Literature Survey
CHAPTER 2

LITERATURE SURVEY

2.1 Introduction

Request brokers like Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), The Adaptive Communication Environment (ACE) ORB (TAO) and Distributed Component Object Model (DCOM) shield the application developers from low-level, tedious and error prone platform details. They thus provide an easy method for transparent presentation conversion.

A study and analysis has been carried out on the existing techniques in presentation conversion. Clark and Tennenhouse [36] have demonstrated that presentation conversion is responsible for 97% of the total protocol overhead. Message Passing Interface (MPI), Java RMI, DCOM, CORBA and XML provide transparent presentation conversion procedures. MPI [37][38] library has good performance, but lacks flexibility. It uses interpretation and does not have facility for dynamic discovery of data types of messages. Java RMI marshals data using Java object serialization and so is language dependent [1]. While DCOM is platform dependent [3], CORBA provides interoperability support [5]. CORBA 3.0 [39] offers components, objects by value and interfaces for object implementation, but CORBA suffers from communication efficiency. XML offers flexibility but suffers from high encoding/decoding costs [40][41].

A survey of literature on the performance and reuse analysis in different distributed object models is listed in Section 2.1. Section 2.2 lists the literature survey on CORBA object life cycle and inheritance. Survey of literature on the current language and kernel specific marshalling optimizations has been conducted and it is listed in section 2.3. Section 2.4 gives an analysis on common encoding mechanisms. Section 2.5 lists the common security mechanisms used in CORBA. Section 2.6 gives an analysis
on the existing methods for generation of stub code in CORBA based systems.

2.2 Analysis of Distributed Object Models

Distributed Object Models facilitate the usage of components. Component based software engineering [42] brings forth reusability at a higher level. Atkinson [43] discusses the major architectural issues affecting component-based software development. It includes versioning, representation of components and architectural dimension of the components, but not the relationship between the different dimensions. Component based development and component reuse [44] are established practices. Component based software development is employed in software industries to an extent of 40% to 80%. EJB and COM are used to an extent of 38% each in the software companies, while CORBA and other component models contribute towards 19% and 5% each [45]. SYNTHESIS [46] project investigates on the issues involved in a systematic approach to develop component based information systems. This includes the specification requirements, canonical modeling facilities, modeling facilities for all phases of component development and mapping from canonical modeling facilities to formal ones. Two-level architecture of a hierarchical workflow management system [47] is modeled and developed using a coordination language and components. The main advantage of a hierarchical, coordination-based architecture is that individual workflow entities can be easily replaced with others, without disrupting the overall workflow process. Each individual workflow entity exhibits a certain degree of flexibility and autonomy. This makes possible the construction of workflow systems that bring improvements to process automation and dynamic management, such as dynamic allocation of activities to actors and reusability of coordination (collaboration) patterns. The design of distributed applications in a CORBA based environment can be carried out by means of an incremental approach, which starts from the specification and leads to the high level architectural design. Formal specification can be written in TRIO and a high level design document written using an extension of TRIO, a logic language for execution
specification of RT systems [48] named TC (TRIO/CORBA). The TC language is suited to formally describe the high level architecture of a CORBA based application.

Schmidt [49] has discussed the implementation of a highly-flexible, just-in-time dynamic linking framework using CORBA Component Model (CCM) in TAO. This framework reduces the memory footprint of TAO and CCM through dynamic configuration of middleware infrastructure, components and services. CORBA Portable Interceptors are objects that an ORB invokes in the path of an operation invocation to monitor or modify the behavior of the invocation transparently [50]. Implementing portable interceptors using metaprogramming mechanism and components facilitates applications to extend and control the behavior of a framework. The ADAPTIVE Service eXecutive (ASX) framework [51] is an object-oriented framework composed of automated tools and reusable components. These tools and components help to simplify the development, configuration, and reconfiguration of applications in a distributed environment. ADAPTIVE [52] componentized framework helps in determining the functionality of customized lightweight protocol configurations that efficiently support multimedia applications and mapping this functionality onto efficient parallel process architectures. Conduit+ [53] is an OO framework for configuring network protocol software to support ATM signaling. Key portions of the Conduit+ protocol framework such as demultiplexing, connection management and message buffering are designed using patterns like Strategy, Visitor, and Composite [54]. ZEN-kit [55], a componentised graphical design tool for using CORBA in embedded systems, allows application developers to model and build low-footprint, custom middleware for the specific functionality required by each application. The goals for ZEN-kit include control over middleware customization, to achieve minimal memory footprint and minimizing the difficulty of this custom configuration for designers of distributed, embedded systems. PARallel And DIStributed (PARDIS) approach to CORBA [56] provides interoperability between heterogeneous components in parallel and distributed applications, by specifying their interfaces in a meta-language, the CORBA IDL, which is translated into the language of interacting components.
Some of the important Distributed Object Models include RMI, CORBA and DCOM. RMI [57] implementations are built on top of TCP/IP, which is not suitable for high Performance Computing. NexusRMI [58] implements Java RMI on top of NexusJava. NexusJava is a Java interface to the Nexus communication systems. NexusRMI has support for object serialization of all object types except for exception objects. NinjaRMI [59] has multiple communication protocols. Asynchronous RMI (ARMI) [60] adds asynchronous remote method invocation to RMI. It achieves asynchronous communication by replacing the RMI client-side stub, adding a thread mechanism. Reflective RMI (RRMI) [61] adopted the reflection feature of Java programming language to invoke remote methods. RRMI makes it possible of use RMI without stubs and skeletons. A remote object is created dynamically at the run time by implementing its own class loader on the server side. It also supports both synchronous and asynchronous remote method invocations. MantaRMI [62] is based on native compilation and generates executable code rather than byte code for better optimization. The serialization code is generated by the compiler which makes it possible to avoid the overhead of dynamic inspection of the object structure. Krishnaswamy [63] improves RMI performance by adding object caching at the reference layer and using UDP-based reliable message delivery protocol. KaRMI [64] improves the performance of RMI by providing interfaces between layers and the separation of RMI's design from transport level sockets. Its serialization uses explicit marshalling/unmarshalling routines instead of reflection and a minimal encoding of type information to achieve better performance.

Rod [65] gives the details of the architecture, implementation and services provided by the Distributed Computing Environment (DCE), the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM) and Java RMI. Emerald [66] gives an architectural comparison of DCOM and CORBA at three different layers, namely, basic programming architecture, remoting architecture and the wire protocol architecture. Ronald [67] and Owen [68] discuss the factors influencing the decision on the choice of COM or CORBA distributed object technologies for an application. The performance assessment framework for
distributed object models [69], considers the performance criteria for all important aspects of distributed object computing, including single and multi-client scenarios. It presents the high and low level design of the framework and gives insights into implementation details for several important distributed object models, like CORBA, RMI and RMI-IIOP. An analytic CORBA model [70] suggests that marshalling and demarshalling has a significant impact on communication transmission times and is dependent upon the IDL specification. This thesis presents a performance and reuse analysis on RMI/JRMP, RMI/IIOP, CORBA and DCOM distributed object models. Based on this analysis, the major sources of overhead in CORBA have been identified.

2.3 Distributed object lifecycle and inheritance

The Object Lifetime Manager pattern [71] is a complementary pattern which addresses issues associated with object destruction. It completes creational patterns by considering the entire lifetime of objects and providing mechanisms for object destruction at program termination. Gutsavo [72] uses back tracking instead of mark and sweep to eliminate the need of global synchronization in garbage collection during destruction of objects. The algorithm chooses an object that is likely to be garbage. Then the algorithm recursively traces back the references to find the transitive closure of objects that reach the suspect. If this closure does not contain any root, the objects in the closure are garbage, and therefore collected. Reference lists with remote back-references to remote proxies from the exported objects and usage of local garbage collector to obtain back pointer information facilitate the implementation of the algorithm. CORBA objects use factories for object creation and should be explicitly destroyed by the client. This type of life cycle management service is implemented in Omnibuilder [73]. This leads to an underutilization of resources. This thesis proposes a novel method of creation/destruction of CORBA objects using reference counting mechanism.

Subclasses can inherit the implementation or interfaces of a superclass. Similarly selective inheritance can be based on the implementation or interfaces. IBM's System Object Model (SOM) [74]
supports implementation inheritance. SOM allows OO-style inheritance for in
process objects. It uses composition if the base class resides in a different
address space or host machine. Rules describing semantic information of the
subclasses can be inferred from their superclasses for subtype, parameterized and selective implementation inheritance [75]. A complex
object model based on specialization constraint (SC) to support selective
inheritance prevents a subclass from inheriting certain properties from its
superclass [76]. The above methods are based on selective implementation
inheritance. COM and CORBA specifications support interface inheritance.
Component Object Model (COM) allows only single interface inheritance.
Instead of supporting multiple inheritances, COM uses multiple interfaces to
an object. There is no selective interface inheritance in COM [77]. CORBA
supports multiple interface inheritance but not selective interface inheritance.
Omnibuilder [73], a CORBA implementation, supports selective
implementation inheritance only. This thesis proposes a novel method of
achieving selective interface inheritance in CORBA.

2.4 Language and Kernel Specific Marshalling Optimisations

Acute [78], a distributed language supports type safe marshalling,
rebinding to local resources, abstract types and version change marshalling
optimisations. Rossberg [79] uses marshalling with abstract types, using a
calculus with run-time new type generation and type coercion optimisations. It
is used in the Alice [80] project. Gavin Biermann [81] uses rebinding
marshalled values to local resources, in a simply-typed lambda calculus
setting. Phantom [82], an interpreted language for distributed programming,
has features of object based RPC systems and statically compiled languages,
including automatic marshalling, transparent remote procedure call, secure
authentication and concurrency support. Phantom's interpreted nature permits
the use of programming techniques, such as true object migration, remote
evaluation, and dynamic extensibility optimisations, not available in statically
compiled languages and RPC systems. Concert C [83] programs run on a
heterogeneous set of machine architectures and operating systems and
communicate over multiple RPC and messaging protocols transparently. But,
it does not support extra services like fault tolerance and group communication services. Concurrent C++ [84] implements concurrent programming facilities in 'C++'. It also takes advantage of data abstractions in C++. This combined advantage facilitates a robust, higher-level interface for the services provided by the processes. It also facilitates the use of data abstractions for parallel programming. Synchronous C++ (sC++) [85] defines active objects that contain their own execution threads and can communicate with each other by means of synchronizing method calls in C++. When compared to C++, sC++ implements dynamic and functional models easily by providing a mapping between the models and sC++ statements. Concurrency is brought about by delayed evaluation, using asynchronous channels and running the active objects in parallel with other objects. In sC++ communication is established through active TCP/IP objects whereas in CORBA IIOP protocol is used. AspectlX [86] extends CORBA by generic quality-of-service support and the implementation of AspectlX needs completely different stub objects than standard CORBA. IDLflex [87] is used to design the stub objects for AspectlX. It can generate stub code for multiple languages using an XML-based mapping program and a language-specific utility class. Thus, IDLflex can be adapted to a number of target programming languages and ORB implementations in a very fast way, but it is specific to AspectlX. Mockingbird [88], a prototype tool for developing interlanguage and distributed applications, compiles stubs from pairs of interface declarations, allowing existing data types to be reused on both sides of every interface. Its stubs convert structurally equivalent types by interacting with the programmer to refine the original declarations. Other multilanguage stub compilers impose data types on the application, complicating development. PARDIS [89], a parallel approach to CORBA, extends the CORBA object model by introducing Single Program Multiple Data (SMPD) objects that represent data-parallel computations. A special data type called distributed sequence which is language specific is used to bring about distributed argument transfer.

Firefly RPC avoids excessive data copy by mapping messages to a common user space [90] (shared memory), which is platform dependent. X-kernel [91] implements protocols in a modular, highly structured way, without
sacrificing performance by optimising TCP/IP and RPC protocols for its platform. In the LRPC scheme presented by Bershad [92], client and server use privately shared memory for argument passing. The remapping approach uses move rather than copy semantics and is thus limited to situations where the client does not need further access to the data after the call. But, shared memory optimisations can compromise the security model of the system. Spring [93] is an object oriented operating system designed as a successor to UNIX. Spring has the notion of a subcontract which has similar functionality to the remote reference layer in the RMI system. This system combines aspects of both the Modula-3 Network Objects system and Spring's subcontract. It uses pickling to transmit arguments and return values and Java to dynamically load stub code to clients. Bartoli [94] uses a low level interpreter called a marshalling engine for marshalling. For this a new data type, namely, distributed sequence is used to improve the performance of the distributed applications in a more generic way. Devil [95] is a description language for the hardware. The description for the registers and memory for the device has to be included in the interface definition. The IDL compiler generates C communication code to perform the manipulations. The DROPS IDL Compiler (DICE) [96], for Dresden Real-Time Operating System (DROPS) project, is designed specifically for the L4 platform. The methods of optimizations suggested in [78-96] are specific to a particular kernel and/or particular programming language. These methods are thus not generic in nature.

The ADAPTIVE Communication Environment (ACE) [97] is an object-oriented (OO) toolkit that implements fundamental design patterns for communication software on UNIX and Win32 platforms. The ACE ORB's (TAO's) [98] pluggable protocols framework focuses on implementing and adapting to transport protocols beneath the standard CORBA programming API. TAO's GIOP messaging is implemented in the monolithic design and therefore each message type marshaler / demarshaler consumes memory footprint whether or not they are needed and used by an application. In contrast, ZEN's [99] pluggable GIOP messaging framework focuses on selecting a subset of the many standard CORBA GIOP message readers / writers. Therefore, existing communication frameworks can provide building
blocks for ZEN's pluggable GIOP messaging framework. This is possible because, ZEN's ORB Core and Object Adapter are designed using patterns like Virtual Component [100], Acceptor-Connector [101] and Active Object [102]. These ORBs use IIOP as their transmission medium. As IIOP faces speed limitations, this thesis proposes improvements in marshalling by incorporating changes in transmission medium to improve the performance of CORBA-based applications in a more generic way.

2.5 Common Encoding Standards

Some of the common standards used for the transfer of information between heterogeneous systems are Basic Encoding Rules (BER) [103], Sun's XDR [104] and Network Data Representation (NDR) [105]. Portable Binary Input Output (PBIO) uses Natural Data Representation (NR) of source machine with alignment and meta information to transfer data [106]. It is flexible, as marshalling takes place at run time. It uses caching of data to improve its performance, but does not have static stubs. CORBA [5] uses Common Data Representation (CDR) format of representation for data transfer. CDR uses extra padding bytes to achieve alignment of data. This thesis proposes certain changes in CDR alignment to improve the time efficiency of presentation conversion.

2.6 Common Methods of CORBA Security

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 [5]. 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). Embedded Information Systems Assurance (EISA) uses both network level (IPSec) and middleware (CORBA SSLIOP) features to make it suitable for defense
applications [107]. This thesis proposes a novel symmetric key encryption algorithm and its analysis to bring about link level security in CORBA based applications.

2.7 Time and Space Optimisations in Marshalling

Marshalling optimizations improve the performance of Distributed Real-time Embedded Systems (DRE). Application of optimization patterns [108][109] in TAO has lead to the development of time and space efficient CORBA IIOP protocol, which is highly suitable to be applied to distributed real time systems. RT CORBA [110] can be applied to a wide variety of mission critical applications. To enhance predictability, the Real-time CORBA specification defines standard middleware features that allow applications to allocate, schedule, and control key CPU, memory, and networking resources necessary to ensure end-to-end quality of service support. Ryan [111] discusses the design of flexible publisher/subscriber architecture for CORBA middleware suitable for DRE applications with low latency and jitter, periodic rate-based event processing, and event filtering and correlation. Multiple middleware QoS management technologies [112] help to manage quality and timeliness of imagery adaptively within a representative distributed avionics system and the performance of the integration on fundamental trade-offs between timeliness and image quality in that system.

The requirements for improved software quality and productivity motivate the use of Distributed Object Computing middleware for real-time embedded systems [113]. ZEN, a RTCORBA implementation [114] and jRate, a RT java implementation incorporate the strategies necessary to enable predictability for Real-time CORBA and Real Time Specifications for Java (RTSJ) using efficient demultiplexing strategies. Zen's POA [115] uses perfect hashing for O(1) time servant lookups and flattening the POA hierarchy to prevent a lookup for every level in the POA hierarchy and thus improves the efficiency for DRE systems. Zen uses micro-ORB architecture with real time CORBA and RT java features to minimize the memory footprint of the ORB used in DRE applications. The Virtual Component design pattern [100] helps to reduce the memory footprint of middleware by transparently migrating
component functionality into an application on-demand. It applies the factory method and component configurator design patterns to achieve its goals.

The interpretive stubs generated by TAO IDL [116] compiler achieve comparable performance to handcrafted stubs with a relatively less code size. Users can configure TAO IDL compiler to generate interpretive/compiled marshalling code on per-file or per-operation basis using command line options. Flick IDL compiler gives very high performance by using compiled stubs with aggressive inlining [117]. Rainer [118] proposes function inlining in C compilers for embedded processors by aiming at a maximum program speedup under a global limit on code size. The core of this approach is a branch-and-bound algorithm which permits the quick exploration of the large search space. Paul [119] has designed a two stage compiler. The first stage translates type system information into byte code optimized for describing marshalling optimizations. The second stage uses byte code string of the first stage to decide on whether to use synthesized stubs, cached stubs, fast marshalling or generic marshalling interpreter. OpenFusion e*ORB C Edition (CE) [31], an embedded real-time CORBA middleware product works on the Texas Instruments Incorporated Digital Signal Processor platforms. Its unique modular micro-kernel architecture reduces the size of ORB. ORBexpressGT is a commercial ORB. It is suitable for the development of embedded systems, where a lightweight, high performance CORBA implementation with a small memory footprint is required. It has a segmented architecture for minimal memory footprint. In ORBexpressGT and ORBExpressRT [32], DII and DSI mechanisms are not implemented. IONA Orbix, a commercial ORB, is reported to have good performance [11]. It treats DII and SII as two different invocation mechanisms. Interpretation can be used for flexible applications and compilation can be used to improve the speed of execution [120]. Deferring work at runtime avoids doing work that is never required [121] and hence brings about optimisation. By delaying the actual stub generation [122] until run time allows late adaptations to current needs and restrictions.

Hoschka [123] discusses about using control flow analysis to generate time and space efficient stubs using greedy methods. It aims to strike a
balance between interpretation and compilation. A Markov model in combination with a heuristic branch predictor is used for estimating execution frequencies. Only time objective is considered and it is used to generate stubs under space constraint. Pitmann [124] proposes using hybrid code between interpretation and compilation. It requires manual intervention for optimisation. Tempo [125] removes genericity of code at compile time by optimizing the output of the compiler to a particular platform. It consists of a preprocessing phase which forms semantic actions from the input program and a processing phase which executes these actions. This facilitates the integration of compile time and run-time specialization into the system. Core CORBA specifications do not propose any such specializations [126]. Embedded systems are subject to space constraints. Mayur [127] has proposed a code-size directed compiler with register allocation and code generation expressed as an integer linear programming problem, where the upper bound on the code size is expressed as an additional constraint. Although this compiler generates a compact code, it does not satisfy timing constraints for real time systems.

The stream of bytes carrying marshaled objects can be expressed as a sequence of expressions of some marshaling language [128]. This programmatic view adds more meaning to the exchange protocol, stresses the asymmetry between the marshaler (a compiler) and the demarshaler (an interpreter) and brings all useful language technologies to the level of marshaling. Data compression [129] techniques can be used to reduce the size of infrequently executed portions of the generated code. The compressed code is decompressed dynamically prior to execution. The use of data compression techniques increases the amount of code size reduction that can be achieved and the application to infrequently executed code limits the runtime overhead due to dynamic decompression. The primary motivation for flexible presentation is for programmer convenience and improved interoperability. But, flexible presentation using marshalling with annotated definitions can optimize RPC [130] by avoiding unnecessary user-space overhead and also minimizing the marshalling time.
Although CORBA facilitates flexible presentation conversion, the stub code generated suffers from size and speed limitations. So this thesis proposes a novel method to achieve a balance between the size and speed of the stub code using genetic algorithms.

An overhead and reuse analysis has been conducted on RMI, RMI/IIOP, CORBA and DCOM to identify the bottlenecks and to improve the performance of distributed object models.