Improvements in CDR
6.1 Introduction

Networking formats also influence the efficiency of presentation code. Different encoding mechanisms like Network Data Representation (NDR), Common Data Representation (CDR) and External Data Representation (XDR) facilitate intercomponent communication efficiently and transparently. NDR is used by Microsoft, while CDR and XDR is used by OMG and Sun's rpcgen respectively.

Sun's rpcgen is an IDL compiler that converts interface specifications into stub code. The stub code marshals data into XDR format. XDR is a standard description and encoding of data. It is used to transfer data between different computer architectures like Sun workstations, VAX, IBM-PC and Cray. It fits into the ISO presentation layer. The hardware device should encode the bytes into various media in such a way that other hardware devices can decode the bytes without loss of meaning. Ethernet standard encodes the bytes in "little-endian" format. The number of bytes that contain the encoded data is in multiples of 4. If the data bytes are not in multiples of 4, then it is padded with zeros. The following are the disadvantages of XDR:

- There is no representation for bit fields and bit maps. It is based on bytes.
- There is no BCD representation
- Since there is only one byte ordering, it cannot be used on certain machines.
- Some machines like Cray do not use 4-byte alignment of data.
- XDR uses implicit data types. Even though this avoids redundancy, only one representation of the data is possible.
Microsoft uses NDR to encode data into a common network representation. MSIDL compiler generates stub code, which takes care of marshalling data into NDR format. It maps MSIDL data types into octet streams. Each primitive type in NDR has various data representations. For example, the character type can be represented in EBCDIC/ASCII format. The byte ordering can be little/big-endian format. NDR has a format label, which occupies 4 bytes. It gives the representation of integer, character and floating-point types used. So NDR supports multichannel approach to data conversion. It has a fixed set of alternative representations for data types. It can represent floating point suitable for IEEE, VAX, Cray and IBM machines. Integer and float can be big-endian and little-endian format. The character representation can be in ASCII/EBCDIC format. NDR label identifies the type of the representation for character, integer and float types. Like XDR, the data bytes are aligned in multiples of 4. So for primitive types it is padded with zero to achieve alignment. The various datatypes in CDR, XDR and NDR format are tabulated in table 6.1.

Abstract Syntax Notation ver 1 (ASN.1), a data definition language, uses OSI Basic Encoding Rules (BER) canonical format to transfer information between heterogeneous systems. But it uses annotations and tags extensively to describe data. Further, encoding/decoding takes place even when the client and server are collocated.

CORBA uses CDR mechanism to encode data. It is a neutral, bicanonical on-wire representation of data. The features of CDR include variable byte ordering, data alignment at word boundaries and complete IDL mapping. General Inter-operable Protocol (GIOP) is used as the basic communication protocol in CORBA. Commercial ORBs like Visibroker, MICO use Internet Interoperable Protocol (IIOP) for communication between distributed objects. IIOP implements GIOP specifications over TCP/IP. The GIOP is intended to provide a protocol that fits and incorporates the features of application, presentation and session layers in Open Systems Interconnection.
(OSI) model. It aims at providing interoperability between different ORBs. It has three core elements – Message formats, CDR and complete IDL mapping.

Table 6.1: Data types in NDR, XDR and CDR formats

<table>
<thead>
<tr>
<th>S.No</th>
<th>XDR (bytes)</th>
<th>NDR (bytes)</th>
<th>CDR (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>boolean (1)</td>
<td>boolean (1)</td>
<td>boolean (1)</td>
<td>An 8-bit value</td>
</tr>
<tr>
<td>2</td>
<td>char(1)</td>
<td>char(1)</td>
<td>char(1)</td>
<td>An 8-bit value</td>
</tr>
<tr>
<td>3</td>
<td>octet (1)</td>
<td>octet (1)</td>
<td>octet (1)</td>
<td>An 8-bit value with no marshalling</td>
</tr>
<tr>
<td></td>
<td>small (1)</td>
<td>-</td>
<td>-</td>
<td>A 8-bit integer [-2, 2-1]</td>
</tr>
<tr>
<td>4</td>
<td>short (2)</td>
<td>short (2)</td>
<td>short (2)</td>
<td>A 16-bit integer [-2, 2-1]</td>
</tr>
<tr>
<td></td>
<td>unsigned short (2)</td>
<td>unsigned short (2)</td>
<td>unsigned short (2)</td>
<td>A 16-bit integer [0, 216-1]</td>
</tr>
<tr>
<td>5</td>
<td>int (4)</td>
<td>unsigned long (4)</td>
<td>long (4)</td>
<td>A 32-bit integer [-2, 2-1]</td>
</tr>
<tr>
<td></td>
<td>unsigned int (4)</td>
<td>unsigned long (4)</td>
<td>unsigned long (4)</td>
<td>A 32-bit integer [0, 232-1]</td>
</tr>
<tr>
<td>6</td>
<td>hyper int (8)</td>
<td>hyper int (8)</td>
<td>long long (8)</td>
<td>A 64-bit integer [-2, 2-1]</td>
</tr>
<tr>
<td>7</td>
<td>unsigned hyper int (8)</td>
<td>unsigned long (8)</td>
<td>unsigned long (8)</td>
<td>A 64-bit integer [0, 264-1]</td>
</tr>
<tr>
<td>8</td>
<td>float (4)</td>
<td>float (4)</td>
<td>float (4)</td>
<td>A 32-bit value</td>
</tr>
<tr>
<td>9</td>
<td>double (8)</td>
<td>double (8)</td>
<td>double (8)</td>
<td>A 64-bit value</td>
</tr>
<tr>
<td>10</td>
<td>long double</td>
<td>-</td>
<td>-</td>
<td>A 128-bit value conforming to IEEE double-precision floating-point standard.</td>
</tr>
<tr>
<td>11</td>
<td>wchar (1,2,4)</td>
<td>Wchar (1,2,4)</td>
<td>-</td>
<td>An 8-bit, 16-bit or 32-bit value that represents an international character data.</td>
</tr>
<tr>
<td>12</td>
<td>string (multiple of 4 bytes)</td>
<td>string (varying/conformant)</td>
<td>String /wstring</td>
<td>A string of characters</td>
</tr>
<tr>
<td>13</td>
<td>array size is in multiple of 4</td>
<td>unidimensional/multidimensional/conformant arrays</td>
<td>array size is multiple of 4 and depends on the type of the array element</td>
<td>Fixed length arrays</td>
</tr>
<tr>
<td>14</td>
<td>struct: each component size is multiple of 4</td>
<td>struct: alignment depends on the size of the largest component</td>
<td>struct: elements of struct undergoes alignment</td>
<td>Structure</td>
</tr>
<tr>
<td>15</td>
<td>union size = discriminant size of 4 bytes and the size of the largest case.</td>
<td>union size = discriminant size of 4 bytes and the size of the largest case.</td>
<td>union size = discriminant size of 4 bytes and the size of the selected case.</td>
<td>Union</td>
</tr>
<tr>
<td>16</td>
<td>void</td>
<td>-</td>
<td>-</td>
<td>Zero byte</td>
</tr>
<tr>
<td>17</td>
<td>const</td>
<td>-</td>
<td>-</td>
<td>Symbolic name</td>
</tr>
<tr>
<td>18</td>
<td>enum (4)</td>
<td>enum (2)</td>
<td>enum (2)</td>
<td>Enumerated data type</td>
</tr>
<tr>
<td>19</td>
<td>Opaque</td>
<td>-</td>
<td>-</td>
<td>Multiple of 4 bytes</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>Pipes</td>
<td>-</td>
<td>Ordered chunks</td>
</tr>
</tbody>
</table>
a) **Message Formats:** Each message has a GIOP header and its byte ordering. GIOP supports eight messages as shown in table 6.2.

i) The following messages originate from the client

- Request message to encode the object invocation from the client to the server
- LocateRequest message to obtain some information from the server like the validity of the object reference (OR) and state of the server.
- CancelRequest is sent by the client to the server to terminate a prior request.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Message Type</th>
<th>Source</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Request Message</td>
<td>Client</td>
<td>To encode the object invocation</td>
</tr>
<tr>
<td>2</td>
<td>Locate Request</td>
<td>Client</td>
<td>To request information like OR from the server</td>
</tr>
<tr>
<td>3</td>
<td>Cancel Request</td>
<td>Client</td>
<td>To terminate prior requests</td>
</tr>
<tr>
<td>4</td>
<td>Response</td>
<td>Server</td>
<td>Reply returned</td>
</tr>
<tr>
<td>5</td>
<td>Locate reply</td>
<td>Server</td>
<td>Response to locate request</td>
</tr>
<tr>
<td>6</td>
<td>Close Connection</td>
<td>Server</td>
<td>No response is returned from server</td>
</tr>
<tr>
<td>7</td>
<td>Error Message</td>
<td>Client/Server</td>
<td>Client/Server detects error</td>
</tr>
<tr>
<td>8</td>
<td>Fragment Message</td>
<td>Client/Server</td>
<td>To break request/reply into blocks</td>
</tr>
</tbody>
</table>

ii) The following messages originate from the server

- Response message is sent from the server to the client if reply is expected by it.
- LocateReply message responds to LocateRequest message.
- CloseConnection message informs the client that no response will be returned from the server.

iii) The messages supported by both clients and servers include the following:

- The Error Message is sent when a client or a server detects an error.
The Fragment message is sent when a request or reply is broken into blocks that are sent independently.

b) The Common Data Representation (CDR): The data representations in different machines vary, since the machines have their own word byte ordering. So the data must undergo some transformation process before transmission. This ensures that both the transmitting and the receiving parties understand the data. CORBA uses CDR for this purpose. It is a data-formatting rule that allows variable byte ordering and support for OMG's IDL. CDR has the following features:

i) **Variable byte ordering:** The sender sends the data in its own byte ordering. The receiver swaps this ordering to have the data in the correct order for the receiver. Thus the client need not know the details of the server machine architecture.

ii) **Data alignment:** In CDR all data is aligned at the word boundaries. CDR defines alignment policies for primitive types in big and little-endian format as shown in table 6.3. All complex types are broken into its constituent simple types. The access to a data type that is larger than a byte must be aligned using, \(a \mod n = 0\), where 'a' is the byte address of the data type and 'n' is the size of the data type in bytes.

<table>
<thead>
<tr>
<th>Alignment</th>
<th>IDL data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char, octet, Boolean</td>
</tr>
<tr>
<td>2</td>
<td>short, unsigned short</td>
</tr>
<tr>
<td>4</td>
<td>long, unsigned long, float, enumerated types</td>
</tr>
<tr>
<td>8</td>
<td>long long, unsigned long long, double, long double</td>
</tr>
<tr>
<td>1, 2 or 4</td>
<td>wchar (alignment depends on code set)</td>
</tr>
</tbody>
</table>

CDR specifies the layout for little and big-endian formats for primitive types. The layout of complex types is based on the primitive types that comprise the complex data type. Complex data types include structures, unions and arrays.
- **Structure:** The encoding is based on the primitive types that comprise the structure. It is encoded in the same order as declared in the IDL. The elements in the structure must undergo alignment based on its primitive elements. The structure itself is aligned based on the size of its largest element.

- **Union:** The encoding of a union starts with the discriminant tag of the type specified in the union declaration. It is followed by the encoding of the selected number.

- **Arrays:** An array encodes its elements in sequence. The types of the elements in an array determine its encoding. No encoding of the array lengths occur since they are given in the IDL.

iii) **Complete IDL mapping:** All data types defined in the OMG IDL can be represented in CDR format. Primitive types are encoded in multiples of octets. Complex types are built from primitive types. Client data is transmitted as an octet stream of arbitrary length. It is an abstract notation that specifies a memory buffer that is to be sent to another process or machine over IPC or network. All data must undergo marshalling before insertion into the octet stream. Marshalling involves conversion of machine data into CDR format and then performing byte alignment at the word boundaries.

Since CDR uses excessive alignment at the word boundaries, this chapter proposes new encoding rules based on CDR to generate time-efficient stub code. Section 6.2 proposes changes in the representation of boolean arrays in CDR, while section 6.3 and 6.4 propose changes in alignment in CDR. Section 6.5 proposes a novel link level encryption mechanism. Section 6.6 demonstrates the improvements obtained on incorporating these mechanisms.

### 6.2 Proposed Representation of Boolean arrays in CDR

The boolean data type in CDR is represented as an octet. In CDR, the boolean array of size 10 requires 10 bytes as shown in figure 6.1a. In the proposed representation, it is represented in the form of bits. Hence a Boolean array of size 10 requires 2 bytes as shown in figure 6.1b. This
method is particularly beneficial when images are transmitted over the network.

<table>
<thead>
<tr>
<th>byte</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

| bit  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 16 |

Figure 6.1a: CDR representation for boolean array

Figure 6.1b: Proposed representation for boolean array

The improvements in RTT obtained on making the proposed changes is shown in section 6.7.1

6.3 Proposed Changes in the Alignment Policies of CDR

In CDR, the alignment of all primitive types occurs on their natural boundaries, where an alignment boundary is the size of the primitive datum in octets. A primitive datum of size 'n' starts at an index in a stream that is a multiple of 'n'. For example consider a big-endian byte stream with the following types and values, “boolean = True, long = 132, short = 56, char = Q, long long =56”, the aligned CDR is shown in figure 6.2a. It requires 24 bytes to transfer the data. If the alignment is removed at the word boundaries, the unaligned CDR requires 16 bytes as shown in figure 6.2 b.

![Figure 6.2a: CDR Octet stream](image)

![Figure 6.2b: Proposed unaligned CDR Octet stream](image)

From figures 6.2a and b, it can be seen that due to alignment constraints, CDR requires 24 octets whereas the proposed method requires
only 16 bytes to transfer the data. This amounts to 33% increase in the number of octets.

In the case of a structure also the elements undergo alignment. For example consider a structure, my_struct defined in figure 6.3a.

```c
struct my_struct {
    char c;
    short s;
    char c;
    long l;
    long long d;
};
```

**Figure 6.3a: Structure my_struct definition**

Let the char type have value 'a' and all the other elements have a value 1. CDR aligned format for a big-endian stream is shown in figure 6.3b and the proposed representation is shown in figure 6.3c. From figure 6.3a and b, it is seen that 24 octets are needed in CDR format whereas only 16 octets are required in the proposed representation. This amounts to 33% reduction in the number of octets.

**Figure 6.3b: CDR Octet Stream for my_struct**

```plaintext
0x61 0x20 0x00 0x01 0x65 0x20 0x20 0x20 0x00 0x00 0x00 0x01 0x20 0x20 0x20 0x20 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
```

**Figure 6.3c: Proposed Unaligned Octet Stream**

The worst case alignment occurs when an IDL data type aligned at offset zero, is followed by a data type aligned at offset 8. The padding overhead introduces greater length messages. This in turn is decided by the
order of the parameters of the operation signatures and the order of the elements in a structure.

Inlining of marshalling procedures in the stub code has also been carried out for some data types. This is especially beneficial for short, long and character, to improve the marshalling speed. The size of the stub code is also not increased very much for these data types due to marshalling.

The improvements in RTT obtained on making the proposed changes is shown in section 6.7.2.

6.4 Improvements in Presentation Conversion by Reordering of the CDR octet stream

Byte ordering and alignment in CDR inhibits its ability to optimize for speed. Section 6.3 proposes a method to improve RTT by eliminating the alignment at the word boundaries. As hardware uses the concept of byte alignment to improve efficiency of operations, misalignment causes hardware complexities and misaligned memory accesses take multiple aligned memory references. Thus this method reduces the performance of unmarshalling when a machine requires alignment at the word boundaries. So this section proposes a method to improve RTT as well as to maintain the interoperability in CDR. For this a reordering module is used.

A reordering module rearranges the parameters in the invoking methods. Then the octet stream is constructed in CDR format. At the receiver end, the data is unmarshalled and is reordered back into a format suitable to invoke the remote method.

Consider a big-endian byte stream as parameters of a remote method in the following order along with the values “boolean = True, long = 132, short = 56, char = Q, long long =56”. Figures 6.2a and 6.4 show the representation for CDR before and after ordering the parameters in descending order of their size. The octet stream is reordered as “long long =56, long = 132, short = 56, char = Q and boolean = True”. It is seen that before ordering, the octet
stream requires 24 bytes, where as after ordering only 16 bytes are needed. This accounts to 33% reduction in size of the octet stream.

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
0x00 0x00 0x00 0x01 0x00 0x01 0x61 0x01
```

Figure 6.4: Reordered CDR

Similarly in the case of a structure shown in figure 6.3a, it is seen that CDR before ordering requires 24 bytes as shown in figure 6.3b. In the reordering module, the order of the elements in the structure is changed as "long long, long, short, char, char". The octet stream after reordering the parameters is shown in figure 6.5. The number of octets required is 16. Hence there is a reduction in the size of the octet stream by 33%.

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
0x00 0x00 0x00 0x01 0x00 0x01 0x61 0x61
```

Figure 6.5: Reordered CDR for my_struct

The design of the CORBA system with the reordering module for static and dynamic invocation of the CORBA objects has been shown in Section 6.6. The improvements in RTT obtained on making the proposed changes is shown in section 6.7.2.

6.5 Proposed Enhancements in Link Level Security features

As CORBA is fully platform and language independent, it has a high potential to provide security to distributed systems. CORBAsec is a powerful toolkit for secure, distributed applications rather than a plug-in that automatically secures CORBA systems [181]. The interface specified for security of client-target object invocations should hide the security mechanism used from both the application object and ORB. Security mechanisms that have been considered so far include Kerberos, Simple Public Key Mechanism (SPKM), Secure European System for Applications in a Multivendor Environment (SESAME) and Secure Socket Layer (SSL). Pluggable protocols
represent an abstraction of the Object Request Bus (ORB) transmission mechanism [182][183]. It facilitates the replacement of the transmission mechanism used by the ORB. CORBA also uses Link Level Encryption mechanism [184]. Link-Level Encryption (LLE) establishes data privacy for messages moving over the network links. The objective of LLE is to ensure confidentiality in transmission of CORBA application-generated messages. It employs the symmetric key encryption technique, specifically RC4, which uses the same key for encryption and decryption.

This section proposes a novel symmetric key crypto algorithm to provide efficient link level security for CORBA based applications. The crypto algorithm uses a 128-bit key. It comprises of a 64-bit key called the 'base', $K_b$ and another 64-bit key called 'randsrc', $K_o$. The algorithm takes a 48-bit plain text block and encrypts it into a 64-bit cipher. Both the encryption and the decryption process execute a set of computationally complex initialization procedures using the 128-bit key. These procedures generate the 'base' domain and the 'rand' domain each of $2^{64}$ words, each word comprising of 64 bits. The core of the remaining encryption process consists of mapping of the plain text with a base string to validate various functions. A function thus validated undergoes another transformation to give the cipher. The decryption process is a logical backward retrace of the encryption process. A detailed description of the encryption and decryption mechanism is given in the following sub-sections.

### 6.5.1 Encryption

This section describes the process of encryption as illustrated in Fig. 6.6. Two 64-bit keys namely 'base' and 'randsrc' are given to the initialization block which executes a series of swap and bit-wise operations iteratively to build two domains namely the 'base' domain and the 'rand' domain each of $2^{64}$ words. This is a computationally complex block requiring $2^{64}$ iterations and it is also the source for randomness in the encryption process. The large size of the 'base' domain makes the algorithm invulnerable to many attacks. This phase also aids to provide 'unconditional security' to the proposed algorithm which is illustrated in the chapter 6.7.4.
The subsequent block namely the 'base string constructor' picks up 48 words from the large base domain randomly to frame the base string (basestr). The source of randomness is the 'rand' domain. The base string 'basestr' (48 words), the plaintext $P_0$ (48 bits) and a random parameter

Figure 6.6: Schematic diagram of the proposed method of encryption

The subsequent block namely the 'base string constructor' picks up 48 words from the large base domain randomly to frame the base string (basestr). The source of randomness is the 'rand' domain. The base string 'basestr' (48 words), the plaintext $P_0$ (48 bits) and a random parameter
rand[val](64 bits) are passed to gmape block. This block evaluates functions as shown in the figure 6.7.

**Figure 6.7:** Schematic Diagram of valid function generation in gmape

**Figure 6.8:** Schematic diagram of function validation in gmape
The random parameter FX is given to the 'master function generator' block which frames the 64-bit value into a polynomial function to give \( f(x) \) (figure 6.7) The \( f(x) \) so formed is given to the 'validator' block which tests for the validity of the function as shown in figure 6.8.

The index vector is first generated from \( f(1:48) \). It is used to build the observed vector. The observed vector is built by traversing basestr and obtaining the bit values at the appropriate locations specified by the index vector. The result vector is the XOR of the input and the observed vectors (48 bits). If the result vector is null, then the function under evaluation is valid. If not, then we evaluate the next function obtained by incrementing FX.

Once we get a valid function of 64 bits, the base string (48 words) is passed to the 'agglomerator' which transforms it into baseX of 64 bits. XOR between function and baseX yields the cipher.

6.5.2 Decryption

The decryption process follows the same suit as that of the encryption process with regards to the initialization procedure and the construction of the 'base' domain and the 'rand' domain. The base 'string constructor' constructs a 48-word basestr and the 'agglomerator' generates the 64-bit baseX in the same manner as that of the encryption process.

Once baseX is generated, XOR of the cipher and baseX yields FX which is given to the 'gmapd' block shown in figure 6.9. This block transforms FX into a polynomial function \( f(x) \) and then builds the 48-bit index vector from \( f(1:48) \). The bit values in the corresponding locations as given by the index vector yields the 48-bit plain text block.
6.5.3 Algorithm:

The algorithm for the initialization procedure is shown in figure 6.10. It is common to both the encryption and decryption process. It is used to build the random base domain and rand domain by using iterative bitwise operations. base2 and rand2 are generated by using XOR operation on three octets of keys $K_b$ and $K_o$. It is mainly used to bring about randomization and to increase the complexity of key generation procedure.

The basestr is formed from 48 words in the base array. A random function is chosen from the random array. Using these, the gmape algorithm,
encrypts the plain text to form the cipher text as shown in figure 6.11. First the observed vector is formed using the base array, formed from 'basestr'. It is 'XOR'ed with input string. If both match, then the function is chosen. To randomize the starting point for function generation, rand[val] array is used.

```
Initialization _ procedure
{
    base1 = K₀  rand1 = K₀
    for index = 1 to 2^64
        rand[index] = index
    end for
    for index = 1 to 2^64
        base2 = function1(base1)
        rand2 = function1(rand1)
        Swap(rand[base2],rand[rand2])
        base1 = base1 xor rand[base2]
        rand1 = rand1 xor rand[rand2]
        base[index] = base1
    end for
}
function1(arg)
{
    XOR 3 octets
}
```

Figure 6.10: Encryption/Decryption Initialization Algorithm
Encryption_procedure
{
    call initialization();
    ind = K
    For index = 1 to Plaintext blocks(of 48 bits each)
        val = rand[ind+index]
        basestr = base[base[val + 0]] + base[base[val + 1]]
        + base[base[val + 2]] + base[base[val + 3]]
        + .... base[base[val + 47]] // concatenation
        FX = gmape (basestr, rand[val], plaintext)
        baseX = function2(basestr)
        Cipher = FX xor baseX
        ind = rand[baseX]
    end for
}
Algorithm function2(arg)
{
    ret(XOR all 48 words)
}
cipher_text gmape(basestr, Func, plaintext)
{
    Frame the function f(x) from Func
    for l = 1 to 48 (plain text block size)
        index[l] = f(x)
        observed[l] = base[(index[l] % 3072)]
        //basestr has 48 x 64 = 3072 bits
        input[l] = toBits(plaintext)
    end for
    output[] = observed[] xor input[]
    if output[] = 0 then
        return(FX)
    else
        gmape(basestr,(FX+1),plaintext)
    }
}

Figure 6.11 : Encryption Procedure
The decryption algorithm (figure 6.12) uses the same initialization procedure as encryption algorithm. Using the function and 48 words of the base array, the plain text is extracted.

```plaintext
Decryption_procedure
{
    call initialization()
    ind = K₀
    for index = 1 to Plaintext length or Plaintext blocks
        val= rand[ind+index]
        basestr = base[base[val + 0]] + base[base[val + 1]]
            + base[base[val + 2]] + base[base[val + 3]]
            + .... base[base[val + 47]]
        baseX = function2(basestr)
        ind = rand[baseX]
        FX = Cipher xor baseX
        Plaintext = gmapd(basestr,FX)
    end for
}
Algorithm function2(arg)
{
    ret(XOR all 48 words)
}
pt gmapd(basestr,FX)
{
    Frame the function f(x) from FX.
    for i = 1 to 48
        index[i] = f(x)
        pt[i] = base[(index[i] % 3072)]
    end for
    return(pt)
}
```

Figure 6.12 : Decryption Procedure
6.6 Design of the CORBA system

The design of the CORBA system with the proposed changes is given in this section. An Implementation Repository (IMR) is a repository, which maintains the mapping between the abstract object name and its physical address. It has the following fields:

- IP address and adapter name
- Names of registered classes from the CORBA server.
- Object path

When a new class defining the operations in an interface is implemented in the CORBA server, it is registered in the implementation repository (IMR).

Figure 6.13: Remote CORBA Object Invocation
The various steps involved in invoking a remote CORBA object as shown in figure 6.13. Steps 0 to 4 depict the steps involved in returning the OR from IMR to the client. It is explained in figure 5.27. The other steps are explained below:

5) From the Interface Definition of the remote object, Interface Repository (IR) is constructed.
6) The function and its parameters are checked for validity using the Interface Repository.
7) The IDL file is given to the ordering module, where the parameters are reordered
8) The ordering module produces the optimized IDL file. Both the IDL file and its optimized version are used in the generation of static stubs and skeletons.
9) Call invocation is done dynamically or statically as shown below:
   a. During dynamic invocation, the client issues the operation request to the CORBA server object through Dll, using OR.
   b. During static invocation, the client issues the operation request to the CORBA server object through Sll, using OR.
10) Ordering and marshalling of the parameters takes place in the stubs.
    In Dll the following operations are done:
    a. The parameters and composite types are reordered for optimization
    b. The realigned parameters are converted to CDR format, encrypted and it is transported over IIOP to the server object.
    c. SII is generated using optimised IDL file to maintain an orderly encoding of the parameters. In SII the parameters are encoded in ordered format and encrypted.
11) The data in CDR format is encrypted using the proposed method and it is transmitted through the IIOP.
12) Re-ordering and unmarshalling of the parameters takes place in the skeletons.
    In DSI the following operations are done:
a. The parameters are decrypted and then decoded from CDR to local machine format.
b. They are aligned in proper order and the function call is assembled.
c. SSI is generated using optimised IDL file to maintain an orderly decoding of the parameters. In SSI the parameters are decrypted, decoded and then reordered.

13) Invoking of the CORBA object function.

a. The server object function is invoked from the functional call assembled in DSI and the function is executed.
b. The server object function is invoked from SSI and the function is executed.

14) The CORBA server returns the results of the operation to the client in the similar manner. The reordering occurs if a composite type, like a structure is returned as the result.

CORBA objects can be transient or persistent objects. The invocation methods can be static and dynamic. The time taken to transfer a request from client to server is calculated as described below:

\[ T_{\text{trans}}(n) \]: It is transmission time needed to send \( n \) octets in CDR format.

\[ T_{\text{mar}}(\text{type}, \text{no}, \text{order}) \]: It is the marshalling time needed to convert data into CDR format. It is dependent on the data type of the parameter (type), number of parameters (no) and the order of the parameters (order). This includes the time to order, encode and encrypt the parameters.

\[ T_{\text{unmar}}(\text{type}, \text{no}, \text{order}) \]: It is the unmarshalling time needed to convert from CDR format to local machine representation format. This includes the time to decrypt, decode and reorder the parameters.

\[ T_{\text{req-build}}(\text{type}, \text{no}) \]: It is the time spent in dynamic invocations while querying for the Interface Repository (IR) to construct the request object.
\( T_{\text{look-up}} \): It is the time to look up for persistent objects using Implementation Repository (IMR).

\( T_{\text{exec}} \): Time taken by IMR to execute the startup command.

\( T_{\text{IMR}} \): Time needed for the client to get the OR from IMR.

\[ T_{\text{IMR}} = T_{\text{look-up}} + T_{\text{exec}} \] .................(6.5)

So in the case of dynamic invocation and transient CORBA object, the total time taken to transfer a request from client to server is:

\[ T_{\text{total-trans}} = T_{\text{mar}}(\text{type, no, order}) + T_{\text{trans}}(b) + T_{\text{unmar}}(\text{type, no, order}) + T_{\text{req-build}}(\text{type, no}) \] ............(6.6)

In the case of dynamic invocation and persistent CORBA object, the total time taken to transfer a request from client to server is:

\[ T_{\text{total-pers}} = T_{\text{mar}}(\text{type, no, order}) + T_{\text{trans}}(b) + T_{\text{unmar}}(\text{type, no, order}) + T_{\text{req-build}}(\text{type, no}) + T_{\text{IMR}} \] .................(6.7)

In the case of static invocation and transient CORBA object, the total time taken to transfer a request from client to server is:

\[ T_{\text{total-trans}} = T_{\text{mar}}(\text{type, no, order}) + T_{\text{trans}}(b) + T_{\text{unmar}}(\text{type, no, order}) \] ............(6.8)

In the case of static invocation and persistent CORBA object, the total time taken to transfer a request from client to server is:

\[ T_{\text{total-pers}} = T_{\text{mar}}(\text{type, no, order}) + T_{\text{trans}}(b) + T_{\text{unmar}}(\text{type, no, order}) + T_{\text{IMR}} \] ............(6.9)

Section 6.7.3 shows a quantitative improvement in the performance of CDR due to proposed modifications.
6.7 Experimental Results

6.7.1 Proposed Representation of Boolean arrays in CDR

An IDL compiler has been designed to generate the stub code. A performance analysis of the stub code, with the proposed modifications, has been carried out. The measurements have been carried out with Linux 7.1 as the operating system. The round trip travel time (RTT) which consists of the marshalling, unmarshalling and network transmission time (both the ways) has been measured. The measurements are repeated and the average is taken.

Figure 6.14: RTT for Boolean array for the proposed model and CDR

CDR, XDR and NDR do not use bit representation for Boolean arrays. Figure 6.14 shows the RTT for Boolean array of various sizes for the proposed method and CDR. In the proposed method, as each element of the boolean array is encoded as a bit, its performance is much better than CDR. It is also seen that as the size of the array increases this difference in performance becomes more prominent.

The proposed method also adopts inlining of procedures for some data types like integer, character and long. It is found that the performance of the stub code is much better in the case of primitive types, but it is more prominent in the case of composite data types like arrays and structures as
shown in figures 6.15a and 6.15b. In figure 6.15c, the combined interface consists of five operations with various combinations of primitive data types like char, int, long, float and double.
6.7.2 Proposed Changes in the Alignment Policies of CDR

This section gives the experimental results for the RTT obtained in CDR with and without alignment. The RTT is compared with that for an object invocation using TCP/IP sockets. Figure 6.15 shows the performance of the stub code for the different data types – char, short, float, float array of size 256, struct. From the figure 6.15 it is seen that proposed method performs much better than CDR representation.

6.7.3 Improvements in RTT due to reordering the CDR stream

A performance analysis of marshalling with proposed modifications on reordering the octet stream has been carried out. The measurements have been carried out with Linux 7.1. Pentium IV - 2.6 GHz computers with 512 MB RAM has been used. The total invocation time has been calculated. The measurements have been repeated and are averaged.

The combinations of parameters considered in an example interface ex1 are listed below:
- cs – character and short
- cd – character and double
- cl – character and long
- cf – character and float
- ex1 – char, float, boolean, short and double
- struct – with char, double, boolean, long, char and float as elements
- c_struct – with char and struct as parameters

a) Performance improvement for dynamic invocation of objects

From the figure 6.16a, it can be seen that there is only marginal increase of performance, when less amount of data is transmitted. In the case of transmitting complex data types like a structure, the performance increase is 4.98%. When a structure is transmitted along with a character, the performance increase is 8.54%. For arrays, since all the elements are of the same type, no reordering can be done. So there is no difference in performance between an ordered and unordered octet stream in the case of arrays. It is also noted that the time taken for ordering/reordering is very less. It is only 10 microseconds. Thus adding the ordering module does not impact much on the performance of the system.

Figure 6.16a: Performance of ordered and unordered CDR Octet Streams in Dynamic Invocation
Fig 6.16b compares the time taken for referring to IR, marshalling, transmission and unmarshalling of ordered and unordered octet streams with primitive types as its parameters. Fig. 6.16c shows the same for an octet stream with struct as its parameter. The transmission time is high in dynamic invocation, since the time taken to pass the parameter from client to generic stub code and the transmission time from stub code to server have to be considered.

**Figure 6.16b:** $T_{\text{req-build}}, \text{order, T\text{mar}, T\text{trans}, T\text{unmar, reorder, T\text{total}}}$ for ex1
In Dynamic Invocation (ordered and unordered octet stream)

**Figure 6.16c:** $T_{\text{req-build}}, \text{order, T\text{mar}, T\text{trans}, T\text{unmar, reorder, T\text{total}}}$ for struct
In Dynamic Invocation (ordered and unordered octet stream)
b) Performance improvement for static invocation of objects

From the figure 6.17a, it can be seen that there is only marginal increase of performance, when less amount of data is transmitted. In the case of transmitting complex data types like a structure, the performance increase is 18.9%. When a structure is transmitted along with a character, the performance increase is 19.1%. For arrays, since all the elements are of the same type, no reordering can be done. So there is no difference in performance between an ordered and unordered octet stream in the case of arrays. It is found that the static invocation is much faster when compared to dynamic invocation method.

![Figure 6.17a](image)

*Figure 6.17a: Performance of ordered and unordered CDR Octet Streams for Static Invocation*

Fig. 6.17b compares the time taken for refer marshalling, transmission and unmarshalling of ordered and unordered octet streams with primitive types as its parameters. Fig. 6.17c shows the same for an octet stream with struct as its parameter. Here ordering/reordering parameter time need not be considered, since it is done statically.
6.7.4 Analysis of the encryption system

The proposed symmetric key encryption algorithm has the following properties:

1) Each bit of the cipher depends on all bits of the plaintext
2) Altering a single plaintext character or key bit alters each cipher bit with probability 0.5

3) Altering a cipher bit results in an unpredictable change to the recovered plaintext block.

4) An analysis of the storage complexity for 64-bit and 128-bit algorithm, based on the number of operations is shown in table 6.4. From this table, it is clear that the encryption and decryption algorithm is highly complex.

Table 6.4: Complexity Analysis Table

<table>
<thead>
<tr>
<th>Attack Method</th>
<th>Data Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64 bit</td>
<td>128 bit</td>
</tr>
<tr>
<td>Exhaustive Search</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

5) Unconditional Security:

Since the base used is very large, different bases can be provided for every plain text block. On an average, if $n^{th}$ plaintext block uses the 48 word base $K_b$, the probability for the $(n+1)^{th}$ plaintext is $2^{48} P_{48}$.

Computationally complex procedure generates base array. The size of rand array is $2^{64}$. Hence polynomial attacks are of the order $O(2^{64})$. Though they are theoretically feasible under the model, it is computationally infeasible.

The many-to-many correspondence existing between the plaintext and cipher text in our algorithm protects it from the following attacks:

- Cipher-text only attack
- Known plaintext attack
- Chosen plaintext attack

A plaintext block $P_1$ uses a base $B_1$ to generate a cipher $C_1$. If the same plaintext block $P_1$ occurs again it will use a different base $B_2$ (Since base is independent of plaintext pattern) and generate cipher $C_2$. Here we can see that the one-to-many relationship is established (table 6.5).

Table 6.5: One to many mapping of plain text to cipher text

<table>
<thead>
<tr>
<th>Plain Text</th>
<th>Base</th>
<th>Cipher</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$B_1$</td>
<td>$C_1$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$B_2$</td>
<td>$C_2$</td>
</tr>
</tbody>
</table>
On an average with a constant base, a plaintext can generate $2^{16}$-cipher text. Because of this one-to-many relationship, cipher-text only attacks fails (figure 6.19).

A cipher $C_1$ can be obtained from two different bases $\text{base}X_m$ and $\text{base}X_n$ as given below:

$$C_1 = f_{X_n} \text{ XOR } \text{base}X_m.$$  

$$C_1 = f_{X_l} \text{ XOR } \text{base}X_n$$

Two different plaintext blocks can thus the same cipher (figure 6.20).

The total number of base patterns that could be obtained is $2^{64} \times 48$.

$$= 2^{64} \times (2^{64} - 1) \times (2^{64} - 2) \times \ldots \times (2^{64} - 47) = 2^{64} \times 48$$
Because of the existence of large number of base patterns, the known plaintext and chosen plaintext attacks are practically infeasible.

6) A theoretical analysis of the performance of the encryption algorithm on different machines has also been done based on the number of the operations involved in the algorithm and the processing speed of the machines. The time taken to hack data that is encrypted using the proposed algorithm, with two different key sizes is tabulated in table 6.6. An analysis on key generation, shows that around $2^{64}$ are are to be tried on an average, for valid keys. 128 ANUPAM takes $202 \times 10^9$ sec for one operation. Hence a 128 bit key takes $(2^{64})/(202 \times 10^9)$ seconds or 2.9 years. Similarly it takes 0.021 seconds for a 64-bit key.

7) The algorithm was also practically executed and the timing measurements have been taken. The time taken to hack data that is encrypted using the proposed algorithm, with two different key sizes is tabulated in table 6.6.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
System name & 64-bit & 128-bit  \\
\hline
128 ANUPAM & 0.021 sec & 2.9 years  \\
PARAM 1000 & 0.04 sec & 5.85 years  \\
NEC's Earth Simulator & 0.0001 sec & 5.21 years  \\
HP's Itanium 2 & 0.0035 sec & 0.05 years  \\
\hline
\end{tabular}
\caption{Theoretical Analysis of the time taken to hack data encrypted using the algorithm}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Processing Speed & 64-bit & 128-bit (years)  \\
\hline
1.7 GHz & 58 Hours & $3.2 \times 10^5$  \\
2.1 GHz & 46 Hours & $2.1 \times 10^5$  \\
\hline
\end{tabular}
\caption{Practical Analysis of the time taken to hack data encrypted using the algorithm}
\end{table}
6.8 Summary

CORBA is highly suitable for distributed computing environment, since it provides a flexible communication medium. But CORBA suffers from communication efficiency. This chapter explains the changes incorporated in CORBA encoding rules to reduce the size of the data passed in the network and hence reduce the marshalling time.

The boolean data type is represented as an octet in CDR. It is represented in bit format. On using this bit format, it is found that the throughput increases exponentially with the size of the Boolean array. Removing the alignment at the natural word boundaries, eliminates the extra padding bytes. Using this representation, the RTT for the operations using primitive and composite types is much less than RTT in the case of CDR format. It depends on the size of the parameters being transmitted. The send and receive buffer space are allocated depending upon the size of the data being transmitted. The back end of the compiler analyses the type of parameters being transmitted by an operation and allocates the correct size of buffer statically.

The order in which the parameters are passed in an operation also affects the efficiency of transmission. When the parameters are reordered in the descending order of size, the number of padding bits in CDR can be minimized. This can be done by regenerating the interface file by the presentation generator to minimize the wastage of space and reassembling the bits in correct order at the server side. It facilitates interoperability as well as reduces RTT. The system has been tested for static and dynamic invocation mechanisms. Depending on the data types transmitted as parameters and their ordering, this method achieves an improvement in the order of 2% to 20%.

LLE employs symmetric key encryption technique to ensure confidentiality. When LLE is being used, the CORBA system encrypts data before sending it over a network link and decrypts it as it comes off the link. A novel link level encryption scheme to bring about security in CORBA based
applications has been proposed. It uses a novel advanced polynomial transformation algorithm, for efficient encryption and decryption. It makes use of algebraic polynomial functional mappings for the plaintext to cipher transformation, which provides for poly-alphabetic substitution. The proposed method transforms the plaintext blocks of size 48 bits into 64-bit ciphers achieving an input-output bit transformation ratio of 3:4. The properties of this cryptoalgorithm has been evaluated and based on this, the cryptoalgorithm is found to work efficiently.