jsou k dispozici volani MPI_BUFFER_ATTACH a MPI_BUFFER_DETACH, kterymi muze uzivatel alokovat pro MPI buffer.

Krom blokujicich operaci jsou k dispozici jeste neblokujici, ty se jmenuji MPI_ISEND, MPI_IRECV, MPI_WAIT, MPI_TEST, plus opet varianty pro buffered, synchronous a ready rezimy. ISend a IRecv vraci MPI_REQUEST, ktery se da predhodit MPI_WAIT (ceka na dokonceni) nebo MPI_TEST (rekne zda je dokonceno). Take MPI_WAITANY, MPI_WAITALL, MPI_WAITSOME, MPI_TESTANY, MPI_TESTALL, MPI_TESTSOME.

Jako drobnosti MPI_PROBE pro cekani na zpravu bez jejiho prijmuti, MPI_IPROBE pro neblokujici cekani na zpravu, MPI_CANCEL pro preruseni MPI_WAIT. Pak se take daji delat persistentni requesty pro odeslani nebo prijem paketu, ktere se daji opakovane startovat volanim MPI_START nebo MPI_STARTALL.

Group Communication

Nasleduje skupinova komunikace. Strucny seznam obsahuje:

- MPI::Comm::Bcast (odesilatel A, prijemci A ...)
- MPI::Comm::Gather (odesilatele A, B, C, prijemce ABC)
- MPI::Comm::Scatter (odesilatel ABC, prijemci A, B, C)
- MPI::Comm::Allgather (odesilatele A, B, C, prijemci ABC ...)
- MPI::Comm::Alltoall (odesilatele ABC, DEF, GHI, prijemci ADG ...)
- MPI::Comm::Reduce (odesilatele A, B, C, prijemce A+B+C)
- MPI::Comm::Allreduce (odesilatele A, B, C, prijemci A+B+C ...)
- MPI::Comm::Reduce_scatter (odesilatele ABC, DEF, GHI, prijemci A+D+G ...)
- MPI::Comm::Scan (odesilatele A, B, C, prijemci A, A+B, A+B+C)
- MPI::Comm::Barrier (rendez vous)

Pro redukce je mozne definovat user funkci, kterou MPI pousti na data pri redukovani.

Remote Memory Access

Remote memory access nejdrive specifikuje okno pameti, ktere ma byt zpristupneno ostatnim aplikacim, pomoci MPI::Win::Create (void *base, int size ... MPI::Intracomm &comm, MPI::Win &win). Data se pak do okna ciziho procesu zapisi pomoci MPI::Win::Put (void *source, int count ... void *destination ... MPI::Datatype &type, MPI::Win &win), nebo z okna ciziho procesu prectou pomoci MPI::Win::Get (void *source, int count ... void *destination ... MPI::Datatype &type, MPI::Win &win). Krom nich existuje jeste MPI::Win::Accumulate, ktery prida data do okna ciziho procesu danou operaci.

Volani jsou asynchronni, pro synchronizaci je k dispozici napriklad MPI::Win::Fence (), coz je collective call, ktery se vrati teprve az jsou operace nad pameti vykonany.

Miscellanea

Volani vzdy popisuji datovy typ, ke kteremu se pristupuje, ten je vytvoreny pomoci sady type manipulation volani podobne jako se konstruuji datove typy v beznych jazycich.

Krom toho obsahuje MPI od verze 2 takzvane generalized requests, coz je mechanismus, jak nadefinovat vlastni komunikacni primitiva. Skipped ;-) ...

Zbyvajici funkce zahrnuji naming, error handling, data type construction, praci se soubory (collective volani open a close, collective i noncollective volani read a write, synchronni i asynchronni, s popisem datovych typu).

[http://www.mpi-forum.org]

Examples

The examples were tested on Linux with the OpenMPI MPI implementation installed in a default location. The OpenMPI MPI implementation needs no special configuration to run the examples.

First, an example where the process with rank 0 broadcasts a Hello World message. All the remaining processes receive and display the message.

Example 7-1. Broadcast Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <mpi.h>

int main (int iArgC, char *apArgV [])
{
  int iRank;
  int iLength;
  char *pMessage;
  char acMessage [] = "Hello World !";

  MPI_Init (&iArgC, &apArgV);

  MPI_Comm_rank (MPI_COMM_WORLD, &iRank);

  if (iRank == 0)
  {
    iLength = sizeof (acMessage);
    MPI_Bcast (&iLength, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast (acMessage, iLength, MPI_CHAR, 0, MPI_COMM_WORLD);
    printf ("Process 0: Message sent\n");
  }
  else
  {
    MPI_Bcast (&iLength, 1, MPI_INT, 0, MPI_COMM_WORLD);
    pMessage = (char *) malloc (iLength);
    MPI_Bcast (pMessage, iLength, MPI_CHAR, 0, MPI_COMM_WORLD);
    printf ("Process %d: %s\n", iRank, pMessage);
  }

  MPI_Finalize ();

  return (0);
}
```

To compile the example, store the source in a file named Broadcast.c and use the following command:

```
mpicc Broadcast.c -o Broadcast
```

To run the example, use the following command:

```
mpirun -np 5 Broadcast
```

The -np 5 command line option tells the middleware to run the example in 5 parallel processes. The output of the example should look like this:

```
Process 0: Message sent
Process 3: Hello World !
Process 1: Hello World !
Process 2: Hello World !
Process 4: Hello World !
```

Next, an example where the processes perform a scan reduction with their rank as the input and multiplication as the operation and display the output. With rank 0 excluded from multiplication, the processes will in effect display a factorial of their rank.

Example 7-2. Scan Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <mpi.h>

int main (int iArgC, char *apArgV [])
{
  int iOutput;
  int iInput;
  int iRank;

  MPI_Init (&iArgC, &apArgV);
  MPI_Comm_rank (MPI_COMM_WORLD, &iRank);
  if (iRank == 0) iInput = 1;
             else iInput = iRank;
  MPI_Scan (&iInput, &iOutput, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
  printf ("Process %d: Factorial %d\n", iRank, iOutput);
  MPI_Finalize ();

  return (0);
}
```

To compile and run the example, use similar commands as before. The output of the example should look like this:

```
Process 0: Factorial 1
Process 3: Factorial 6
Process 1: Factorial 1
Process 4: Factorial 24
Process 2: Factorial 2
```

Rehearsal

Questions

1. Explain why all communication functions in MPI that deal with messages require a description of the message data type.
2. Pick three of the functions provided by MPI for collective communication, listed below, and explain what they do:

```
int MPI_Barrier (MPI_Comm comm);
int MPI_Bcast (void *buffer, ..., int root, MPI_Comm comm);
```

```
int MPI_Gather (void *sendbuf, void *recvbuf, ..., int root, MPI_Comm comm);
int MPI_Scatter (void *sendbuf, void *recvbuf, ..., int root, MPI_Comm comm);
int MPI_Alltoall (void *sendbuf, void *recvbuf, ..., int root, MPI_Comm comm);
int MPI_Reduce (void *sendbuf, void *recvbuf, ..., MPI_Op op, int root, MPI_Comm c
int MPI_Scan (void *sendbuf, void *recvbuf, ..., MPI_Op op, int root, MPI_Comm com
```

.NET Remoting

.NET Remoting is a remote procedure call mechanism integrated within the .NET programming environment. Standard features of the .NET environment are used both to define the remotely accessible types and to serialize the invocation arguments.

Interface

The type of a remotely accessible object is a standard .NET type, with several simple restrictions. A remotely accessible object must inherit from the `MarshalByRefObject` base class, which marks it as an object to be passed by reference. The base class also implements a reference creation method. Only serializable types can be passed by value.

```
public class Example : MarshalByRefObject
{
  public void printMessage (string message)
  {
    System.Console.WriteLine (message);
  }
}
```

The code of the stubs is generated from the type description at runtime.

Implementation

An implementation must register the type either as a well known service by calling the `RegisterWellKnownServiceType` function, or as a client activated service by calling the `RegisterActivatedServiceType` function, which internally registers a well known activation service for the type. A well known service runs in one of two server activation modes, `Singleton` or `SingleCall`. In the `Singleton` mode, a single instance of the implementation is created and used by all calls, in the `SingleCall` mode, a new instance of the implementation is created for each call.

With client activated service, the client can create instances on the server simply by calling `new`. For this, the remote implementation must be registered by the `RegisterActivatedClientType` function. To avoid the need to refer to the remote implementation type on the client, a factory interface is typically registered.

Apart from the type, the server must create and register a channel, which represents the transport layer. TCP and HTTP channels are available. The TCP channel supports binary and SOAP encoding over TCP. The HTTP channel supports SOAP encoding over HTTP.

```
TcpChannel channel = new TcpChannel (12345);
ChannelServices.RegisterChannel (channel);

RemotingConfiguration.RegisterWellKnownServiceType (
  typeof (Example), "Example",
  WellKnownObjectMode.Singleton);
```

Besides registering the type and the channel programmatically, a configuration file describing the type and the channel can be read in by a single method call to `RemotingConfiguration.Configure`.

```
TcpChannel channel = new TcpChannel ();
ChannelServices.RegisterChannel (channel);

Example obj = (Example) Activator.GetObject (
  typeof (Example), "tcp://localhost:12345/Example");

obj.printMessage ("Hello World !");
```

The call to `GetObject` requires the caller to supply the type for which the proxy is to be created. Typically, this is an interface type rather than a class type.

Lifecycle

The lifecycle of instances is directed by a system of *leases* and *sponsors*. A lease specifies the minimum time an instance should exist. When the lease expires, sponsors of the instance are queried whether they want to renew the lease. When the lease is not renewed, the instance can be reclaimed by the garbage collector. By default, an instance gets an initial lease of 5 minutes and no sponsor, leases can be set and sponsors registered programmatically.

Note that the fact that sponsors are queried makes it possible to only query one of potentially many live sponsors of an instance at the time of lease expiration.

Java 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.

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.

```
public interface Example extends Remote
{
    void printMessage (String message) throws RemoteException;
}
```

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.

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.

```
public class ExampleImpl
  extends UnicastRemoteObject
  implements Example
{
  public ExampleImpl () throws RemoteException { }
  public void printMessage (String message) { System.out.println (message); }
}

static Remote exportObject (Remote obj, int port)
static Remote exportObject (Remote obj, int port,
                            RMIClientSocketFactory csf,
                            RMIServerSocketFactory ssf)
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.

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.

Lifecycle

The lifecycle of instances is directed by a system of *leases*. A lease is requested by a client at unmarshalling of incoming object reference, the lease is renewed by the client periodically and returned by the client after the object reference is garbage collected. The lease duration is configured in system properties, default is 10 minutes. To prevent errors due to network delays, clients renew lease 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.

```
public interface Unreferenced
{
  void unreferenced ();
}
```

Persistent objects can inherit from the `Activatable` class, which registers the object instance with the RMI daemon for later activation.

Naming

Naming uses the **rmiregistry** server to register object references under string names and to look up the references using the names.

```
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.

```
// Server side registration.
ExampleImpl obj = new ExampleImpl ();
Naming.rebind ("//localhost/Example", obj);

// Client side lookup.
Example obj = (Example) Naming.lookup ("//localhost/Example");
obj.printMessage ("Hello World !");
```

Rehearsal

Questions

1. The RMI technology uses standard Java interface definitions to describe the remotely accessible interfaces. Explain the limitations imposed on the interfaces by the RMI technology.

Sun RPC

Sun RPC (Remote Procedure Call) je Sun a nyní RFC 1831 middleware pro komunikaci pomocí vzdáleného volání, původně navržený pro podporu NFS. Definuje jazyk pro popis rozhraní a formát kódování dat, dohromady označovaný jako XDR (External Data Representation). XDR je založený na C syntaxi, příklady lze zpravidla najít v /usr/include/rpcsvc/*.x.

```
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;
```

Každá RPC procedura je identifikována jednoznačným číslem služby, číslem verze a číslem procedury. Jejich právě nainstalovaný seznam lze vypsat.

```
> 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
```

Z XDR popisu se vygenerují stuby a skeletony zpravidla pomocí utility nazývané rpcgen, ze souboru foo tak vznikne foo.h s definicemi typů a funkcí, foo_svc.c jako server side stub, foo_clnt.c jako client side stub, foo_xdr.c jako marshalling code, s parametrem -a také foo_server.c jako sample server code a foo_client.c jako sample client code a Makefile.foo jako sample makefile.

Co se runtime týče, klient se dotáže portmapperu, default na localhost, na port, na kterém běží požadované číslo služby a číslo verze. Servery se u portmapperu registrují při spuštění, pohoda.

Examples

The examples were tested on Linux with the default RPC implementation. The default RPC implementation needs no special configuration to run the examples, but it will require running portmapper.

First, an example of an interface definition that contains a function for printing a string.

```
program PRINT
{
  version VERSION
  {
    void PRINT_STR (string STR) = 1;     /* function number 1  */
  } = 1;                                 /* version number 1   */
} = 666;                                 /* service number 666 */
```

Figure 7-1. Print Interface Example

To generate stubs, store the interface definition in a file named Print.x and use the following command:

```
rpcgen -a Print.x
```

The -a command line option tells the middleware to generate sample client and server in addition to stubs. The generated files are:

```
Makefile.Print
Print_client.c
Print_clnt.c
Print.h
Print_server.c
Print_svc.c
```

Next, an example modification of the sample client and server that prints a Hello World message using the function for printing a string.

```
void print_1 (char *host)
{
  ...
  char * print_str_1_arg = "Hello World !\n";
  ...
}
```

Figure 7-2. Sample Client Modification Example

```
#include <stdio.h>
...
void *print_str_1_svc (char **argp, struct svc_req *rqstp)
{
  static char *result = 0;
  printf ("%s", *argp);
  return (void *) &result;
}
```

Figure 7-3. Sample Server Modification Example

To compile and run the example, use the following commands, with the server and the client in different windows:

```
make -f Makefile.Print
./Print_server
./Print_client localhost
```

DCOM

DCOM (Distributed Component Object Model) is Microsoft middleware for communication using remote procedure calls. DCOM is integrated with a wide spectrum of technologies including COM (Component Object Model) and OLE (Object Linking And Embedding).

Interface Definition Language

The interface definition language of DCOM is called MIDL (Microsoft Interface Definition Language). The language is based on the interface definition language of DCE and syntactically resembles C++ where additional attributes are used to provide the necessary information for generating stubs.

```
[object, uuid(12345678-9ABC-DEF0-1234-56789ABCDEF0), ]
interface ISomething : IUnknown
{
  typedef unsigned char BUFFER [1234];
  HRESULT MethodOne ([in] short InOne,
                     [out] long *pOutOne,
                     [in, out] BUFFER *pBuffer);
};
```

Figure 7-4. MIDL Interface Definition Example

Interface And Component Attributes

The language defines interfaces as collections of data types and function prototypes. The description of an interface contains attributes that are valid for the entire interface. Some of the attributes are:

- auto_handle - automatically bind functions that have no explicit binding
- endpoint - the default protocol and address to be used by the server
- local - an indication of a local rather than a remote interface
- object - an indication of a COM rather than an RPC interface
- uuid - a universally unique identifier used to distinguish the interface
- version - a major and a minor version number of the interface, only for RPC interfaces

An RPC interface describes an arbitrary interface. A COM interface describes a component interface. Unlike an RPC interface, a COM interface has to inherit, directly or indirectly, from either the IUnknown or the IDispatch interface, and must have the uuid attribute.

The language can define components that group together multiple interfaces. The description of a component contains attributes that are valid for the entire component. Some of the attributes are:

- aggregatable - indicates that the component supports aggregation
- appobject - marks the component as a complete application
- control - marks the component as a user interface component
- hidden - marks the component as a hidden component

Type Attributes

- allocate - adjusts how memory for a type is allocated and freed
- context_handle - type contains server side context that is not accessed by client side
- decode - functions for deserialization are made accessible to the programmer
- encode - functions for serialization are made accessible to the programmer
- ignore - ignore the target of the associated pointer when marshalling
- represent_as - instructs certain wire type to be presented as certain local type, the programmer must supply conversion functions
- transmit_as - instructs certain local type to be transported as certain wire type, the programmer must supply conversion functions
- user_marshal - use marshalling functions supplied by the programmer for certain local type
- wire_marshal - use marshalling functions supplied by the programmer for certain wire type

When the server needs to keep a context between the calls that is not accessed by the client, it can use the context_handle attribute to define a context type. The value of the context type is kept on the server, only a reference to the value is transported over the network to the client.

Function Attributes

- async - generate client stub for asynchronous call with asynchronous call handle as first argument
- bindable - function is an accessor function for which change notification is provided
- broadcast - function call should be delivered to all available servers
- call_as - specifies simplified remote function to be used in place of complex local function
- callback - function exists on the client and can be called by the server within context of remote call
- idempotent - function will have the same effect if executed multiple times
- immediatebind - function is an accessor function for which changes should be made persistent immediately
- maybe - function does not need to be executed reliably
- message - call should be delivered as asynchronous message
- notify - generate server stub that calls notification procedure in case of marshalling failure
- propget - the function is a getter accessor function for a property
- propput - the function is a setter accessor function for a property
- usesgetlasterror - the function signals error code using SetLastError and GetLastError

For a local function with complex arguments, a remote function with simple arguments can be specified using the call_as attribute. The marshalling code is only generated for the simple function, the programmer must provide a pair of helper functions that convert the complex function call to the simple function call on the client side and vice versa on the server side.

Argument Attributes

- in - the argument is passed from client to server
- out - the argument is passed from server to client
- optional - the argument is optional
- readonly - the argument cannot be assigned to
- partial_ignore - when passing pointer from client to server, only transport information on wheter it is NULL
- defaultvalue - specifies a default value for an optional argument
- retval - the argument will hold the return value of the function

- ptr - the argument is a full pointer, which can be NULL and have aliases
- ref - the argument is a reference pointer, which cannot be NULL and cannot have aliases
- unique - the argument is a unique pointer, which can be NULL but cannot have aliases
- force_allocate - the argument will always be allocated dynamically

- byte_count - specifies a variable which holds size of referenced data
- first_is - specifies index of first array item to be transported
- last_is - specifies index of last array item to be transported
- length_is - specifies length of array to be transported
- switch_is - specifies discriminant of a union

- pipe - the argument represents a stream opened between the client and the server
- comm_status - the argument will hold failure code on communication error
- fault_status - the argument will hold failure code on server error

Pointers of three types are distinguished. The ref attribute denotes a pointer that cannot be NULL and cannot be aliased by having another pointer point to the same data. The value of the pointer does not change, only the data the pointer points to can be overwritten. The unique attribute denotes a pointer that can be NULL but cannot be aliased. The value of the pointer can change and the data the pointer points to can be overwritten. The ptr attribute denotes a pointer that can be NULL and can be aliased. The value of the pointer can change and the data the pointer points to can be overwritten.

Pointers annotated with partial_ignore are useful when a function expects a pointer argument that can be either NULL or point to uninitialized memory to be filled with data. When the client supplies a NULL pointer, the server receives a NULL pointer. When the client supplies a pointer to uninitialized memory, the server allocates zero filled memory to be filled with data before invoking the function and returns its content after invoking the function.

Arguments defined as pipe represent a stream opened between the client and the server within the context of the remote call. The client supplies a pull function for an input pipe, a push function for an output pipe, and an allocation function. The server can use the server stub to invoke the supplied functions on the client to pull or push data.

Components

IUnknown

```
interface IUnknown
{
  HRESULT QueryInterface (REFIID iid, void **ppvObject);
  ULONG AddRef (void);
  ULONG Release (void);
};
```

Figure 7-5. IUnknown Interface

IDispatch

```
interface IDispatch
{
  HRESULT GetTypeInfoCount (unsigned int FAR *pcTInfo);
  HRESULT GetTypeInfo (
    unsigned int iTInfo,
    LCID lcid,
    ITypeInfo FAR* FAR* ppTInfo);
  HRESULT GetIDsOfNames (
    REFIID riid,
    OLECHAR FAR* FAR* rgszNames,
    unsigned int cNames,
    LCID lcid,
    DISPID FAR* rgDispId);
  HRESULT Invoke (
    DISPID dispIdMember,
    REFIID riid,
    LCID lcid,
    WORD wFlags,
    DISPPARAMS FAR* pDispParams,
    VARIANT FAR* pVarResult,
    EXCEPINFO FAR* pExcepInfo,
    unsigned int FAR* puArgErr);
};
```

Figure 7-6. IDispatch Interface

Containment And Aggregation

Pri kombinovani objektu dve moznosti, nazyvane containment a aggregation. Containment je normalni, kdy vnejsi objekt ma instanci vnitrnich a deleguje jim volani. Aggregation je podobne dedicnosti, kdy vnejsi objekt vyvazi pointery na vnitrni. Tohle ma pochopitelny problem s IUnknown, protoze vyvezenim pointeru dostanu pristup ke QueryInterface agregovaneho objektu a musim nejak zarucit, ze mi bude umet najit i ostatni objekty agregace.

Reseni ala Microsoft je rici, ze agregovatelny objekt musi vedet o tom, ze je soucasti agregatu. Vsechna volani QueryInterface, AddRef a Release na jina rozhrani nez IUnknown se pak musi forwardovat na agregat.

Lifecycle

Sprava pameti je podobna jako v CORBE, tedy reference counting na objektech, in parametry musi alokovat a uvolnovat volajici, out parametry alokuje volany, inout parametry alokuje volajici a realokuje volany. Pamet musi byt alokovana pomoci COM alokatoru, pristupneho skrze interface IMalloc s metodami void *Alloc (ULONG), void *Realloc (void *, ULONG), void Free (void *), ULONG GetSize (void *), int DidAlloc (void *), void HeapMinimize (void).

Reference counting se lisi v local a remote pripadech. Protoze prenaseni kazdeho AddRef a Release na server by bylo pomale, zpristupni se k objektu jeste IRemUnknown s metodami RemAddRef a RemRelease. Klient pak vicenasobna volani na svem IUnknown prevadi na mene volani na IRemUnknown. Aby se resily situace, kdy klienti neudelaji RemRelease, je povinnosti kazdeho klienta jednou za cas pingnout server. Server pak uvolnuje objekty, na ktere nechodi pingy. Momentalne se pinga jednou za 2 minuty a uvolnuje po 3 chybejicich pingach, k dispozici je take architektura sdruzujici pingy pro objekty se stejnym OXID kvuli network loadu.

Objekty bezi bud v in process nebo v out of process serverech. Misto object adapteru ma COM apartments, rozdelene na single a multi threaded. In process servery registruji svuj threading model v registry pod svym class ID, out of process servery volaji CoInitializeEx a jako parametr uvedou ve flazich threading model. I v jednom procesu se pro komunikaci mezi apartments pouziva marshalling, kazdy apartment ma svou vlastni message loop.

Reference na objekty se ziskavaji opet podobne jako v CORBE, tedy bud volanim class factory s prislusnym class ID, nebo volanim jine metody, ktera vrati referenci. Class factory vyzaduje, aby server implementoval rozhrani IClassFactory s metodou HRESULT CreateInstance (iUnknown *pUnkOuter, REFIID iid, void **ppvObject), server poskytujici tuto factory se pak registruje v registry tak, aby ho Windows umely spustit z CoCreateInstance a pri vytvoreni factory zavola CoRegisterClassObject. Factory muze bezet i na vzdalenem stroji, to se bud zada v registry jako soucast registrace tridy, nebo uvede jako parametr CoCreateInstanceEx.

Obdobou klasickych externalized object references jsou monikers. Moniker ma interface IMoniker s dulezitou metodou BindToObject na ziskani pointeru na referovany objekt. Monikeru je vice druhu (File Moniker na referovani objektu, ktere jsou ulozene v souboru, Pointer Moniker na referovani objektu, ktere jsou pouze v pameti, URL Moniker na referovani objektu dostupnych pres URL, atd.). Vytvorene objekty se registruji v running object table pomoci volani Register rozhrani IRunningObjectTable, aby je moniker nevytvarel pri opakovanem volani BindToObject vicekrat.

Monikery je mozne take externalizovat do human readable form a zase prevest zpatky. Pouziti je celkem zrejme, vysledek vypada jako URL.

Pri marshalling a unmarshalling se object reference prenaseji jako ctverice IID (interface ID), OXID (object exporter ID), OID (object identifier ID) a IPID (interface pointer ID). Z IID se pozna, jakou proxy vytvorit, OXID identifikuje server, OID a IPID nevim (nejspis konkretni instanci a konkretni interface).

Extras

Pro persistenci jsou k dispozici interfaces IPersistStream, IPersistFile a dalsi, vesmes s metodami Load, Save, IsDirty. Nic dvakrat zajimaveho.

K dispozici jsou take transakce, z nich v podstate vznikly EJB. Objekty maji transakcni atributy, COM+MTS se stara o dvoufazovy commit. Tusim od COM+ ve W2K mohou objekty registrovat vlastni resources.

Ve spojeni s MSMQ se podporuji queued components. Pozadavky na operace jsou podobne jako na oneway v CORBE. Vysledek se pak chova jako CORBA OTS nad CORBA messaging, tedy v transakci se zpravy zapisi do fronty, po commit se forwarduji, jejich vykonavani je nezavisla transakce.

Rehearsal

The notable features of DCOM whose purpose and design you should understand include:

- The interface definition language with its use of annotated C types.
- The mechanism for distinguishing interfaces based on UUID.
- The mechanism for opening a stream between the client and the server based on pipes.
- The model of composable components with multiple interfaces.
- The approach to lifecycle management with factories and reference counting.

References

1. Robert Antonio: Component Object Model. http://www.antonio.cz/com/index.html

Questions

1. Microsoft DCOM rozlišuje in process a out of process servery podle toho, zda běží v procesu klienta nebo ve vlastním procesu. Vysvětlete, proč Microsoft DCOM tyto druhy procesů nabízí.

   Hint: Consider how the options differ in their impact of the communication efficiency, state sharing, stability. Can you come up with examples of servers that fit one of the options well ?

2. Microsoft DCOM uses a combination of keepalive pinging and reference counting to garbage collect unused objects. Evaluate this approach from the scalability point of view.
3. The IUnknown interface in Microsoft DCOM has the following methods:

   ```
   interface IUnknown
   {
     HRESULT QueryInterface (REFIID iid, void **ppvObject);
     ULONG AddRef (void);
     ULONG Release (void);
   };
   ```

   Explain the semantics of the methods.

JAXB

JAXB (Java Architecture for XML Binding) is a standard for manipulating XML data. The data is made accessible as programming language objects with unique types that correspond to structural constraints imposed on the XML data by the associated schema.

The standard defines a mapping from XML Schema to Java and a mapping from Java to XML Schema. Both mappings map common constructs naturally, for example XML namespaces correspond to Java packages, XML elements to Java value classes with

XML attributes and children corresponding to Java properties, XML integer types to Java integer types. Custom classes or adaptors handle constructs that do not have a natural mapping. Mappings can be customized by XML declarations or Java annotations.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="purchase" type="OrderType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="OrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="Address"/>
      <xsd:element name="billTo" type="Address"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="date"type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="Some Country"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            ...
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  ...

  <!-- Example adjusted from JAXB 2.2 Specification. -->
```

Figure 7-7. Mapping Example XML Schema Input

```
public class OrderType {
  Address getShipTo () {...}
  void setShipTo (Address) {...}
  Address getBillTo () {...}
  void setBillTo (Address) {...}

  String getComment () {...}
  void setComment (String) {...}

  Items getItems () {...}
  void setItems (Items) {...}

  XMLGregorianCalendar getOrderDate () {...}
  void setOrderDate (XMLGregorianCalendar) {...}
};
```

```
public class Address {
  String getName () {...}
  void setName (String) {...}
  String getStreet () {...}
  void setStreet (String) {...}
  String getCity () {...}
  void setCity (String) {...}
  int getZip () {...}
  void setZip (int) {...}

  static final String COUNTRY=”Some Country”;
};

public class Items {
  public class ItemType {...}

  List<Items.ItemType> getItem () {...}
}

// Example adjusted from JAXB 2.2 Specification.
```

Figure 7-8. Mapping Example Java Output

The standard also defines the interface to read and write XML data. Besides reading files, the interface can also access data through SAX or DOM. The interface can unmarshal only a part of a document, and, after that part of the document is modified through the binding, marshal the part back into the document.

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.

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.

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.

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.

```
interface BundleActivator {
  void start (BundleContext context) throws Exception;
  void stop (BundleContext context) throws Exception;
}

interface BundleContext {
```

```
  // Access to framework properties
  String getProperty (String key);

  // Access to objects representing bundles
  Bundle getBundle ();
  Bundle getBundle (long id);
  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);

  ...
}
```

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.

```
interface BundleContext {

  ...

  // Support for service management
  ServiceRegistration registerService (String [] clazzes, Object service, Dictionary pr
  ServiceRegistration registerService (String clazz, Object service, Dictionary propert

  Filter createFilter (String filter) throws InvalidSyntaxException;
  ServiceReference [] getServiceReferences (String clazz, String filter) throws Invalic
  ServiceReference [] getAllServiceReferences (String clazz, String filter) throws Inva

  ServiceReference getServiceReference (String clazz);
  Object getService (ServiceReference reference);
  boolean ungetService (ServiceReference reference);

  // Support for service lifecycle notifications
  void addServiceListener (ServiceListener listener, String filter) throws InvalidSynta
  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.

Chord

Chord is a middleware that provides basic support for distributed hashing. Chord assigns random unique identifiers to nodes. Given a key, Chord can deliver a message to a node whose identifier is the closest higher identifier than the key. For N nodes, it takes O(log N) steps to deliver the message, provided each node maintains a routing table of size O(log N).

Routing Protocol

Logical ring. Routing table with fingers, successor, predecessor. Periodic stabilization, asking successor for predecessor, routing to calculated finger position.

Application Interface

The interface of Chord exposes the Chord protocol by providing a callback function that notifies about nodes joining and leaving the neighborhood of a given node, and a function that calculates the successor when routing to a given node.

```
void event_register ((fn)(int));
ID next_hop (ID j, ID k);
```

On top of this interface, another interface is provided that stores and retrieves an item from a distributed hash table.

```
err_t insert (void *key, void *value);
void *lookup (void *key);
```

Rehearsal

References

1. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications.

Questions

1. The Chord middleware uses routing along a logical ring to deliver a message to a node with the address that is nearest to the message key. Describe the routing tables kept by the individual nodes and show the complexity of sending a message on a brief description of the routing algorithm.

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.

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++.

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.

```
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.

```
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.

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

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

```
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.

Constructed Data Types

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

```
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.

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

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.

```
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.

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

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

```
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.

```
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.

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

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

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

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.

```
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;
}
```

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.

```
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);
}
```

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.

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.

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.

```
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);
  }
}
```

Figure 7-9. Holder Class Example

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.

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.

```
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]);
    }

    ...
}
```

Figure 7-10. Var Class Example

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

```
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);
}
```

Figure 7-11. Var Class Usage

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.

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.

```
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;
}
```

Figure 7-12. Any Class Example

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 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);
```

Figure 7-13. Any Class Insertion

```
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)) ...;
```

Figure 7-14. Any Class Extraction

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.

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.

```
class Exception
{
  public:

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

Figure 7-15. Exception Class Example

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.

```
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

    ...
}
```

Figure 7-16. Union Class Example

```
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 ();
```

Figure 7-17. Union Class Usage

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.

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.

```
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) {...};
}
```

Figure 7-18. Enum Class Example

```
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: ...;
}
```

Figure 7-19. Enum Class Usage

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.

```
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;

    ...
}
```

Figure 7-20. Sequence Class Example

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.

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.

```
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);
...
```

Figure 7-21. Fixed Class Example

Java

The mapping simply uses the `BigDecimal` class.

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.

```
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 ();

    ...
}
```

Figure 7-22. Proxy Interface Class Example

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

```
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); }

    ...
}
```

Figure 7-23. Proxy Var Class Example

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.

```
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) {...}

  ...
```

```
}
```

Figure 7-24. Proxy Class Example

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.

```
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 ();

    ...
}
```

Figure 7-25. Servant Base Class

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.

```
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 (...); }
}
```

Figure 7-26. Servant Class Example

C++11

The servant mapping starts with a 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 ();
}
```

Figure 7-27. Servant Base Class

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.

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

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

    ...
}

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

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

Figure 7-28. Servant Class Example

Java

The servant mapping starts with a servant base class.

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

Figure 7-29. Servant Base Class

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.

```
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 (...); }
}
```

Figure 7-30. Servant Class Example

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.

```
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;

    ...
}
```

Figure 7-31. Value Mapping Example

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.

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.

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*.

```
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 ServantRetentionPolicyValu
  RequestProcessingPolicy create_request_processing_policy (in RequestProcessingPolicyV

  ...
};

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);
};
```

Figure 7-32. Object Adapter Configuration

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.

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.

```
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, WrongAdapte
```

Figure 7-33. Object Activation

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

Figure 7-34. Current Object Interface

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.

```
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};
};
```

Figure 7-35. Servant Activator 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);
};
```

Figure 7-36. Servant Locator Interface

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

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

Figure 7-37. Request Forward Exception

Network Protocol

Základem je GIOP, který definuje Common Data Representation (CDR), Message Formats a Transport Assumptions. Nadstavbou je IIOP.

Common Data Representation umí variable byte ordering, má aligned basic types. Zajímavé jsou type codes, které umí rekurzivně popsat typ, encapsulations, které umí předat stream bajtů s vlastním byte ordering a používají se na type codes a tedy na Any, object references, které umí profiles.

Message Formats jsou dva základní, request a reply, dohromady s dalšími je jich sedm. Všechny zprávy mají header, obsahuje magic, version, flags, message type,

message size. Request obsahuje service context, request ID, flags, object key, operation name, arguments. Reply obsahuje request ID, status, service context, arguments. Locate Request a Locate Reply. Request ID při fragmentaci.

Messaging

Quality of service for Rebinding, Synchronization, Priority, Timeout, Routing, Ordering. Callback and polling. Callback void sendc_OP (ReplyHandler, in a inout argumenty), reply handler AMI_IFHandler s operací void OP (return value, inout a out argumenty) a operací void OP_excep (holder) pro výjimky. Polling Poller sendp_OP (in a inout argumenty), poller AMI_IFPoller s operací void OP (timeout, return value, inout a out argumenty).

Components

To be done.

```
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;
  };

  ...
};
```

Real Time

Priorities and priority mapping. Priority propagation models. Thread pools and lanes. Banded connections. Private connections. Timeouts.

Rehearsal

The notable features of CORBA whose purpose and design you should understand include:

- The interface definition language with its spectrum of basic types and constructed types.
- The mappings of the interface definition language into various implementation languages with various approaches to bridging incompatibilities.
- The object adapter with its emphasis on the configurability of the threading model and the association between the object references and servant instances.
- The network protocol with its support for efficient interoperability between hegerogeneous platforms.
- The naming and trading services.

References

1. DOC Group: TAO Implementation Repository. http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/implrepo/paper.html

Questions

1. The CORBA middleware relies on a specialized interface definition language to describe interfaces of remotely accessible objects. Explain why this choice was made rather than relying on the interface definitions in an implementation language.
2. Even though CORBA IDL interfaces are mapped into classes both in C++ and in Java, the CORBA IDL interface attributes are not mapped into class attributes in C++ nor in Java. Explain why and outline how the attributes are mapped.
3. When mapping the CORBA IDL integer types into C++, the C++ integer types cannot be used directly because C++ does not define the precision of the integer types in as much detail as CORBA IDL. Explain how this problem is solved.
4. When mapping the CORBA IDL integer types into Java, not all the integer types can be mapped directly because Java does not provide all the precisions of the integer types available in CORBA IDL. Explain how this problem is solved.
5. The CORBA IDL sequence type, which describes a variable length vector, has no direct counterpart in the basic C++ types. Explain how and why the type is mapped into C++.
6. The CORBA IDL union type, which describes a container that can hold any single item of a choice of items and preserves information on which item is held at a given time, has no direct counterpart in the basic C++ nor Java types. Choose either C++ or Java and explain how and why CORBA IDL union is mapped into that language.

7. Nakreslete strukturu CORBA aplikace. Vyznačte, kde jsou umístěné nebo se používají IIOP, POA, proxy, servant, skeleton a stub a vysvětlete funkci těchto prvků.
8. Při podpoře paralelního zpracování více požadavků na serveru se může použít model thread pool, ve kterém má server množinu vláken připravených k obsluze požadavků. Vysvětlete přednosti tohoto modelu a doplňte zde uvedený popis modelu o podrobnosti, o které se tyto přednosti opírají.
9. Portable Object Adapter ve standardu CORBA dovoluje definovat default servant, kterému jsou doručeny všechny požadavky, pro které nebyl nalezen jiný servant. Vysvětlete, v jakých situacích je vhodné default servant použít.
10. Portable Object Adapter ve standardu CORBA dovoluje definovat servant activator, který je požádán o vytvoření servantu v situaci, kdy pro příchozí požadavek nebyl žádný servant nalezen. Vysvětlete, v jakých situacích je vhodné servant activator použít.
11. Portable Object Adapter ve standardu CORBA dovoluje nastavit pomocí servant retention policy zda se mají v tabulce active object map evidovat všechny aktivní servanty. Vysvětlete, v jakých situacích je vhodné upustit od evidence aktivních servantů v active object map.
12. Protokol GIOP ve standardu CORBA umožňuje pomocí zprávy LOCATION FORWARD informovat klienta o tom, že server se nachází jinde než se klient domníval. Vysvětlete, jak je tento mechanismus možné použít k implementaci persistentních objektů, jejichž reference zůstávají v platnosti i při ukončení serveru.
13. Standard CORBA definuje službu Naming sloužící k registraci a vyhledávání serverů. Základní operace této služby jsou bind a resolve:

    ```
    void bind (in Name n, in Object obj) raises (AlreadyBound ...);
    Object resolve (in Name n) raises (NotFound ...);
    ```

    Popište sémantiku těchto operací.
14. Standard CORBA definuje službu Trading sloužící k registraci a vyhledávání serverů. Základní operace této služby jsou export a query:

    ```
    OfferId export (
      in Object reference,
      in ServiceTypeName type,
      in PropertySeq properties);
    void query (
      in ServiceTypeName type,
      in Constraint constr,
      in Preference pref,
      ...
      out OfferSeq offers,
      out OfferIterator offer_itr,
      ...);
    ```

    Popište sémantiku těchto operací.
15. Standard CORBA nabízí možnost neblokujícího volání pomocí callbacku. Následující fragment CORBA IDL uvádí příklad blokující operace a k ní vygenerované rozhraní pro neblokující volání pomocí callbacku:

    ```
    // Příklad blokující operace
    void example_operation (in string anInArgument, out double anOutArgument);

    // Rozhraní pro neblokující volání
    void sendc_example_operation (
      in AMI_ExampleHandler ami_handler,
      in string anInArgument);

    interface AMI_ExampleHandler
    {
    ```

```
  void example_operation (in double anOutArgument);
}
```

Vysvětlete na tomto příkladu fungování neblokujícího volání pomocí callbacku.

16. Standard CORBA nabízí možnost neblokujícího volání pomocí pollingu. Následující fragment CORBA IDL uvádí příklad blokující operace a k ní vygenerované rozhraní pro neblokující volání pomocí pollingu:

```
// Příklad blokující operace
void example_operation (in string anInArgument, out double anOutArgument);

// Rozhraní pro neblokující volání
AMI_ExamplePoller sendp_example_operation (in string anInArgument);

valuetype AMI_StockManagerPoller
{
  void example_operation (in unsigned long timeout, out double anOutArgument);
}
```

Vysvětlete na tomto příkladu fungování neblokujícího volání pomocí pollingu.

JAX-RS

JAX-RS (Java API for REST Web Services) is a standard for implementing REST based services in Java. The major part of the standard is a mapping of REST service resources to Java classes, which is accompanied by a definition of implementation packaging.

In the mapping, annotations are used to denote classes that represent resources. Further annotations specify path templates for those resources. Method annotations specify handlers for requests. Resource classes are instantiated for individual requests.

The mapping also defines provider classes, responsible for custom marshalling and custom context. Standard provider classes handle marshalling of common content such as strings, maps, XML.

```
@Path ("example")
public class ExampleResource {
  @GET
  @Path ("{key}")
  @Produces ("application/value+xml")
  public KeyValueType getKeyValue (@PathParam ("key") String key) {...}

  @DELETE
  @Path ("{key}")
  public void deleteKeyValue (@PathParam ("key") String key) {...}
}
```

Figure 7-38. Service Example

JAX-WS

JAX-WS (Java API for XML-Based Web Services) is a standard for implementing WSDL interfaces in Java. Major parts of the standard are a mapping from WSDL to Java, a mapping from Java to WSDL, and the associated interfaces for publishing services and creating proxies.

In the mappings, WSDL port types correspond to Java interfaces and WSDL operations to Java methods, types are mapped using the JAXB standard.

```
package com.example;

@WebService
public interface StockQuoteProvider {
  float getPrice (String tickerSymbol)
    throws TickerException;
}

// Example adjusted from JAX-WS 2.2 Specification.
```

Figure 7-39. Mapping Example Java Input

```
<types>
  <xsd:schema targetNamespace="...">
    <xsd:element name="getPrice" type="tns:getPriceType"/>
    <xsd:element name="getPriceResponse" type="tns:getPriceResponseType"/>
    <xsd:element name="TickerException" type="tns:TickerExceptionType"/>
    ...
  </xsd:schema>
</types>

<message name="getPrice">
  <part name="getPrice" element="tns:getPrice"/>
</message>
<message name="getPriceResponse">
  <part name="getPriceResponse" element="tns:getPriceResponse"/>
</message>
<message name="TickerException">
  <part name="TickerException" element="tns:TickerException"/>
</message>

<portType name="StockQuoteProvider">
  <operation name="getPrice">
    <input message="tns:getPrice" wsam:action="..."/>
    <output message="tns:getPriceResponse wsam:action="..."/>
    <fault message="tns:TickerException wsam:action="..."/>
  </operation>
</portType>

<!-- Example adjusted from JAX-WS 2.2 Specification. -->
```

Figure 7-40. Mapping Example WSDL Output (Document Style)

```
<types>
  <xsd:schema targetNamespace="...">
    <xsd:element name="TickerException" type="tns:TickerExceptionType"/>
    ...
  </xsd:schema>
</types>

<message name="getPrice">
  <part name="tickerSymbol" type="xsd:string"/>
</message>
<message name="getPriceResponse">
  <part name="return" type="xsd:float"/>
</message>
<message name="TickerException">
  <part name="TickerException" element="tns:TickerException"/>
</message>
```

```
<portType name="StockQuoteProvider">
  <operation name="getPrice">
    <input message="tns:getPrice"/>
    <output message="tns:getPriceResponse"/>
    <fault message="tns:TickerException"/>
  </operation>
</portType>

<!-- Example adjusted from JAX-WS 2.2 Specification. -->
```

Figure 7-41. Mapping Example WSDL Output (RPC Style)

Django

Django is a website implementation framework that provides separation of concerns especially for persistence and templating.

Hadoop

Hadoop is a distributed storage system and a distributed processing system that supports map reduce operation.

References

1. The Hadoop Website. http://hadoop.apache.org

Pastry

Pastry is a middleware that provides basic support for distributed hashing. Pastry assigns random unique identifiers to nodes. Given a key, Pastry can deliver a message to a node whose identifier is the closest higher identifier than the key. For N nodes, it takes O(log N) steps to deliver the message, provided each node maintains a routing table of size O(log N). Pastry configuration allows the base of the logarithm to be set to an arbitrary number.

Routing Protocol

Identifiers as sequences of digits with configurable base. Routing table, lists of nodes with matching prefixes, list of leaves, list of neighbors. Repairs within leaf set, repairs within routing table.

Application Interface

```
interface Node
{
  Endpoint registerApplication (Application application, java.lang.String instance);
  ...
}

interface Endpoint
{
  Id getId ();
```

```
  // Get a list of nodes that can route towards a node
  NodeHandleSet localLookup (Id id, int num, boolean safe);
  // Get the list of neighbor nodes
  NodeHandleSet neighborSet (int num);

  // Get the range of keys we are responsible for
  IdRange range (NodeHandle handle, int rank, Id lkey);
  // Get a list of nodes that can store a replica
  NodeHandleSet replicaSet (Id id, int maxRank);

  // Send a message to a node
  void route (Id id, Message message, NodeHandle hint);
  ...
}
```

Figure 7-42. Pastry Endpoint API

```
interface Application
{
  void deliver (Id id, Message message);
  boolean forward (RouteMessage message);
  void update (NodeHandle handle, boolean joined);
}
```

Figure 7-43. Pastry Application API

References

1. Antony Rowstron, Peter Druschel: Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems.

0MQ

0MQ is a middleware for versatile messaging communication.

Chimera

Application Interface

```
// Initialization for listening on given port
ChimeraState *chimera_init (int port);

// Join a network using a known bootstrap host
void chimera_join (ChimeraState *state, ChimeraHost *bootstrap);

// Manage objects representing hosts
ChimeraHost *host_get (ChimeraState *state, char *hostname, int port);
void host_release (ChimeraState *state, ChimeraHost *host);

// Upcall for host membership change notification
typedef void (*chimera_update_upcall_t) (Key *key, ChimeraHost *host, int joined);
void chimera_update (chimera_update_upcall_t func);

// Register a message type
```

```
void chimera_register (ChimeraState *state, int type, int ack);

// Create a key from a string
void key_makehash (void *log, Key *hashed, char *s);

// Send a message to host responsible for key
void chimera_send (ChimeraState *state, Key key, int type, int len, char *data);

// Upcall for message forward event
typedef void (*chimera_forward_upcall_t) (Key **key, Message **msg, ChimeraHost **host)
void chimera_forward (chimera_forward_upcall_t func);

// Upcall for message delivery event
typedef void (*chimera_deliver_upcall_t) (Key *key, Message *msg);
void chimera_deliver (chimera_deliver_upcall_t func);
```

References

1. Chimera: A Structured Peer to Peer Overlay. http://current.cs.ucsb.edu/projects/chimera

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.

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 module uses a mesh of TCP connections to deliver both multicast and unicast messages. The TUNNEL module can tunnel other transport to a 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 and S3_PING keep track of members in various places ranging from shared file systems and shared database tables to cloud storage services.

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. The MERGE3 module has all members periodically multicast membership information hash, inconsistent membership information is retrieved and merged upon discovery.

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 module uses multicast heartbeat among all members in a group.

The FD_SOCK module uses a TCP socket ring, socket close indicates suspect. The VERIFY_SUSPECT module provides additional verification of suspect members.

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

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

Channels

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

```
public class JChannel ...
{
  // Initialization can accept either options or configuration file
  public JChannel (String props) throws Exception;

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

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

  public void send (Message msg) throws Exception;
  public void send (Address dst, byte[] buf) throws Exception;
  public void send (Address dst, Serializable obj) throws Exception;
  public void receive (Message msg);

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

  ...
}
```

Figure 7-44. Channel Class

```
public interface MessageListener
{
  void receive (Message msg);

  // Group members can share state
  void getState (OutputStream output) throws Exception;
  void setState(InputStream input) throws Exception;
}

public interface MembershipListener
{
  // Notification about membership view change
  public void viewAccepted (View view);

  // Indication of suspect member before actual removal
  public void suspect (Object suspected);

  // Notifies about temporary suspension during view update
  public void block ();
  public void unblock ();
}

public interface Receiver extends MessageListener, MembershipListener;
```

Figure 7-45. Receiver Interface

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.

Bigtable

Bigtable is a distributed storage system for tables with versioned rows.

References

1. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber: Bigtable: A Distributed Storage System for Structured Data.

Memcached

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

References

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

ProActive

ProActive is a middleware for distributed applications that provides support for patterns based on active objects.

Rehearsal

References

1. OASIS Team: ProActive Parallel Suite. http://proactive.ow2.org

Questions

1. The ProActive middleware uses *futures* to pass results of method invocations on active objects. Explain why.

JavaSpaces

JavaSpaces is a middleware for distributed shared memory communication in Java.

Rehearsal

Questions

1. The basic operations of JavaSpace are `write`, `read` and `take`:

```
interface JavaSpace
{
  Lease write (Entry e, Transaction tx, long lease);
  Entry read (Entry template, Transaction tx, long timeout);
  Entry take (Entry template, Transaction tx, long timeout);
  ...
}
```

   Describe what these operations do.