İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

MARSHALLING/DEMARSHALLING PERFORMANCE ANALYSIS OF SUN'S JAVA IDL BY USING STATIC INVOCATION INTERFACE

M.S. Thesis by

Tacettin AYAR, B.S.

Department : COMPUTER ENGINEERING

Programme: Computer Engineering

JANUARY 2003

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

MARSHALLING/DEMARSHALLING PERFORMANCE ANALYSIS OF SUN'S JAVA IDL BY USING STATIC INVOCATION INTERFACE

M.S. Thesis by

Tacettin AYAR, B.S.

504991126

Date of submission : 24 December 2002

Date of defence examination: 15 January 2003

Supervisor (Chairman): Prof. Dr. Bülent ÖRENCİK

Members of the Examining Committee Assoc. Prof.Dr. Nadia ERDOĞAN (İTÜ.)

Dr. Erdal ÇAYIRCI (Turkish War Colleges)

JANUARY 2003

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

SUN'IN JAVA IDL DERLEYİCİSİNİN STATİK ÇAĞRI ARAYÜZÜ KULLANILARAK HİZALAMA/GERİ HİZALAMA BAŞARIMININ İNCELENMESİ

YÜKSEK LİSANS TEZİ

Müh. Tacettin AYAR

504991126

Tezin Enstitüye Verildiği Tarih : 24 Aralık 2002

Tezin Savunulduğu Tarih : 15 Ocak 2003

Tez Danışmanı : Prof.Dr. Bülent ÖRENCİK

Diğer Jüri Üyeleri Doç.Dr. Nadia ERDOĞAN (İ.T.Ü.)

Dr. Erdal ÇAYIRCI (Harp Akademileri)

OCAK 2003

FOREWORD

I am very grateful to my advisor Prof. Bülent Örencik for his guidance throughout this study.

Thanks to my family for their patience.

Special thanks go to Yıldıray Hazır and Ahmet Çağatay Tunalı for the materials they supplied me.

My colleagues and room mates dear Mehmet Siraç Özerdem, Ahmet Cüneyd Tantuğ and Gülşen Cebiroğlu Eryiğit, deserves special thanks. I am very indebted to them.

Tacettin AYAR

December, 2002

TABLE OF CONTENTS

ABBREVIATIONS

CORBA : Common Object Request Broker Architecture
ORB : Object Request Broker
BOMSIG : Business Object Model Special Interest Group
BLO : Business Logic Object
BPO : Business Process Object
OMG : Object Management Group
API : Application Programmer Interface
OMA : Object Management Architecture
OR : Object Reference
IOR : Interoperable Object Reference
IDL : Interface Definition Language
DII : Dynamic Invocation Interface
DSI : Dynamic Skeleton Interface
BOA : Basic Object Adapter
POA : Portable Object Adapter
IR : Interface Repository
GIOP : General Inter-ORB Protocol
CDR : Common Data Representation
IIOP : Internet Inter-ORB Protocol
TCP/IP : Transmission Control Protocol/ Internet Protocol
ESIOP : Environment-Specific Inter-ORB Protocol
SII : Static Invocation Interface
ATM : Asynchronous Transfer Mode
BNF : Backus-Naur Format
CPU : Central Processing Unit
PC : Personal Computer
RAM : Random Access Memory
IEEE : Institute of Electrical and Electronics Engineers
MSB : Most Significant Bits
LSB : Least Significant Bits
SDK : Software Development Kit
L_O_OS : Local Oneway Only Send
L_T_OS : Local Twoway Only Send
L_T_SG : Local Twoway Send Get
L_T_OG : Local Twoway Only Get
R_O_OS : Remote Oneway Only Send
R_T_OS : Remote Twoway Only Send
R_T_SG : Remote Twoway Send Get
R_T_OG : Remote Twoway Only Get

LIST OF TABLES

Page Number

LIST OF FIGURES

SUN'IN JAVA IDL DERLEYİCİSİNİN STATİK ÇAĞRI ARAYÜZÜ KULLANILARAK HİZALAMA/GERİ HİZALAMA BAŞARIMININ İNCELENMESİ

ÖZET

Tezimiz, dağıtılmış sistemlere genel bir bakışla başlamaktadır. Dağıtılmış sistemler aşağıdaki evreleri izleyerek gelişmişlerdir :

- Ana çatılar için yazılmış tek parçalı yazılımları kullanan *Tek Parçalı Sistemler.*
- Hizmetler sağlayan bir sunucu ve sunucudan hizmetler isteyen bir istemciden oluşan *İstemci/Sunucu Sistemleri.*
- Sistemi, kullanıcı arayüzü katmanı, iş kuralları katmanı ve veritabanı erişim katmanı olmak üzere parçalara ayıran *Çok Parçalı İstemci/Sunucu Sistemleri*.
- Uygulamanın tüm işlevselliğini, sistemdeki veya diğer sistemlerdeki diğer nesnelerin sağladığı hizmetleri kullanabilecek nesneler olarak sunan *Dağıtılmış Sistemler.*

Bundan sonra, dağıtılmış nesneleri ve bileşenleri tanıttık. Dağıtılmış nesneler ağ üzerinde herhangi bir yerde bulunabilecek genişletilmiş nesnelerdir. Bileşenler, değişik ortamlarda çalışan, sistemin en küçük kendi kendini yönetebilen, bağımsız ve kullanışlı parçalarıdır.

Bir iş nesnesi üç ana parçadan oluşmaktadır :

- *İş Mantığı Nesnesi*, nesnenin belirli olaylara karşı nasıl davranacağını tanımlar.
- *İş İşleme Nesnesi* tüm sistem için iş mantığının sağlanmasına yardımcı olur.
- *Sunum Nesneleri* kullanıcıya bileşenin bir gösterilimini sağlar.

Genel Nesne İstek Aracısı Mimarisi (CORBA), nesne sistemlerinin geniş çeşitlilikleri arasında entegrasyona izin vermesi için Nesne Yönetim Grubu (OMG) tarafından yapılandırılmıştır. CORBA istemci/sunucu etkileşimini kolaylaştırmak için gerekli olan ve istemci ve sunucu tarafların ikisinde birden koşan bir dağıtılmış yazılımdır, yani bir aracı birimdir.

OMG'nin Nesne Yönetim Mimarisi dört ana bileşenden oluşmaktadır :

- *Nesne İstek Aracısı (ORB), CORBA* nesne yolunu tanımlar.
- *CORBA Hizmetleri,* yolu (ORB) genişleten sistem-düzeyi nesne çalışma çerçeveleri tanımlar.

- *CORBA Kolaylıkları,* iş nesneleri tarafından doğrudan kullanılan yatay ve dikey uygulama çalışma çerçeveleri tanımlar.

- *Uygulama Nesneleri*, iş nesneleri ve uygulamalarıdır.

Bir ORB aşağıdaki parçalardan oluşur :

- *İstemci kütükleri veya statik çağrı arayüzü (SII)*, bir nesnenin OMG IDL tanımlı işlemlerine erişimi sunacaktır.

- *Dinamik çağrı arayüzü (DII),* nesne çağrılarının dinamik inşasına izin verir.

- *Gerçekleme iskeleti*, nesnenin her tipini gerçekleyen metotlara bir arayüzdür.

- *Dinamik iskelet arayüzü (DSI)*, nesne çağrılarının dinamik işlenmesine izin verir.

- Bir *nesne bağdaştırıcısı,* bir nesne gerçeklemesinin, ORB tarafından sağlanan hizmetlere erişiminin temel yoludur.

- *ORB arayüzü,* ORB'a doğrudan giden arayüzdür.

- *IDL derleyicisi,* arayüz tanımlamalarını yüksek-düzey dil yapılarına dönüştürür. Arayüz tanımlamaları OMG Arayüz Tanımlama Dili (OMG IDL) ile belirtilir.

- *Arayüz ambarı* koşma anında bulunabilir bir biçimde, IDL bilgisini temsil eden kalıcı nesneler sağlayan bir hizmettir.

CORBA Hizmetleri, IDL tarafından belirtilmiş arayüzlerle paketlenmiş, sistem düzeyi hizmetlerin koleksiyonlarıdır. OMG onbeş nesne hizmeti için standartlar yayınlamıştır : y*aşam çevrimi, kalıcılık (kalıcı durum)*, *adlandırma, olay, eşanlılık, bölünmez işlem, ilişki, dışarılama, sorgu, lisanslama, özellikler, zaman, güvenlik, tacir ve koleksiyon* hizmetleri.

Genel ORB Arası Protokol (GIOP), ORBlar arasındaki iletişim için bir standart iletim sentaksı ve bir mesaj biçimleri kümesi belirtir. Her CORBA 2.0 ORB'un desteklemek zorunda olduğu *İnternet ORB Arası Protokolü (IIOP),* GIOP mesajlarının TCP/IP bağlantıları kullanılarak nasıl karşılıklı değiştirileceğini belirtir. OMG ayrıca, *Ortama Özel ORB Arası Protokollerin (ESIOP)* açık sonlu bir kümesi için öngörümde bulunur.

Alanlar sistemlerin, genel karakteristiklere sahip olan bileşenlerin koleksiyonlarına ayrılmalarına izin verir. Alanlar arasında karşılıklı ortaklaşma, iki köprüleme mekanizması ile sağlanır :

- *Aracılı köprüleme*de tüm alanlar bir genel protokole köprülenir.

- *Aracısız köprüleme*de ise iki alan, mesajın gerekli parçalarını çeviren bir köprü üzerinden birbirleriyle doğrudan konuşur.

Tezimiz CORBA başarım analizinin yalnızca hizalama/geri hizalama yönüyle ilgilenmiştir. CORBA'nın yalnızca statik yönlerini kapsadık, dinamik yönlerini değil.

CORBA başarım ölçümüyle ilgili birçok yüksek lisans çalışması ve yayın bulunmaktadır. Ayrıca OMG tarafından, CORBA'nın başarım ölçümü için kurulmuş bir özel ilgi grubu da bulunmaktadır.

Başlığımızın da yansıttığı gibi hizalama/geri hizalama başarımını inceledik. Hizalama/geri hizalama, tiplendirilmiş veri nesnelerinin yüksek düzey gösterimlerden düşük düzey gösterimlere çevrilmesini (hizalama) ve tersi işlemi (geri hizalama) anlatır. Düşük düzey gösterimler, Genel Veri Gösterimi (CDR) kuralları izlenerek elde edilir.

Ölçüt takımımız üç oyuncudan oluşmaktadır :

- *Değer Hizmetlisi,* istemciden istekleri kabul eder ve tepkileri geri gönderir.
- *Zaman Hizmetlisi* zaman işlemleriyle ilgilenir.
- *İstemci,* sunucudan çağrılar yapar.

Ölçüt ortamımız aşağıdaki ölçüler ele alınarak inşa edilmiştir :

- Ölçütümüzü Dünya çapında en yaygın bulunabilen CORBA/Java ORBu üzerinde uyguladık. Yani Sun'ın Java IDL derleyicisini kullandık.
- Sunucularımız ve istemcimiz aynı makinede (yerel çağrılar) veya iki kesim farklı makinelerde (uzak çağrılar) konumlandırıldı.
- Metotlarımız tekyönlü veya çiftyönlü olarak tanımlandı.
- Hiçbir parametre almayan ve hiçbirşey geri döndürmeyen bir fonksiyonumuz var. Ayrıca parametrelerimiz istemciden sunucuya, sunucudan istemciye, ve her iki yönde akış yaptı.
- Tiplerimiz üç grup olarak sınıflandırıldı :
    - *İlkel Tipler* boolean, char, wchar, long, unsigned long, long long, unsigned long long, short, unsigned short, float, double, octet ve long double IDL tiplerini içermektedir.
    - *İnşa Edilmiş Tipler* yapıları, arayüzleri, birleşikleri ve söylenmişleri içermektedir.
    - *İçeren Tipler* dizileri, sıraları ve katarları içermektedir.

İnşa ettiğimiz ölçütü koşturduk. Genel sonuçlarımıza göre :

- Yerel çağrılar uzak çağrılardan daha hızlıdır.
- Tekyönlü uyarmalar ikiyönlü uyarmalardan daha hızlıdır. Ancak, tekyönlüler güvenilir değildir ve bazıları testi tamamlayamamıştır. Diğer taraftan, tüm ikiyönlü çağrılar ölçümleri başarıyla tamamlamıştır.
- Küçük boyutlu verilerin her akışı için hemen hemen aynı sonuçları aldık. Ancak daha büyük boyutlar için azalan sırada zamanlar şu şekildedir : iki yönde, istemciden sunucuya, sunucudan istemciye akış.

- İlkel tipler için düşük boyutta parametreler için yakın sonuçlar elde ettik. Yüksek boyutlu parametreler için sonuçlar tiplerin boyutlarına göre sıralanmıştır.

- İnşa Edilmiş Tipler için :

  - Yapılar için ilkellerle aynı sonuçları aldık.

  - Söylenmişler için unsigned long tipi ile aynı sonucu aldık.

  - Arayüzlerle parametreler yavaş geçmektedir.

  - Bir birleşik içinde octet geçirmek, bir double geçirmekten daha kısa sürmektedir.

- İçeren tipler için :

  - Sıralar ve dizilerle aynı sonuçları elde ettik.

  - Katarlarla karakter içerenler için aynı sonuçları elde ettik.

  - Geniş katarlarla geniş karakter içerenler için aynı sonuçları elde ettik.

Sonuç olarak, hemen hemen tüm statik IDL yapılarını bu tezde test ettik. Birçok ham verimiz var ve karşılaştırmalar bu veriler üzerinde yapılabilir. Yalnızca bunların bazıları için sonuçlar veriyoruz. İsteyen herkes verilerimizden gereksinim duyduğu sonuçları çıkarabilir.

MARSHALLING/DEMARSHALLING PERFORMANCE ANALYSIS OF SUN'S JAVA IDL BY USING STATIC INVOCATION INTERFACE

SUMMARY

Our thesis begin with an overview of the distributed systems. Distributed systems evolved by following the following eras :

- *Monolithic Systems* uses monolithic software written for mainframes.
- *Client/server Systems* are comprised of a server that provides services and a client that requests services of the server.
- *Multitier Client/server Systems* partitions the system into a user interface layer, a business rules layer, and a database access layer.
- *Distributed Systems* expose all functionality of the application as objects, each of which can use any of the services provided by other objects in the system or in other systems.

After that, we introduced the distributed objects and components. Distributed objects are extended objects that can reside anywhere on a network. Components are the smallest self-managing, independent, and useful parts of a system that works in multiple environments.

A business object consists of three main parts :

- *Business Logic Object (BLO)* defines how the object reacts to certain events.
- *Business Process Obect (BPO)* helps maintain the business logic for the entire system.
- *Presentation Objects* provide the user with a representation of the component.

The Common Object Request Broker Architecture (CORBA) is structured by Object Management Group (OMG) to allow integration of a wide variety of object systems. CORBA is a middleware which is a distributed software required to facilitate client/server interaction and runs on both the client and server ends of a transaction.

OMG's Object Management Architecture (OMA) is composed of four main elements:

- *Object Request Broker (ORB)* defines the CORBA object bus.
- *CORBA Services* define the system-level object frameworks that extend the bus.

- *CORBA Facilities* define horizontal and vertical application frameworks that are used directly by business objects

- *Application Objects* are the business objects and applications.

An ORB consisted of following parts :

- The *client stubs, or static invocation interface (SII),* will present access to the OMG IDL-defined operations on an object.

- The *dynamic invocation interface (DII)* allows the dynamic construction of object invocations.

- The *implementation skeleton* is an interface to the methods that implement each type of object.

- The *dynamic skeleton interface (DSI)* allows dynamic handling of object invocations

- An o*bject adapter* is the primary way that an object implementation accesses services provided by the ORB.

- The *ORB Interface* is the interface that goes directly to the ORB.

- The *IDL compiler* brings the interface definitions to high-level language constructs. Interface definitions are specified by *OMG Interface Definition Language (OMG IDL).*

- The *Interface Repository* is a service that provides persistent objects that represent the IDL information in a form available at run-time.

CORBA services are collections of system-level services packaged with IDL-specified interfaces. OMG has published standards for fifteen object services : *life cycle, persistence (persistent state), naming, event, concurrency, transaction, relationship, externalization, query, licensing, properties, time, security, trader and collection services.*

The *General Inter-ORB Protocol (GIOP)* specifies a standard transfer syntax and a set of message formats for communications between ORBs. *Internet Inter-ORB Protocol (IIOP),* to which every CORBA 2.0-compliant ORB must supply, specifies how GIOP messages are exchanged using TCP/IP connections. OMG also makes provision for an open-ended set of *Environment-Specific Inter-ORB Protocols (ESIOPs).*

Domains allow partitioning of systems into collections of components which have some characteristic in common. Interoperability between domains is achieved by using two bridging mechanisms :

- In *mediated bridging* all domains bridge to a single common protocol

- In *immediate bridging* two domains talk directly to each other over a single bridge that translates whatever parts of the message require it.

Our thesis only concerned the marshalling/demarshalling aspects of performance analysis of CORBA. We only covered the static aspects of CORBA and not the dynamic aspects.

There are a lot of master studies and publications related with performance analysis of CORBA. Also there is a special interest group founded by OMG on the benchmarking of CORBA.

As our title reflects, we analyzed the marshalling/demarshalling performance. The marshalling/demarshalling refers to the transformations of typed data objects from higher-level representations to lower-level representations (marshalling) and vice versa (demarshalling). The low-level representations are created by following the rules of Common Data Representation (CDR).

We have three players constitutes our benchmarking team :

- *Value Server* accepts requests from the client and sends responses back.
- *Time Server* handles the time operations.
- *Client* makes calls from server.

Our benchmarking environment is constructed by considering the following criterias :

- We applied the benchmark to most commonly available CORBA/Java ORB worldwide, Sun's Java IDL compiler.
- We have the servers and the client at the same computer (local calls) or two sides are located at different computers (remote calls)
- Our methods are defined as oneway and twoway.
- We have a function which takes no parameters and returns nothing. Also our parameters flow from client to server, from server to client and in both directions.
- Our types are classified into three groups :
    - *Primitive types* include the IDL types *boolean, char, wchar, long, unsigned long, long long, unsigned long long, short, unsigned short, float, double, octet and long double.*
    - *Constructed types* include structs, interfaces, unions and enums.
    - *Container types* include arrays, sequences and strings.

We run the benchmark constructed. Our general results show that :

- Local calls are faster than remote calls for big-sized data. For small-sized data, remote calls perform better.
- Oneway invocations are faster than twoway invocations. But oneways are unreliable and some of them could not completed the test. On the other hand, all the twoway calls successfully completed the measurements.

- We see the nearly same results for small sized data for all the flows of data. But for the larger sizes the descending order is from server to client and client to server back, from client to server and from server to client.

- For the primitive types we have the close results for small-sized parameters and results are ordered with sizes of types for larg-sized parameters.

- For constructed types :

    - We have the same results for primitives with structs for small sizes. For big sizes, structs perform worse.

    - We have the same results for unsigned long with enums.

    - We have slow passing of parameters with interfaces.

    - Passing an octet within a union takes less time than passing a double.

- For the container types :

    - We have the same results with sequences and arrays.

    - We have the same results for strings with char containers.

    - We have the same results for wstrings with wchar containers.

As a conclusion we tested nearly all static IDL constructs in this thesis. We have a bulk of raw data, and comparisons can be made on these data. We only give conclusions for some of them. Whoever wants can deduce the conclusions he/she needs from our data.

1. HISTORY OF DISTRIBUTED SYSTEMS

Here we will have an overview of distributed systems. This chapter is mainly taken from [1].

1.1 Monolithic Systems

We can say that distributed systems began with mainframes. Mainframes are managed centrally and software systems written for mainframes were often monolithic, i.e, the user interface, business logic, and data access functionality were all contained in one large application. A typical monolithic application architecture is illustrated in Figure 1.1 [1]**.**

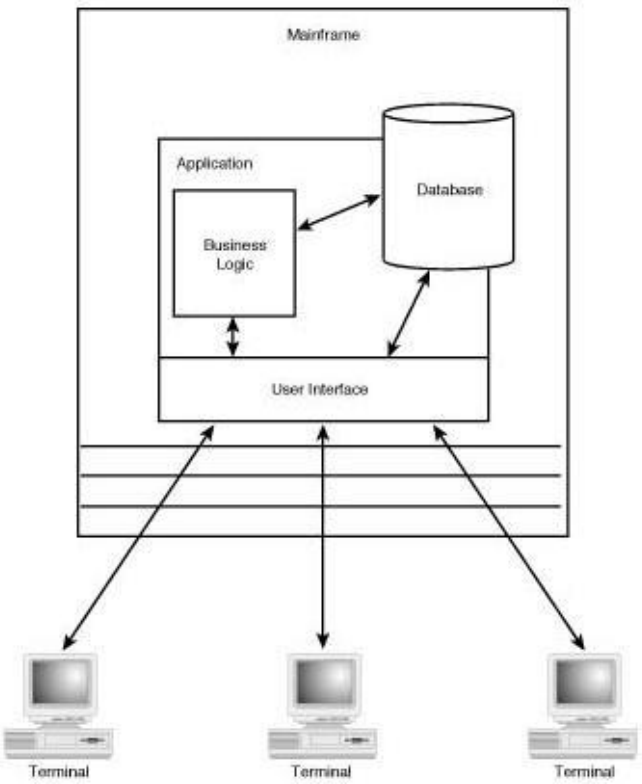


Figure 1.1 : Typical monolithic application architecture.

1.2 The Client/Server Model

After the monolithic systems we see the client/server architecture. Client/server computing systems are comprised of two logical parts :

- a *server* that provides services.
- a *client* that requests services of the server (see Figure 1.2).

Together, the two form a complete computing system with a distinct division of responsibility.

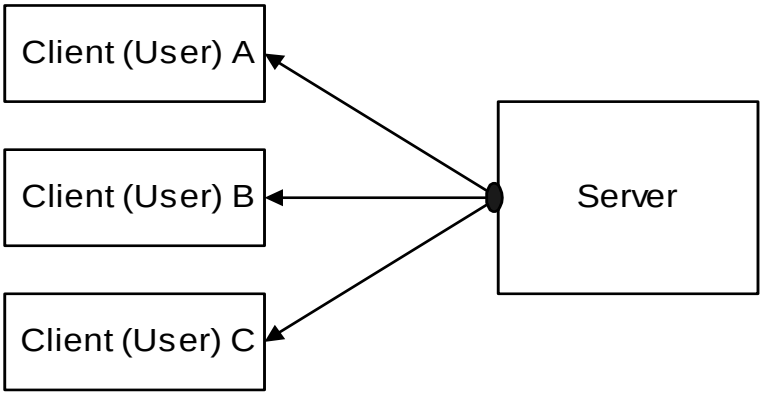


Figure 1.2 : A traditional client/server system. Clients request services of the server independently but use the same interface.

Client/server computing has gained popularity in the recent years due to the proliferation of low-cost hardware and the fact that a model relying on monolithic applications fails when the number of users accessing a system grows too high or when too many features are integrated into a single system.

We defined the client as the component of the client/server architecture which requests services from servers. In addition, clients can also offer services to other clients. That is, a client can act as a server to other clients.

According to the server's behaviour, we can group server side of the client/server architecture into two models :

- Pull Server Model
- Push Server Model

Traditional servers are entities that passively await requests from clients and then act on them. This model is named *pull server model*. But, servers can actively search out changes in the state of clients and take appropriate action. This model is called *push server model*.

Most client/server systems are flexible with regard to the distribution of authority, responsibility, and intelligence. A part of a system with a disproportionate amount of functionality is called *fat*; a *thin* portion of a system is a part with less responsibility delegated to it [2]. The server portion of a client/server system almost always holds the data, and the client is nearly always responsible for the user interface; the shifting of application logic constitutes the distinction between fat clients and fat servers (see Figure 1.3).

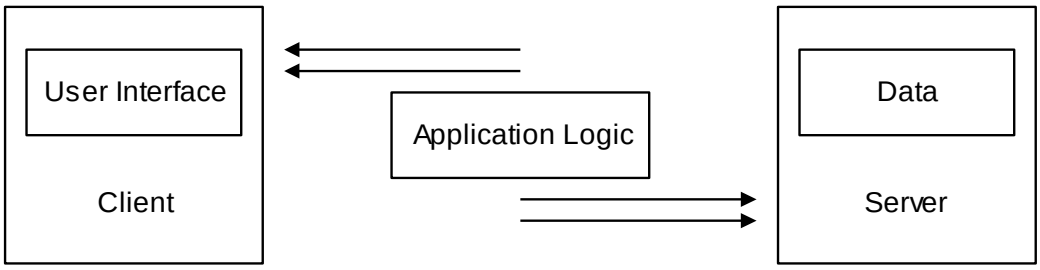


Figure 1.3 : Since the distribution of the user interface and data is fixed, the placement of the application logic is what distinguishes fat-client from fat-server systems.

The fat server model is often used to ensure greater compatibility between clients and servers : the more work the server does, the less dependent it is on the client. The fat client model can be used at the expense of universal compatibility [3].

1.3 Multitier (N-Tiered) Client/Server

The canonical client/server model assumes exactly two discrete participants in the system. This is called a *two-tier system*; the application logic must be in the client or the server, or shared between the two. It is also possible to have the application logic reside separately from the user interface and the data (in other words, to partition the system into three logical tiers : the user interface layer, the business rules layer, and the database access layer, see Figure 1.4 [1]) turning the system into a *three-tier system*. In an idealized three-tier system, all application logic resides in a layer separate from the user interface and data. Decoupling the application logic from the data allows data from multiple sources to be used in a single transaction without a breakdown in the client/server model.

1.4 Distributed Systems

Rather than differentiate between business logic and data access, the distributed system model simply exposes all functionality of the application as objects, each of which can use any of the services provided by other objects in the system, or even objects in other systems. The architecture can also blur the distinction between client and server because the client components can also create objects that behave in server-like roles.

The distributed system architecture achieves its flexibility by enforcing the definition of specific component interfaces. The interface of a component specifies to other components what services are offered by that component and how they are used. As long as the interface of a component remains constant, that component's implementation can change dramatically without affecting other components.

Distributed systems are really multitier client/server systems in which the number of distinct clients and servers is potentially large. One important difference is that distributed systems generally provide additional services, such as directory services, which allow various components of the application to be located by others.

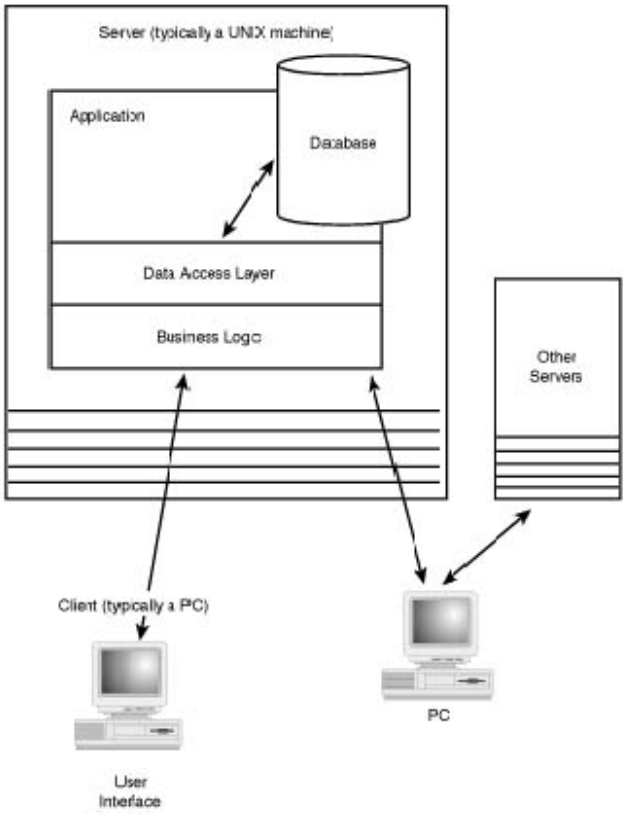


Figure 1.4 : Three-tier client/server architecture.

2. DISTRIBUTED OBJECTS AND COMPONENTS

Distributed systems are built on the object oriented approach. Classical objects do not suffice for distributed systems. They are replaced with distributed objects and components. In this section we will present you with distributed objects and components. This section is mainly taken from [3].

2.1 From Objects to Distributed Objects

As computing systems evolved, the paradigm of algorithmic computation was replaced by the use of interacting objects. Classical objects can be viewed as self-contained entities that encapsulate data, and a set of operations that act on that data.

Distributed objects are extended objects that can reside anywhere on a network and continue to exist as physical standalone entities while remaining accessible remotely by other objects. Robust distributed object systems allow objects written in different languages and compiled by different compilers to communicate seamlessly via standardized messaging protocols embodied by middleware.

2.2 Benefits of Distributed Objects

Distributed objects allow us to construct scaleable client/server systems by providing modularized software that features interchangeable parts.

Self-managing distributed objects take responsibility for their own resources, work across networks, and interact with other objects. These capabilities are frequently given to objects through a distributed object framework that provides middleware to regulate the necessary inter-object communications and provides a resource pool for each object that is deleted when that object ceases to exist.

Self-managing objects are used easily by other objects since no management burdens are imposed on the client object; it receives object services at no cost. Objects crafted to these specifications rely on a solid event model that allows objects to broadcast specific messages and generate certain events. These events are listened for by other objects, which then take action based on them. Each listening object responds to a given event in its own manner. By using object-

oriented techniques such as polymorphism, closely related objects react differently to the same event. These capabilities simplify the programming of complex client/server systems.

Objects can generate events to notify other objects that an action should take place. In this sense, events can be viewed as synchronization objects that allow one thread of execution to notify another thread that something has happened (see Figure 2.1). Using this model, an event can notify a component that it should take a certain action. An object that can listen for events provides a more robust framework for interaction between objects than a model that forces objects to wait for the next instruction.

Because of the strict encapsulation that objects provide, distributed objects are a fundamentally sound unit from which to build client/server applications when separation of data is important. Cooperating objects form the logic portion of most substantial client/server systems because of the rich interaction services they offer [4] [5].

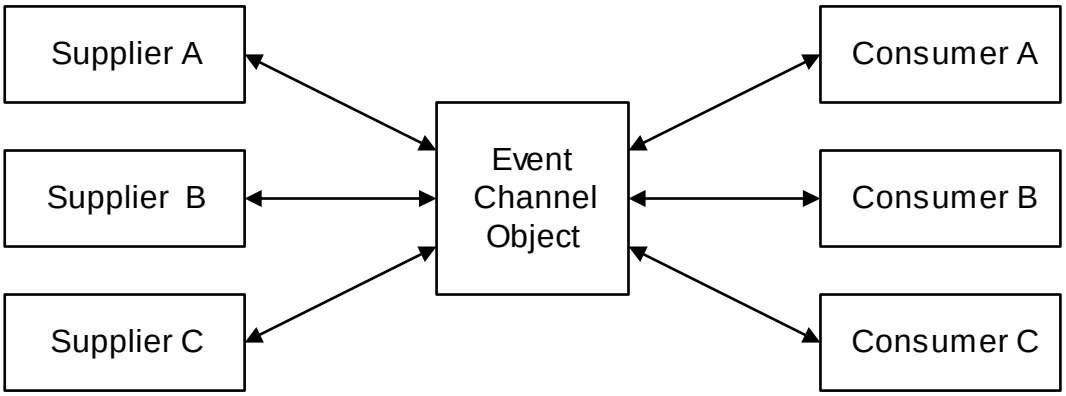


Figure 2.1 : A sample event management system. Depicted is CORBA's Event Service.

Since distributed objects allow applications to be split up into lightweight pieces that can be executed on separate machines, less powerful machines can run demanding applications.

2.3 Components

Components are the smallest self-managing, independent, and useful parts of a system that works in multiple environments. Components promise rapid application development and a high degree of customizability for end users, leading to fine-tuned applications that are relatively inexpensive to develop and easy to learn.

Components are most often distributed objects incorporating advanced self-management features. Components may contain multiple distributed or local objects, and they are often used to centralize and secure an operation.

The interface of a component should be the primary concern of its developer. Since components are designed for use in a variety of systems and need to provide reliable services regardless of context, developers attempting to use a component must be able to identify clearly the function of a component and the means of invoking this behavior.

2.4 Client/Server Using Distributed Objects

Tht client/server computing using distributed objects is depicted in Figure 2.2.

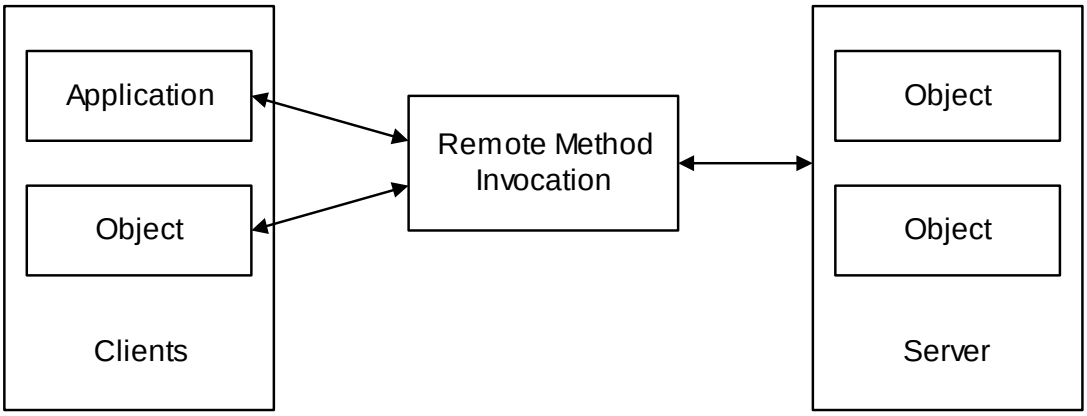


Figure 2.2 : Client/server computing using distributed objects. Communication between components (denoted by arrows) is facilitated through ORBs (which have been omitted for clarity)

2.5 Business Objects

Business objects are self-managing components used to represent key objects or processes in a real-life system. Business objects are shippable products that usually have a user interface and the ability to cooperate with other objects to meet a certain user need. Business objects allow application-independent concepts to be described at a high level, minimizing the importance of languages, tools, and application-level concepts. Business objects represent a major breakthrough in the modeling of business events since they can describe both a portion of a real-world business system and the executing piece of the information system supporting that portion of the business [2] [6].

Like other components, business objects should support late binding so they can be interchanged easily and interact immediately with existing components; they should also support standard component features such as event handling and state maintenance.

The *Business Object Model Special Interest Group (BOMSIG)* has proposed a standard for business objects. The standard calls for each business object to be composed of three types of cooperating objects (see Figure 2.3 [6]).

- *Business Logic Object (BLO)* defines how the object reacts to certain events; it is responsible for the business logic of the component as well as for storing the relevant business data.

- *Business Process Obect (BPO)* helps maintain the business logic for the entire system. The primary difference between a BPO and a BLO is the logical lifetime of the unit of logic : BPOs traditionally handle long-lived processes or processes related to the system as a whole.

- *Presentation Objects* provide the user with a representation of the component, usually but not necessarily visual.

A normal business object is likely to have multiple Presentation Objects, but usually has one BLO and BPO. Because these three objects are managed by one object, collaborating components see only one object that provides the aggregate services of its constituent objects.

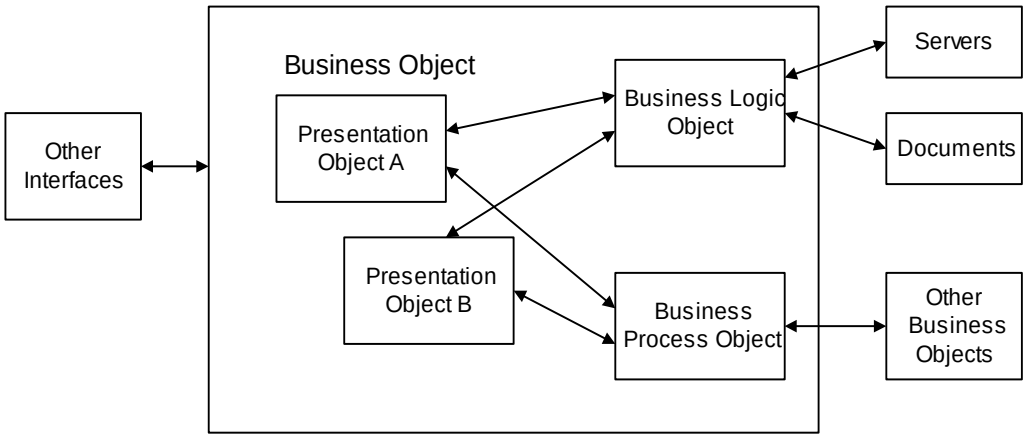


Figure 2.3 : The parts of a business object and their communication with other system objects

This three-object construction can be viewed as a three tier client/server system (see Figure 2.4) :

- *Tier 1 :* Visual aspects of a system, usually handled by a client system.
- *Tier 2 :* Data for the object and the application logic required to meaningfully act on it.
- *Tier 3 :* Data and application logic required to integrate the business object with other business objects and existing systems, such as legacy servers or databases.

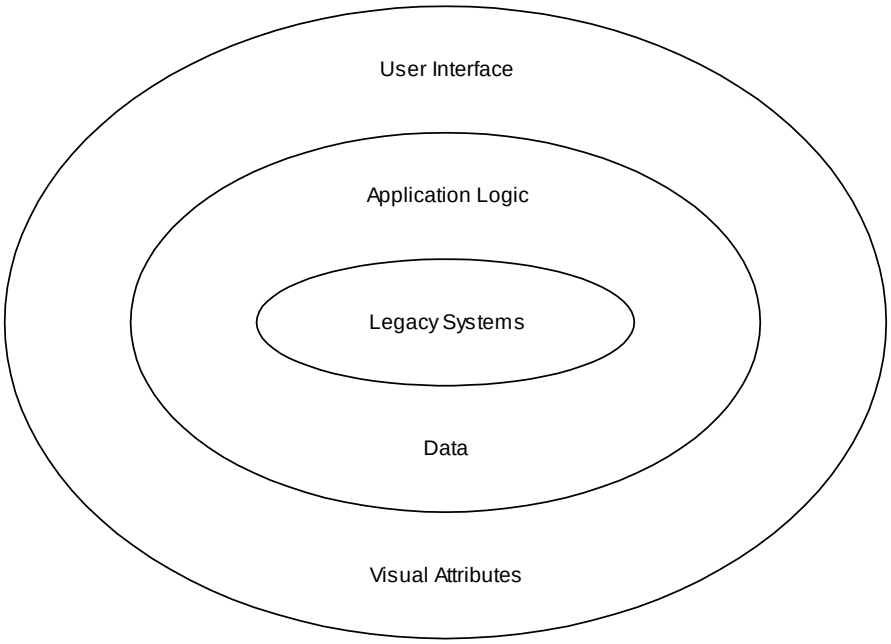


Figure 2.4 : Three tiers in a business object.

The middle tier plays the largest role in this organizational scheme. Tier-two objects communicate directly with the tier-one objects to provide feedback to the user; they also provide the logic for the entire business object. Furthermore, tier-two objects communicate with multiple data repositories (tier three) and collaborate with other business objects to assist them provide services. This model separates the client from data for which it is not logically responsible. By channeling all requests for information through the tier-two servers, major changes (such as the implementation of a new database system) remain completely transparent to the user. If ORBs are used for communication between the clients and the tier-two objects, robust system services such as load balancing and event exchanges are implemented easily and applications remain scaleable.

3. CORBA OVERVIEW

This chapter is mainly taken from the Object Management Group's (OMG) formal documentation describing CORBA [7].

The Common Object Request Broker Architecture (CORBA) is structured to allow integration of a wide variety of object systems. CORBA is a middleware, so first we define what is a middleware and then take a general look at CORBA.

3.1 Middleware

The distributed software required to facilitate client/server interaction is referred to as middleware. *Middle* refers to its place in a software abstraction hierarchy above transport protocols, but below clients and servers written in a high-level programming language [8]. Transparent access to non-local services and resources distributed across a network is usually provided through middleware, which serves as a framework for communication between the client and server portions of a system. Middleware can be thought of as the networking between the components of a client/server system; it is what allows the various components to communicate in a structured manner. Middleware is defined to include the Application Programmer Interfaces (APIs) used by clients to request a service from a server, the physical transmission of the request to the network (or the communication of the service request to a local server), and the resulting transmission of data for the client back to the network. Middleware is run on both the client and server ends of a transaction [3].

3.2 OMG's Object Management Architecture (OMA)

In the fall of 1990, the OMG first published the *Object Management Architecture Guide (OMA Guide)*. After that it have gone under some changes and still goes. Figure 3.1 [10] shows the four main elements of the architecture [9] :

1. *Object Request Broker (ORB)* defines the CORBA object bus.

2. *CORBA Services (Common Object Services)* define the system-level object frameworks that extend the bus

3. *CORBA Facilities (Common Facilities)* define horizontal and vertical application frameworks that are used directly by business objects

4. *Application Objects* are the business objects and applications, they are the ultimate consumers of the CORBA infrastructure.

This section provides a top-level view of the elements that make up the CORBA infrastructure.

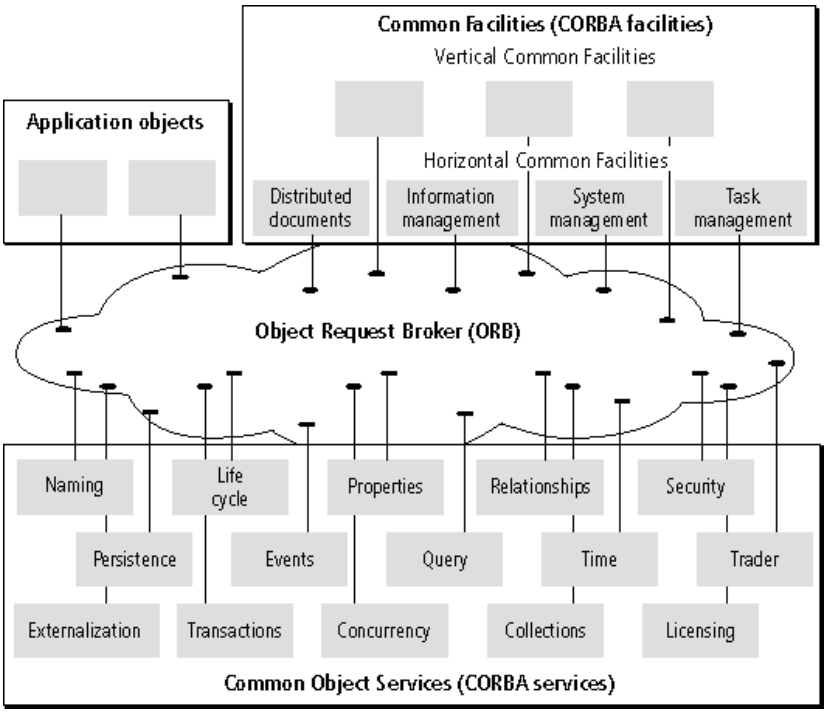


Figure 3.1 : Object Management Architecture

3.3 Object Request Broker (ORB)

The *Objet Request Broker (ORB)* is the object bus. It lets objects transparently make requests to -and receive responses from- other objects located locally or remotely.

A CORBA ORB provides a wide variety of distributed middleware services. Every CORBA ORB provides [9] :

- Static and dynamic method invocations
- High-level language bindings

- Local/remote transparency
- Polymorphic messaging
- Coexistence with legacy systems

and hides [11] [12] :

- Object location (local/remote transparency)
- Object implementation (high-level language bindings used for object implementation)
- Object execution state (If object is not active at the time its method is invoked, ORB activates it)
- Object communication mechanisms (the communication can be done via TCP/IP, shared memory, local method calls etc)

Figure 3.2 [11] shows the structure of an individual Object Request Broker (ORB) and its interactions with the objects (client and servant). The arrows indicate whether the ORB is called or performs an up-call across the interface.

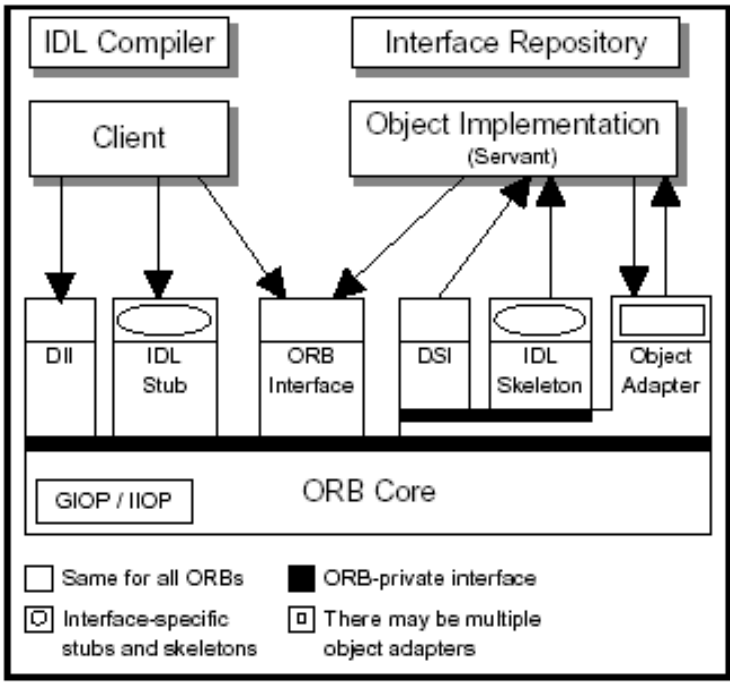


Figure 3.2 : The structure of object request interfaces

Now, we can go over the parts shown in Figure 3.2 and explain each of these.

3.3.1 clients

A client of an object has access to an *object reference* for the object, and invokes operations on the object. A client knows only the logical structure of the object according to its interface.

Clients generally see objects and ORB interfaces through the perspective of a *language mapping*, bringing the ORB right up to the programmer's level. Clients are maximally portable and should be able to work without source changes on any ORB that supports the desired language mapping with any object instance that implements the desired interface.

3.3.1.1 structure of a client

A client of an object has an object reference that refers to that object. An *object reference* is a token that may be invoked or passed as a parameter to an invocation on a different object. Invocation of an object involves specifying the object to be invoked, the operation to be performed, and parameters to be given to the operation or returned from it.

The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB can not complete the invocation, an exception response is provided.

Clients access object-type-specific stubs as library routines in their program (see Figure 3.3). The client program thus sees routines callable in the normal way in its programming language. All implementations will provide a language specific data type to use to refer to objects. The client then passes that object reference to the stub routines to initiate an invocation. The stubs have access to the object reference representation and interact with the ORB to perform the invocation.

Clients most commonly obtain object references by receiving them as output parameters from invocations on other objects for which they have references. An object reference can also be converted to a string that can be stored in files or preserved or communicated by different means and subsequently turned back into an object reference by the ORB that produced the string.

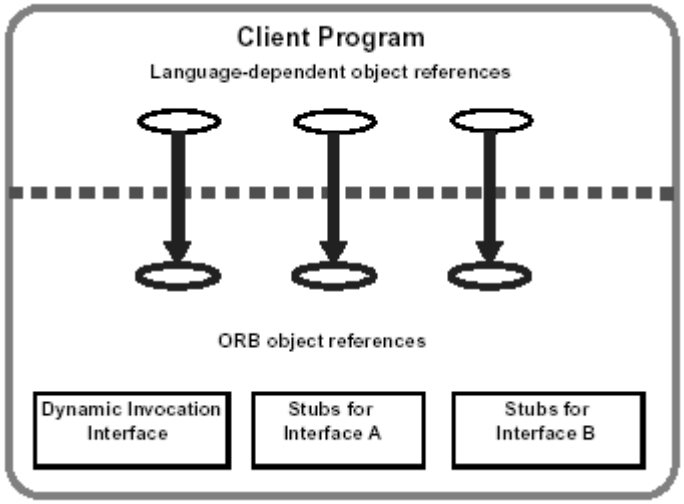


Figure 3.3 : The structure of a typical client

3.3.2 object implementations

An object implementation provides the semantics of the object, usually by defining data for the object instance and code for the object's methods.

Often the implementation will use other objects or additional software to implement the behavior of the object.

Generally, object implementations do not depend on the ORB or how the client invokes the object. Object implementations may select interfaces to ORB-dependent services by the choice of Object Adapter.

3.3.2.1 structure of an object implementation

An object implementation provides the actual state and behavior of an object. The object implementation can be structured in a variety of ways. Besides defining the methods for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other objects or nonobject facilities to make the object state persistent, to control access to the object, as well as to implement the methods.

The object implementation (see Figure 3.4) interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. It primarily does this via access to an Object Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation.

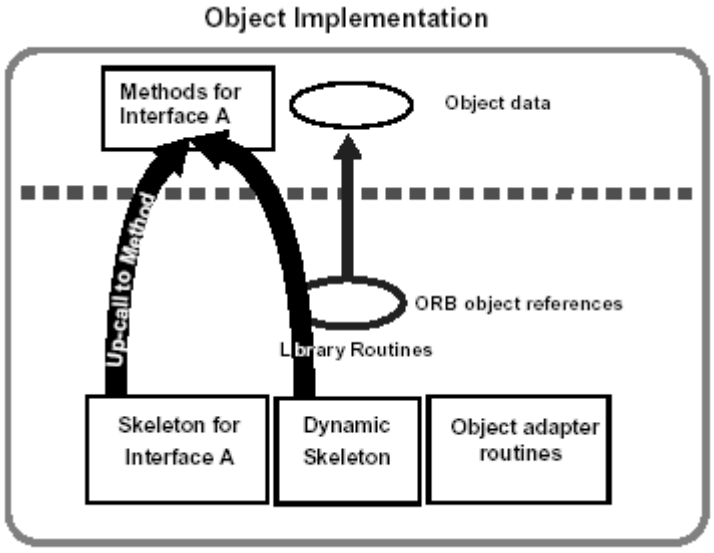


Figure 3.4 : The structure of a typical object implementation

Because of the range of possible object implementations, it is difficult to be definitive about how an object implementation is structured.

3.3.2.2 object references

An *Object Reference (OR)* is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and thus are insulated from the actual representation of them. Two ORB implementations may differ in their choice of Object Reference representations.

There is a distinguished object reference guaranteed to be different from all object references, that denotes no object.

An *Interoperable Object Reference (IOR)* is the information needed to specify an object accross ORBs. This reference can be used when ORBs interoperate. The structure of an IOR includes repository ID, protocol and address details and object key [13].

3.3.3 IDL compiler

As mentioned in section 3.3.1, clients see objects and ORB interfaces through the perspective of a language mapping. Bringing the interface definitions to high-level language constructs is done by the IDL compiler.

An *IDL compiler* transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language like Java [14]. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [15].

3.3.3.1 OMG interface definition language (IDL)

The *OMG Interface Definition Language (OMG IDL)* defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations.

IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

3.3.3.2 mapping of OMG IDL to programming languages

Different object-oriented or non-object-oriented programming languages may prefer to access CORBA objects in different ways. For object-oriented languages, it may be desirable to see CORBA objects as programming language objects. Even for nonobject-oriented languages, it is a good idea to hide the exact ORB representation of the object reference, method names, etc. A particular mapping of OMG IDL to a programming language should be the same for all ORB implementations.

3.3.4 client stubs

The client stubs will present access to the OMG IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with OMG IDL and the language mapping for the particular programming language.

3.3.5 dynamic invocation interface (DII)

An interface is also available that allows the dynamic construction of object invocations, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the

operation to be performed, and the set of parameters for the operation through a call or sequence of calls.

The client code must supply information about the operation to be performed and the types of the parameters being passed.

The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

3.3.6 implementation skeleton

For a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

3.3.7 dynamic skeleton interface (DSI)

An interface is available, which allows dynamic handling of object invocations. That is, rather than being accessed through a skeleton that is specific to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface. Purely static knowledge of those parameters may be used, or dynamic knowledge may be also used, to determine the parameters.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the values of any output parameters, or an exception, to the ORB after performing the operation.

3.3.8 object adapters

An O*bject Adapter* is the primary way that an object implementation accesses services provided by the ORB. There are expected to be a few object adapters that will be widely available, with interfaces that are appropriate for specific kinds of objects. Services provided by the ORB through an Object Adapter often include generation and interpretation of object references, method invocation, security of

interactions, object and implementation activation and deactivation, mapping object references to implementations, registration of implementations.

3.3.8.1 structure of an object adapter

An object adapter (see Figure 3.5) is the primary means for an object implementation to access ORB services such as object reference generation. Object adapters are responsible for the following functions :

- Generation and interpretation of object references
- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- Registration of implementations

These functions are performed using the ORB Core and any additional components necessary. Often, an object adapter will maintain its own state to accomplish its tasks. It may be possible for a particular object adapter to delegate one or more of its responsibilities to the Core upon which it is constructed.

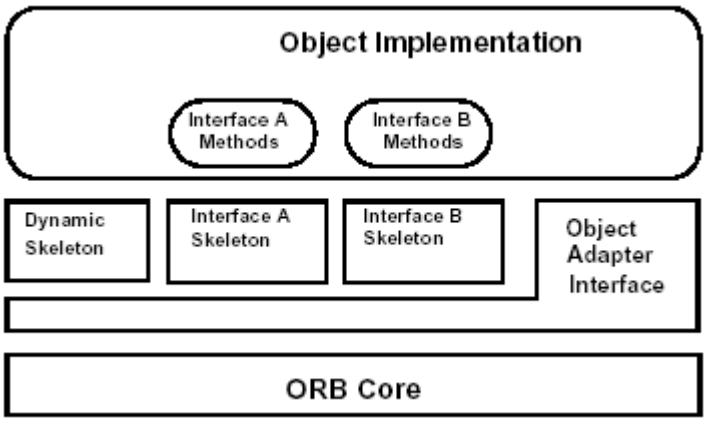


Figure 3.5 : The structure of a typical object adapter

As shown in Figure 3.5, the Object Adapter is implicitly involved in invocation of the methods, although the direct interface is through the skeletons.

3.3.8.2 CORBA required object adapter

There are a variety of possible object adapters; however, since the object adapter interface is something that object implementations depend on, it is desirable that there be as few as practical. Most object adapters are designed to cover a range of object implementations, so only when an implementation requires radically different services or interfaces should a new object adapter be considered.

CORBA used to specify a *Basic Object Adapter (BOA)* that can be used for most ORB objects with conventional implementations. At June 1997, OMG published the specifications for the *Portable Object Adapter (POA)* [16]. POA allows developers to construct CORBA server applications that are portable between heterogeneous ORB implementations [17].

3.3.9 ORB interface

The *ORB Interface* is the interface that goes directly to the ORB, which is the *same for all ORBs* and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects.

3.3.10 interface repository (IR)

The *Interface Repository* is a service that provides persistent objects that represent the IDL information in a form available at run-time. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it.

3.4 CORBA Services

CORBA services are collections of system-level services packaged with IDL-specified interfaces. You can think of object services as augmenting and complementing the functionality of the ORB. OMG has published standards for fifteen object services [9] [18] :

- The *Life Cycle Service* defines services and conventions for creating, deleting, copying and moving objects [19].
- The *Persistence (Persistent State) Service* defines interfaces which present persistent information as storage objects stored in storage homes. Storage homes are themselves stored in datastores, an entity that manages data, for example a database, a set of files, a schema in a relational database [20].
- The *Naming Service* provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use (make requests of) [21].
- The *Event Service* defines two roles for objects : the supplier role and the consumer role. *Suppliers* produce event data and *consumers* process event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests [22].
- The *Concurrency Service* mediates concurrent access to an object such that the consistency of the object is not compromised when accessed by concurrently executing computations [23].
- The *Transaction Service* provides interfaces that combine the transaction paradigm, essential to developing reliable distributed applications, and the object paradigm, key to productivity and quality in application development, together to address the business problems of commercial transaction processing [24].
- The *Relationship Service* allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two kinds of objects: relationships and roles [25].
- The *Externalization Service* defines protocols and conventions for externalizing (recording the object's state in a stream of data) and internalizing objects [26].
- The *Query Service* provides query operations on collections of objects [27].
- The *Licensing Service* defines the interfaces that support management of software licenses [28].

- The *Properties Service* provides the ability to dynamically associate named values with objects outside the static IDL-type system [29].

- The *Time Service* enables a user to obtain current time together with an error estimate associated with it [30].

- The *Security Service* defines a security reference model that provides the overall framework for CORBA security [31].

- The *Trader Service* facilitates the offering and the discovery of instances of services of particular types [32].

- The *Collection Service* provides a uniform way to create and manipulate the most common collections generically [33].

3.5 CORBA Facilities

CORBA facilities are collections of IDL-defined frameworks that provide services of direct use to application objects. The two categories of common facilities—*horizontal* and *vertical*—define rules of engagement that business components need to effectively collaborate.

3.6 Applicatıon Objects

Application objects are business objects which we described at chapter 2.

3.7 CORBA Interoperability

The *General Inter-ORB Protocol (GIOP)* specifies a standard transfer syntax and a set of message formats for communications between ORBs. The *Internet Inter-ORB Protocol (IIOP)* specifies how GIOP messages are exchanged using TCP/IP connections. Every CORBA 2.0-compliant ORB speaks the mandatory IIOP. OMG also makes provision for an open-ended set of Environment-Specific Inter-ORB Protocols (ESIOPs) [7]. The IIOP's standing in networking in a OSI-like layer model is shown in Figure 3.6 [34].

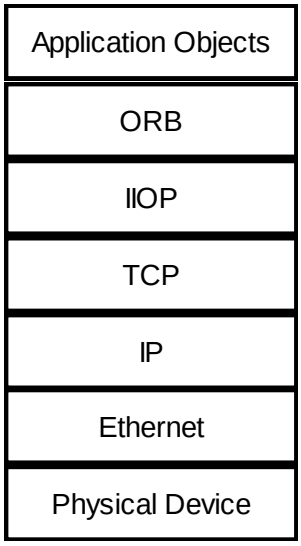


Figure 3.6 : IIOP's place in networking

Figure 3.7 [35] shows how the CORBA ORB-to-ORB communication works : An invocation from a client of ORB 1 passes through its IDL stub into the ORB core. The ORB examines the object reference and if the implementation is local, the ORB passes the invocation through the skeleton to the object for servicing. If the implementation is remote, ORB 1 passes the invocation across the communication pathway to ORB 2, which routes it to the object. The object implementation has no way of knowing whether the client is local or remote.

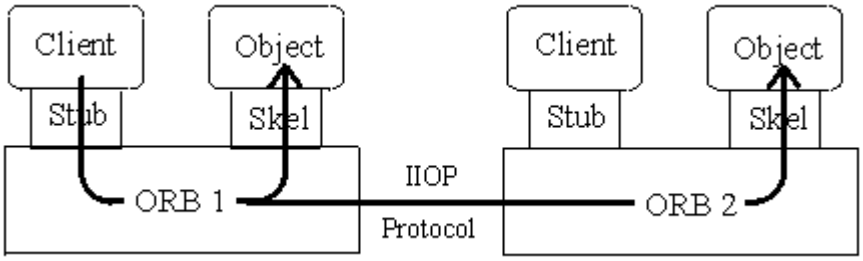


Figure 3.7 : Interoperability uses ORB-to-ORB communication.

For the inter-ORB invocations objects must have the Interoperable Object References (IORs) [36].

3.7.1 CORBA domains

Domains allow partitioning of systems into collections of components which have some characteristic in common. Interoperability between domains is only possible if there is a well-defined mapping between the behaviors of the domains being joined.

When an interaction takes place across a domain boundary, a mapping mechanism, or *bridge*, is required to transform relevant elements of the interaction as they traverse the boundary. There are essentially two approaches to achieving this :

- In *mediated bridging* all domains bridge to a single common protocol
- In *immediate bridging* two domains talk directly to each other over a single bridge that translates whatever parts of the message require it.

4. THE THESIS

Our thesis is about the performance analysis of CORBA. We are only concerned about the marshalling/demarshalling aspects of performance analysis.

4.1 The Goal of The Thesis

Our main goal in this study is to analyze the marshalling/demarshalling performance of a CORBA/Java ORB using *Static Invocation Interface (SII)*. That is, we have not covered the dynamic aspects of CORBA including DII (Dynamic Invocation Interface) with *Request*s created, populated and sent at run-time and dynamic types like *any* which also are created and populated at run-time. Indeed, we have tried to cover all static types of CORBA and IDL-specific features.

4.2 Related Work

There is a number of master studies on this subject. Buble compares three common C++ implementations of CORBA : OmniORB, ORBacus and Orbix [37]. Gopinath analyzes the performances of Real-Time CORBA endsystems by using omniORB [38]. Karlsson compares two C++ ORBs : Orbix and TAO [11].

There are also some related publications. Gokhale and Schmidt analyze the performance of DII and DSI over ATM networks by using Orbix and ORBeline [39]. They, in another paper, optimize the sunSoft IIOP and give measurements before and after applying their optimizations [40]. In another work, they give measurement results for four demultiplexing strategies by using TAO [41]. They also give latency results for two conventional ORBs, Orbix and VisiBroker, and then give their improved results for TAO [42].

Hirano, Yasu and Igarashi compare their lightweight ORB, HORB, with Voyager, VisiBroker and OrbixWeb [43] . Brose compares his ORB jacORB, with VisiBroker, Orbacus and RMI [44].

OMG also has its special interest group for benchmarking and this group published a white paper on benchmarking [45].

4.3 Marshalling And Demarshalling

Marshalling/demarshalling refers to the transformations of typed data objects from higher-level representations to lower-level representations (marshalling) and vice versa (demarshalling) [39].

Marshalling and demarshalling operations take place in user space and are often time consuming [46] [39].

4.3.1 common data representation (CDR)

Low-level representations are created by following the rules of Common Data Representation (CDR). CDR is a transfer syntax, mapping from data types defined in OMG IDL to a bicanonical, low-level representation for transfer between agents. CDR has the following features:

- Variable byte ordering - Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.

- Aligned primitive types - Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.

- Complete OMG IDL Mapping - CDR describes representations for all OMG IDL data types [7].

The CDR transfer syntaxes of IDL types are explained in Appendix.

4.4 The Benchmark's Players

We have three players constituting our benchmarking team :

- *Value Server* implements our benchmark's IDL definitions. It accepts requests from the client and sends responses back to it.
- *Time Server* handles the time operations. We have taken 25 samples of time taken by making 100 calls with each criteria. Time server saves each of these 25 time values. After that it computes the average of these 25 time values and stores them with their average in a file.
- *Client* makes calls from server.

Figure 4.1 shows these three players at work.

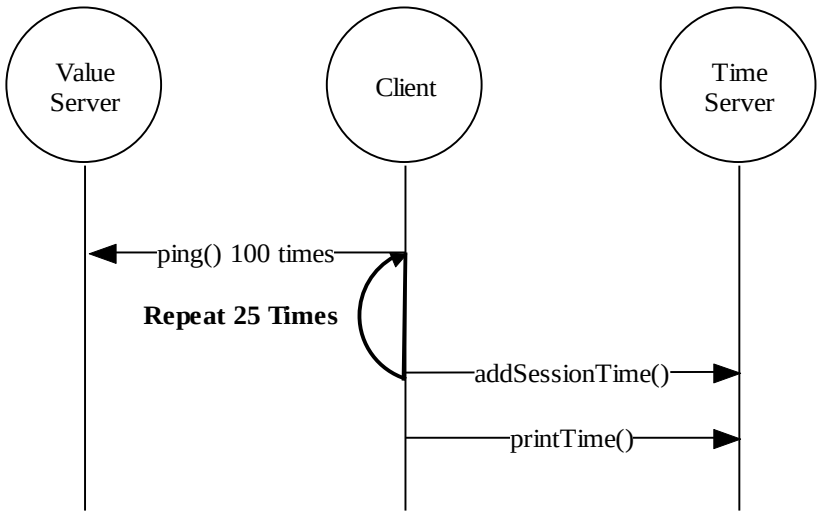


Figure 4.1 : Objects used in our benchmarking framework.

4.5 Structure of The Benchmark

The structure of our benchmark in Backus-Naur Format (BNF) is :

<benchmark> ::= "JDK1.3_02/" (<local> | <remote>)

<local> ::= "Local/" (<oneway> | <twoway>)

<remote> ::= "Remote/" (<oneway> | <twoway>)

<oneway> ::= "Oneway/" (<invoke> | <only_send>)

<twoway> ::= "Twoway/" (<invoke> | <only_send> | <only_get> | <send_get>)

<invoke> ::= “Invoke”

<only_send> ::= “OnlySend/” (<primitve> | <constructed> | <container>)

<only_get> ::= “OnlyGet/” (<primitve> | <constructed> | <container>)

<send_get> ::= “SendGet/” (<primitve> | <constructed> | <container>)

<primitive> ::= “Primitive/” (“Boolean” | ”Char” | “WChar” | “Double”

| “LongDouble” | “Float” | “Long” | “UnsignedLong” | “LongLong”

| “UnsignedLongLong” | “Octet” | “Short” | “UnsignedShort”)

<constructed> ::= “Constructed/” (<struct> | <interface> | <union> | <enum>)

<struct> ::= “Struct/” (<primitive> | “AllPrimitives”)

<interface> ::= “Interface/” (<primitive> | “AllPrimitives” | “Empty”)

<union> ::= “Union/” (“AllPrimitivesOctet” | “AllPrimitivesDouble”)

<enum> ::= “Enum/AllPrimitives”

<container> ::= “Container/” (<array> | <sequence> | <strings>)

<strings> ::= “Strings/” (<string> | <wstring>)

<string> ::= “String/”

(“String1” | “String10” | “String100” | “String1000” | “String10000”)

<wstring> ::= “WString/” (“WString1” | “WString10” | “WString100”

| “WString1000” | “WString10000”)

<array> ::= “Array/” (<container_primitive> | <container_constructed>)

<sequence> ::= “Sequence/” (< container_primitive> | <container_constructed>)

<container_primitive> ::= “ContainerPrimitive/” (<boolean> | <char> | <wchar>

| <double> | <long_double> | <float> | <long>

| <unsigned_long> | <long_long> | <unsigned_long_long>

| <octet> | <short> | <unsigned_short>)

<boolean> ::= “Boolean/” (“Boolean1” | “Boolean10” | “Boolean100”

| “Boolean1000” | “Boolean10000”)

<char> ::= “Char/” (“Char1” | “Char10” | “Char100” | “Char1000” | “Char10000”)

<wchar> ::= “WChar/”

(“WChar1” | “WChar10” | “WChar100” | “WChar1000” | “WChar10000”)

<double> ::= “Double/”

(“Double1” | “Double10” | “Double100” | “Double1000” | “Double10000”)

<long_double> ::= “LongDouble/” (“LongDouble1” | “LongDouble10”

| “LongDouble100” | “LongDouble1000” | “LongDouble10000”)

<float> ::= “Float/” (“Float1” | “Float10” | “Float100” | “Float1000” | “Float10000”)

<long> ::= “Long/” (“Long1” | “Long10” | “Long100” | “Long1000” | “Long10000”)

<unsigned_long> ::= “UnsignedLong/”

(“UnsignedLong1” | “UnsignedLong10” | “UnsignedLong100”

| “UnsignedLong1000” | “UnsignedLong10000”)

<long_long> ::= “LongLong/” (“LongLong1” | “LongLong10” | “LongLong100”

| “LongLong1000” | “LongLong10000”)

<unsigned_long_long> ::= “UnsignedLongLong/ (“UnsignedLongLong1”

| “UnsignedLongLong10” | “UnsignedLongLong100”

| “UnsignedLongLong1000” | “UnsignedLongLong10000”)

<octet> ::= “Octet/” (“Octet1” | “Octet10” | “Octet100” | “Octet1000” | “Octet10000”)

<short> ::= “Short/” (“Short1” | “Short10” | “Short100” | “Short1000” | “Short10000”)

<unsigned_short> ::= “Unsigned_short/”

(“UnsignedShort1” | “UnsignedShort10” | “UnsignedShort100”

| “UnsignedShort1000” | “UnsignedShort10000”)

<container_constructed> ::= “ContainerConstructed/”

(<container_struct> | <container_interface>

| <container_union> | <container_enum>)

<container_struct> ::= “ContainerStruct/”

(<container_primitive> | <container_all_primitives>)

<container_interface> := “ContainerInterface/” (<container_primitive>

| <container_all_primitives> | < container_empty>)

<container_union> ::= ContainerUnion/” (<container_all_primitives_octet>

| <container_all_primitives_double>)

<container_enum> ::= “ContainerEnum/” (<container_all_primitives>)

<container_all_primitives> ::= “ContainerAllPrimitives/”

(“AllPrimitives1” | “AllPrimitives10” | “AllPrimitives100”

| “AllPrimitives1000” | “AllPrimitives10000”)

<container_all_primitivesoctet> ::= “ContainerAllPrimitivesOctet/”

(“AllPrimitivesOctet1” | “AllPrimitivesOctet10”

| “AllPrimitivesOctet100” | “AllPrimitivesOctet1000”

| “AllPrimitivesOctet10000”)

<container_all_primitives_double> ::= “ContainerAllPrimitivesDouble/”

(“AllPrimitivesDouble1” | “AllPrimitivesDouble10”

| “AllPrimitivesDouble100”

| “AllPrimitivesDouble1000”

| “AllPrimitivesDouble10000”)

<container_empty> ::= “ContainerEmpty/” (“Empty1” | “Empty10” | “Empty100”

| “Empty1000” | “Empty10000”)

Every parse of this grammar gives you a test result obtained, of course if it is supported by the ORB. For example, we obtained a result for “JDK1.3_02//Local/Oneway/Invoke”. That is we have a result obtained by using Sun’s *JDK1.3*_02 IDL compiler, making *Local* O*neway* calls by sending and getting no parameters, only *Invoking* an operation which takes no arguments and returns nothing (void) .

4.6 Criteria Used

Our benchmarking environment is constructed by considering the following criteria.

4.6.1 used CORBA/Java ORB

A CORBA/Java ORB is an ORB which is written fully in Java, i.e, it includes no native code. We applied the benchmark to most common CORBA/Java ORB

worldwide, Sun's Java IDL compiler. It is so common because It comes with Java 2 SDK, freely.

4.6.2 local versus remote calls

We have the servers and the client at the same computer (local calls) or two sides are located at different computers (remote calls).

When making local calls the ORB implementor can use more efficient ways of passing parameters than remote calls. As an example, since for local calls, both client and server use the same memory, shared memory can be used. If it is so, then the network overhead is discarded.

But, it is reported that Java VM is CPU-sensitive [47] and using two computers doubles the number of CPUs used. It can eliminate the advantages of using local system and even the network overhead can be defeated.

For local calls we used a PC with a Pentium Celeron 850 processor and 128 MB of RAM.

For remote calls, our servers are run on the PC which is used for local calls and our client is on a PC with Pentium II MMX 400 processor and 64 MB of RAM.

Both of our computers use Microsoft Windows 2000 Professional as operating system.

Our computers are located on an idle 10 Mbps Ethernet for remote calls.

4.6.3 oneway versus twoway invocations

OMG IDL allows you to declare operation attribute at operation declaration that specifies which invocation semantics the communication service must provide for invocations of a particular operation.

When a client invokes an operation with the *oneway* attribute, the invocation semantics are *best-effort*, which does not guarantee delivery of the call; the operation will be invoked *at-most-once*.

If a client invokes an operation without the oneway attribute (i.e, twoway), the operation semantics are at-most-once if an exception raised; the semantics are *exactly-once* if the operation invocation returns successfully [7].

The CORBA standard does not require oneway operations to be non-blocking, but most implementations of CORBA does not block the caller of a oneway operation. The CORBA standard has left a great freedom in how an ORB handles oneway operations [48].

4.6.4 flow of parameters

CORBA IDL defines three directional attributes to parameters :

- in : the parameter is passed from client to server
- out : the parameter is passed from server to client
- inout : the parameter is passed in both directions.

If no parameters will be passed then you must leave the parameter declaration section of the operation empty and operation's return result type must be the keyword *void*.

We have named these conditions as only_send, only_get, send_get and invoke, respectively.

We could use out and inout directional attributes to pass parameters back from the server and to and back from the server. But in that case, as a programmer, we should have handled the creation and use of Holder classes in accord with the generated ones by the IDL-to-Java mapping of these parameters. But we defined our operations as taking (for only_send and send_get) or returning (for only_get and send_get) the type, and in the case of invoke taking no parameters and returning void. In this way we have left the preparation operations for sending and getting the parameters to ORB.

4.6.5 CORBA types used

We classified the CORBA IDL types into three categories :

- Primitive Types
- Constructed Types
- Container Types

4.6.5.1 primitive types

Primitive types are the IDL allowed basic types which consists of the following :

- *boolean* type stores a boolean value. IDL defines two boolean constants: *true* and *false.*
- *char* type stores a single character value. The char type is an 8-bit quantity.
- *wchar is* wide character type. Its size is implementation-dependent.
- *long* is a 32-bit signed quantity with a range of -2^{31} to $2^{31}-1$.
- *unsigned long* is a 32-bit unsigned quantity with a range of 0 to $2^{32}-1$.
- *long long* is a 64-bit signed quantity with a range of -2^{63} to $2^{63}-1$.
- *unsigned long long* is a 64-bit unsigned quantity with a range of 0 to $2^{64}-1$.
- *short* is a 16-bit signed quantity with a range of -2^{15} to $2^{15}-1$.
- *unsigned short* is a 16-bit unsigned quantity with a range of 0 to $2^{16}-1$.
- *float* is an IEEE single-precision floating point value.
- *double* is an IEEE double-precision floating point value.
- *octet* is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.
- *long double* is an IEEE double-extended floating point value having an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. Our ORB does not support this type. OMG says that this type is reserved for future support [14]. So, we do not have results for this type.

4.6.5.2 constructed types

OMG specifies structs, unions and enums as constructed types and mentions interfaces in another header. We added the interfaces to our constructed types, since they can be constructed from other types and can contain no methods.

Indeed, even if you don't declare any methods, ORB creates the *accessor* and *mutator* functions for each attribute of the interface. But from the IDL view, it is correct that we can have interfaces without methods.

We have a struct with only a boolean field, only a char field, etc. Same thing is true for our interfaces : an interface with only a boolean attribute, a char attribute, etc. We have our structs and interfaces for each of the primitive types. And we have an interface, struct, enum, and union which has a field for every primitive type. We have also an empty interface, an interface with no attributes and methods. Empty structs are not allowed, so we have no empty struct. We tested our union with passing an octet value (1 byte) and passing a double value (8 bytes).

4.6.5.3 container types

We have arrays, sequences and strings in this category. OMG defines sequences and strings as *template types* and arrays in the title *complex declarator* [7].

4.6.6 IDL-to-Java mappings of used types

Table 4.1 shows the IDL-to-Java mapping of our tested types [14].

Table 4.1 : IDL-to-Java Mapping of primitive types

IDL Type	**Java type**
boolean	Boolean
char	Char
wchar	Char
octet	Byte
string	java.lang.String
wstring	java.lang.String
short	Short
unsigned short	Short
long	Int
unsigned long	Int
long long	Long
unsigned long long	Long
float	Float
double	Double

4.7 CDR Transfer Syntax

Here is the CDR transfer syntax described by OMG [7]. The Common Data Representation (CDR) transfer syntax is the format in which the GIOP represents OMG IDL data types in an octet stream.

4.7.1 primitive types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats include tags in message headers that indicate the byte ordering in the message. Encapsulations include an initial flag that indicates the byte ordering within the encapsulation. Primitive data types are encoded in multiples of octets. An octet is an 8-bit value.

4.7.1.1 short and unsigned short

Short values are represented as two's complement numbers. Figure 4.2 illustrates the bit ordering and size of shorts. Unsigned shorts also have the same format but they are represented as unsigned binary numbers.

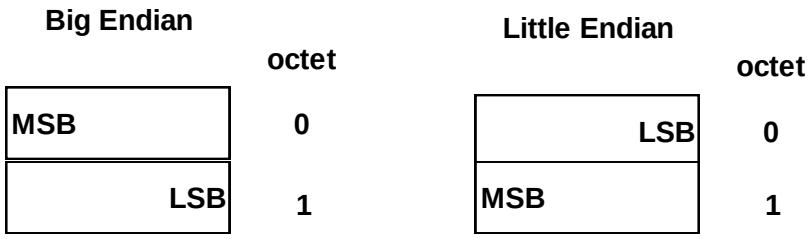


Figure 4.2 : Bit ordering and size of shorts and unsigned shorts in big-endian and little-endian encodings.

4.7.1.2 long and unsigned long

Long values are represented as two's complement numbers. Figure 4.3 illustrates the bit ordering and size of longs. Unsigned longs also have the same format but they are represented as unsigned binary numbers.

4.7.1.3 long long and unsigned long long

Long long values are represented as two's complement numbers. Figure 4.4 illustrates the bit ordering and size of long longs. Unsigned long longs also have the same format but they are represented as unsigned binary numbers.

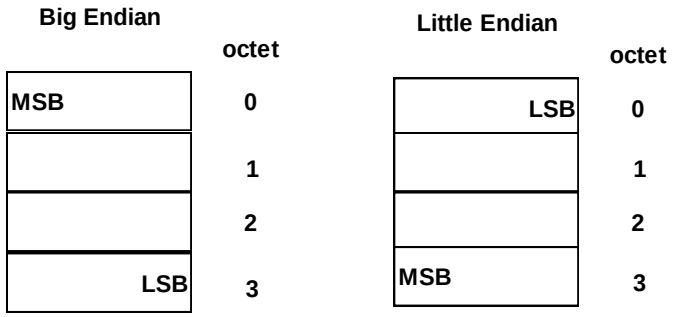


Figure 4.3 : Bit ordering and size of longs and unsigned longs in big-endian and little-endian encodings.

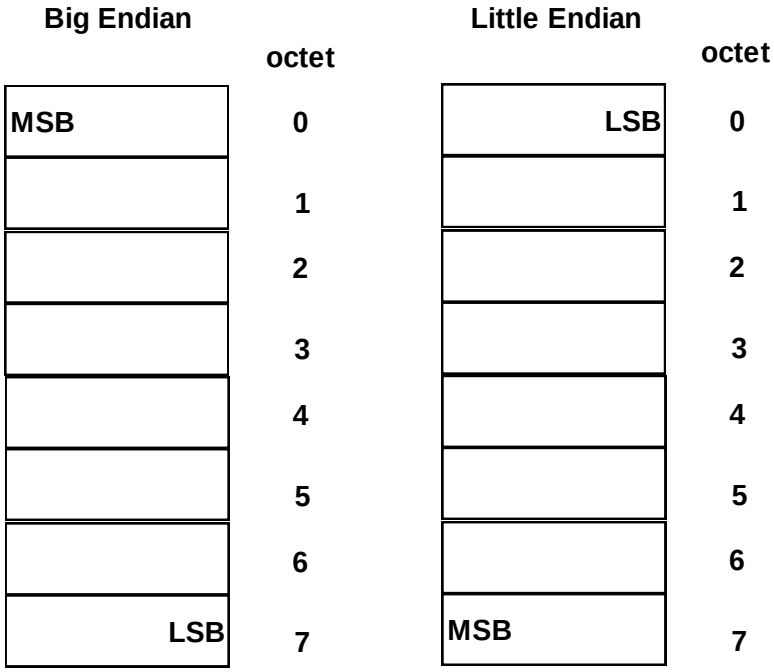


Figure 4.4 : Bit ordering and size of long longs and unsigned long longs in big-endian and little-endian encodings.

4.7.1.4 float

Figure 4.5 illustrates the representation of floating point numbers. The figure shows three different components for floating point numbers, the sign bit (s), the exponent (e) and the fractional part (f) of the mantissa. The sign bit has values of 0 or 1, representing positive and negative numbers, respectively.

For single-precision float values the exponent is 8 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 127. The fractional mantissa (f1 - f3) is a 23-bit value f where 1.0 <= f < 2.0, f1 being most significant and f3 being least significant. The value of a normalized number is described by $1^{sign} \times 2^{(exponent - 127)} \times (1+ fraction)$.

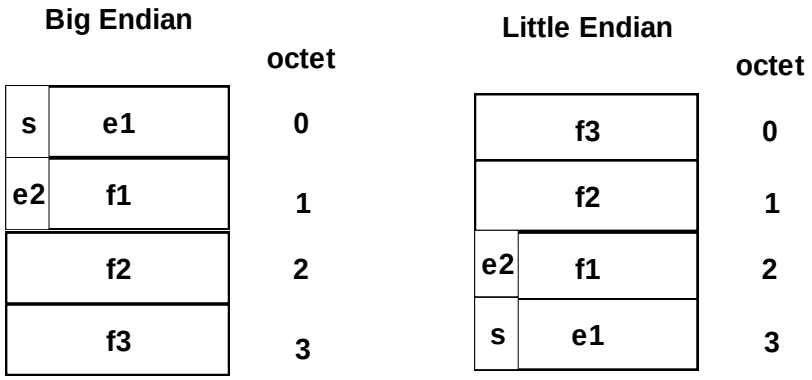


Figure 4.5 : Bit ordering and size of floating point numbers in big-endian and little-endian encodings.

4.7.1.5 double

Figure 4.6 illustrates the representation of double-precision numbers. For double-precision values the exponent is 11 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 1023. The fractional mantissa (f1 - f7) is a 52-bit value m where 1.0 <= m < 2.0, f1 being most significant and f7 being least significant. The value of a normalized number is described by $1^{sign} \times 2^{(exponent - 1023)} \times (1 + fraction)$.

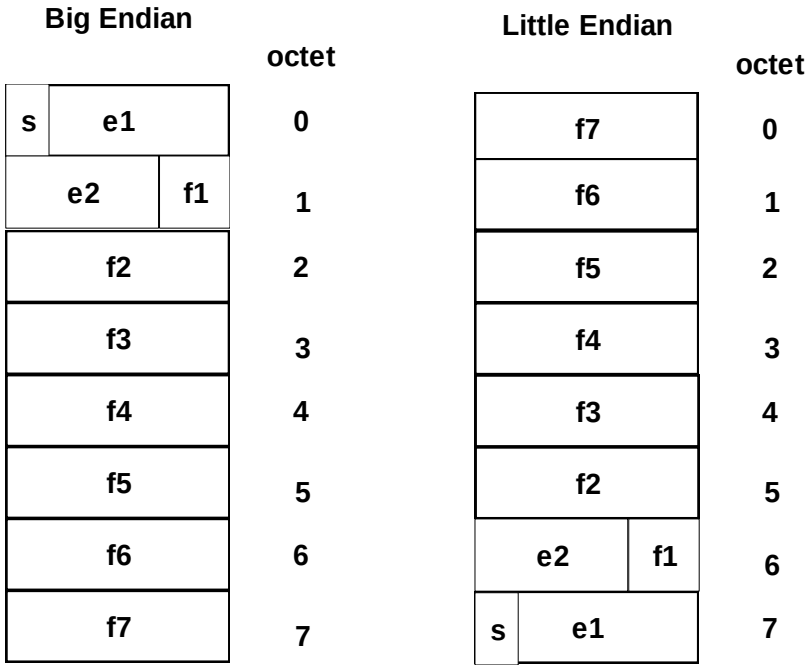


Figure 4.6 : Bit ordering and size of double-precision numbers in big-endian and little-endian encodings.

4.7.1.6 long double

Figure 4.7 illustrates the representation of double-extended floating-point numbers. For double-extended floating-point values the exponent is 15 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are the most significant. The fractional mantissa (f1 through f14) is 112 bits long, with f1 being the most significant. The value of a long double is determined by $1^{sign} \times 2^{(exponent - 16383)} \times (1 + fraction)$. Long double is not supported by our ORB as mentioned at section 4.6.5.1.

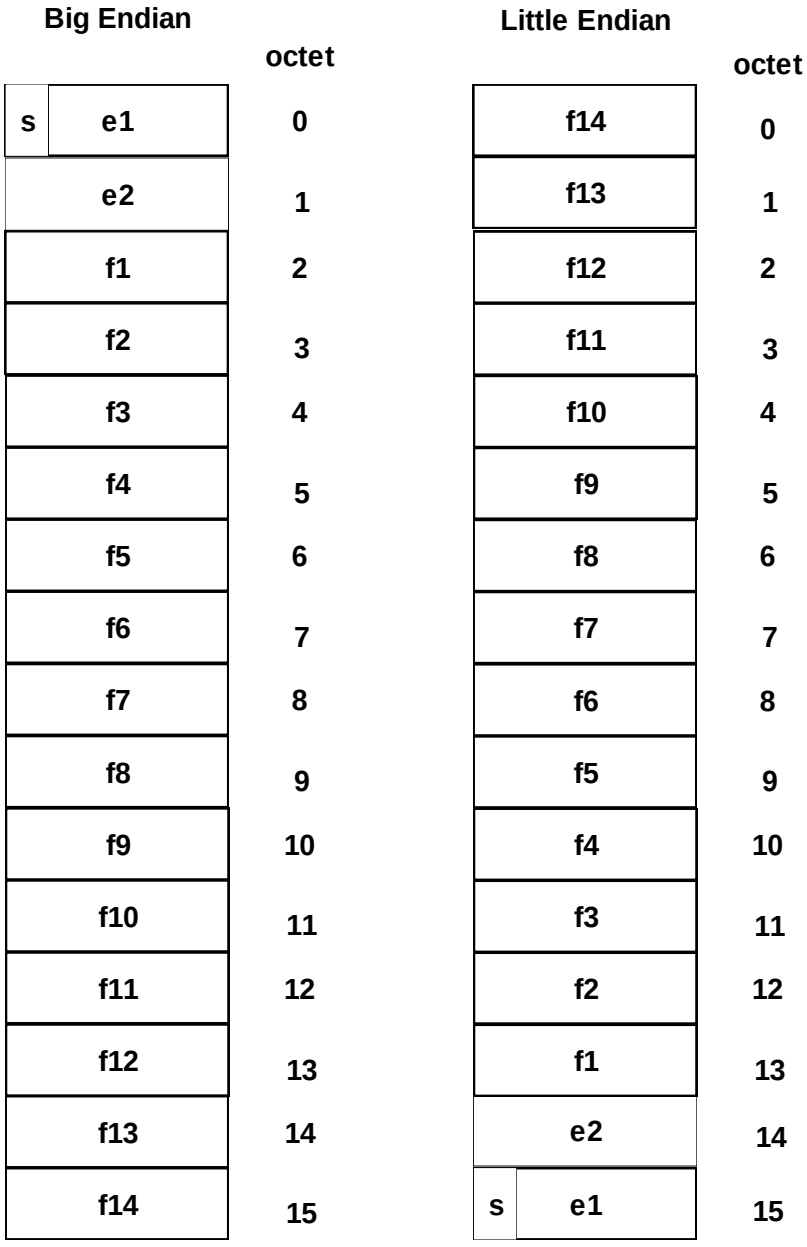


Figure 4.7 : Bit ordering and size of double-extended numbers in big-endian and little-endian encodings.

4.7.1.7 octet

Octets are uninterpreted 8-bit values whose contents are guaranteed not to undergo any conversion during transmission. Octets may be considered as unsigned 8-bit integer values.

4.7.1.8 boolean

Boolean values are encoded as single octets, where TRUE is the value 1, and FALSE as 0.

4.7.1.9 char and wchar

An IDL character is represented as a single octet. If the transmission code set is byte-oriented then each wide character is represented as one or more octets. If the transmission code set is non-byte-oriented then it is dependent on the character set. if the character set contains 2 bytes, then wide characters are represented as unsigned shorts. if the character set contains 4 bytes, then they are represented as unsigned longs.

4.7.2 constructed types

As mentioned before constructed types are derived from other types. CDR rules governing the constructed types are as follows.

4.7.2.1 struct

The components of a structure are encoded in the order of their declaration in the structure. Each component is encoded as defined for its data type.

4.7.2.2 union

Unions are encoded as the discriminant tag of the type specified in the union declaration, followed by the representation of any selected member, encoded as its type indicates.

4.7.2.3 enum

Enum values are encoded as unsigned longs. The numeric values associated with enum identifiers are determined by the order in which the identifiers appear in the

enum declaration. The first enum identifier has the numeric value zero (0). Successive enum identifiers take ascending numeric values, in order of declaration from left to right.

4.7.2.4 interface

We could not see any CDR rule at the OMG's formal specification.

4.7.3 container types

4.7.3.1 array

Arrays are encoded as the array elements in sequence. As the array length is fixed, no length values are encoded. Each element is encoded as defined for the type of the array. In multidimensional arrays, the elements are ordered so the index of the first dimension varies most slowly, and the index of the last dimension varies most quickly.

4.7.3.2 sequence

Sequences are encoded as an unsigned long value, followed by the elements of the sequence. The initial unsigned long contains the number of elements in the sequence. The elements of the sequence are encoded as specified for their type.

4.7.3.3 strings and wide strings

A string is encoded as an unsigned long indicating the length of the string in octets, followed by the string value in single- or multi-byte form represented as a sequence of octets. The string contents include a single terminating null character. The string length includes the null character, so an empty string has a length of 1.

A wide string is encoded as an unsigned long indicating the length of the string in octets or unsigned integers (determined by the transfer syntax for wchar) followed by the individual wide characters. The string contents include a single terminating null character. The string length includes the null character. The terminating null character for a wstring is also a wide character.

5. RESULTS FOR JDK1.3_02

In this chapter we present the results for the benchmark we have constructed.

5.1 About Sun's IDL Compiler

The Java IDL API, introduced in Version 1.2 of the Java 2 platform, provides an interface between Java programs and distributed objects and services built using the CORBA. Java IDL is an implementation of the standard Java Software Development Kit (SDK) in the org.omg.CORBA and org.omg.CosNaming (CORBA naming service support) packages and their subpackages [49]. Sun provides programmers with an idl-to-java compiler named *idlj* (its old name is *idl2java*).

5.2 A Note on Our Graphics

We have used abbreviations for the primitive types in order to save space. The abbreviations and their meanings at their order of appearance in graphics are given in table 5.1.

Table 5.1 : Abbreviations used in our graphics

C : Char	WC : Wide Character	D : Double	L : Long	UL : Unsigned Long
LL : Long Long	ULL : Unsigned Long Long	F : Float	S : Short	US : Unsigned Short
B : Boolean	O : Octet	A : All Primitives	E : Empty	WString : Wide String

5.3 Results for Local Calls

Following are the results for our local calls taken as described at section 4.6.2.

5.3.1 local oneway and twoway invocation results

Figure 5.1 shows the results for oneway and twoway functions which take no arguments and return no result.

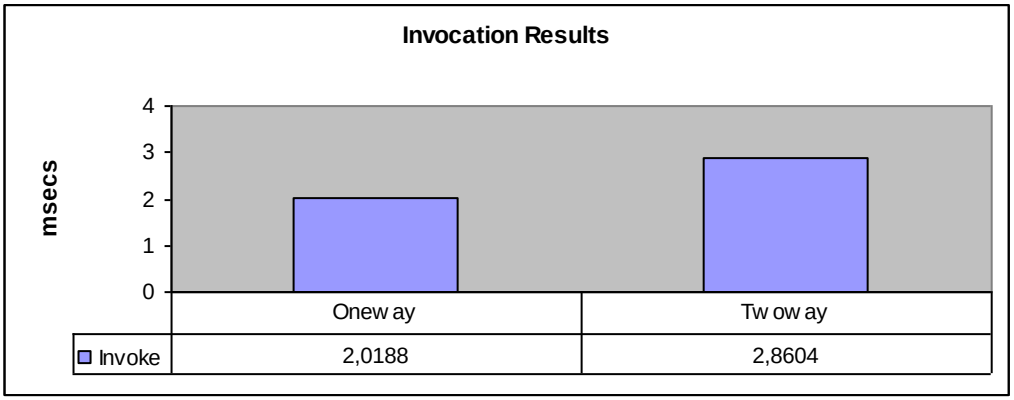


Figure 5.1 : Local results for Oneway and Twoway Invocations

5.3.1.1 comments on oneway and twoway invocation results

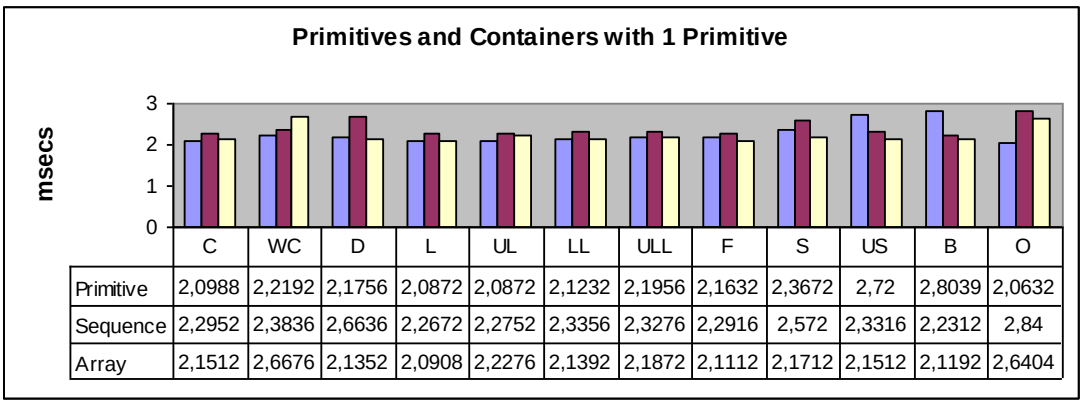
It is seen from the results that oneway invocations are faster than twoways. It is natural to have such a result since oneway has no complaints about reliability and chores related with it is only handled with twoway calls.

5.3.2 oneway – only send results

We will briefly refer to these results as L_O_OS (Local_Oneway_OnlySend) results.

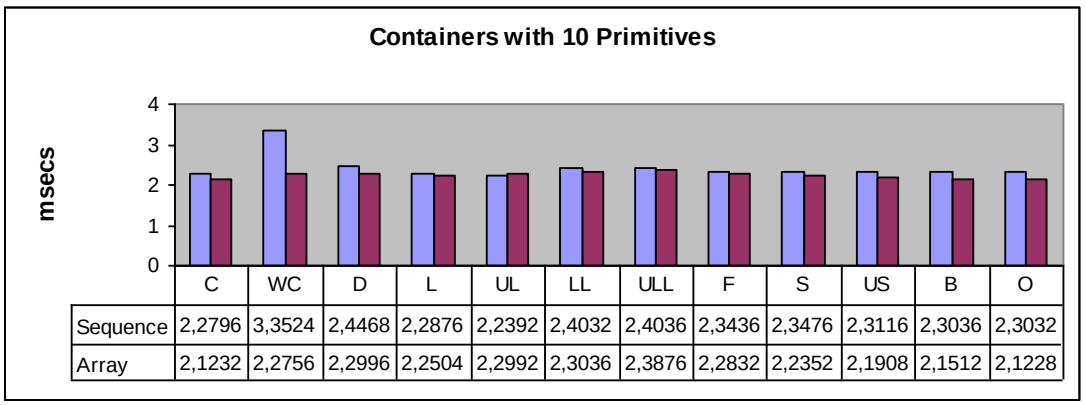
5.3.2.1 L_O_OS primitive and primitive container results

Figure 5.2 through 5.6 shows the results obtained for primitive types and containers with primitive types.



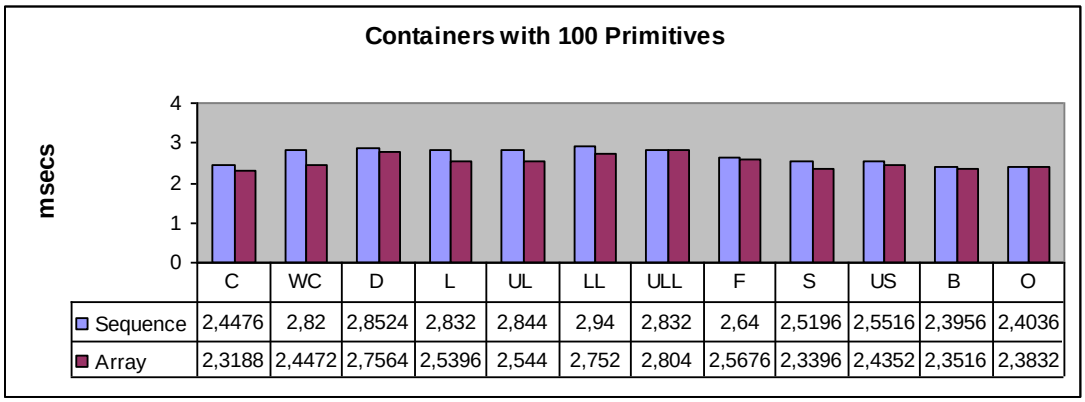
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Primitive	2,0988	2,2192	2,1756	2,0872	2,0872	2,1232	2,1956	2,1632	2,3672	2,72	2,8039	2,0632
Sequence	2,2952	2,3836	2,6636	2,2672	2,2752	2,3356	2,3276	2,2916	2,572	2,3316	2,2312	2,84
Array	2,1512	2,6676	2,1352	2,0908	2,2276	2,1392	2,1872	2,1112	2,1712	2,1512	2,1192	2,6404

Figure 5.2 : L_O_OS Results for Primitives and Containers with 1 Primitive



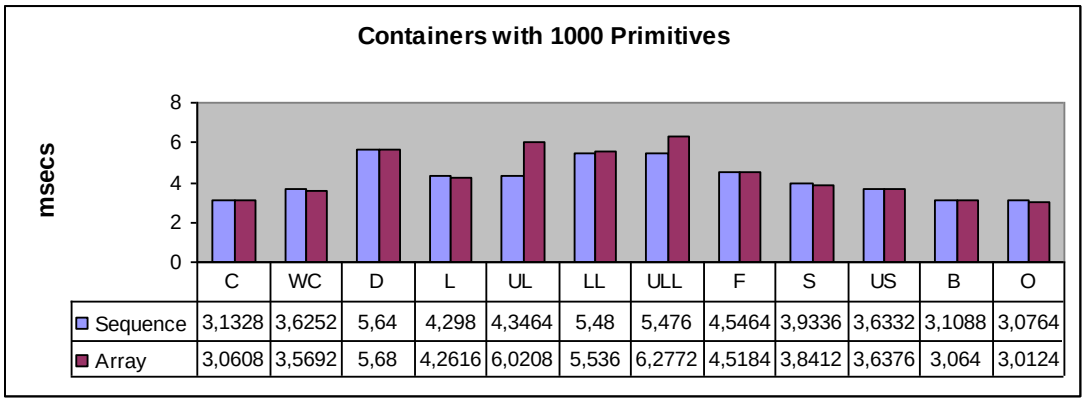
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,2796	3,3524	2,4468	2,2876	2,2392	2,4032	2,4036	2,3436	2,3476	2,3116	2,3036	2,3032
Array	2,1232	2,2756	2,2996	2,2504	2,2992	2,3036	2,3876	2,2832	2,2352	2,1908	2,1512	2,1228

Figure 5.3 : L_O_OS Results for Containers with 10 Primitives



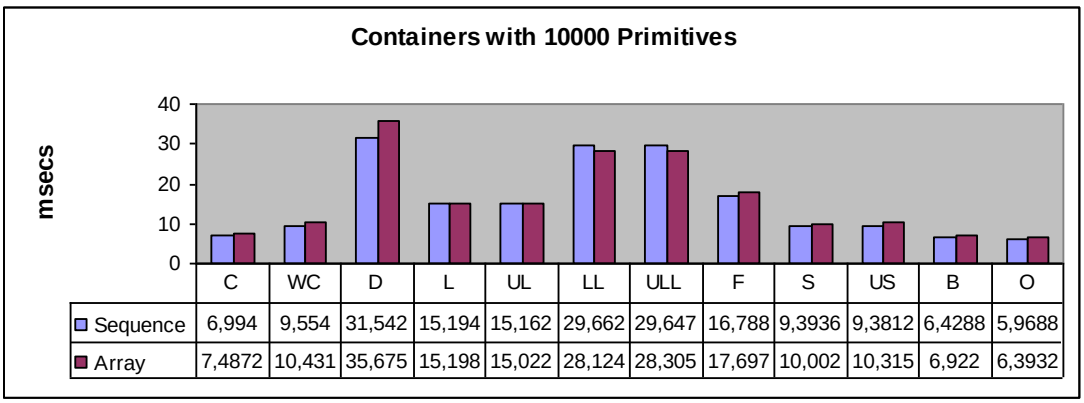
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,4476	2,82	2,8524	2,832	2,844	2,94	2,832	2,64	2,5196	2,5516	2,3956	2,4036
Array	2,3188	2,4472	2,7564	2,5396	2,544	2,752	2,804	2,5676	2,3396	2,4352	2,3516	2,3832

Figure 5.4 : L_O_OS Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,1328	3,6252	5,64	4,298	4,3464	5,48	5,476	4,5464	3,9336	3,6332	3,1088	3,0764
Array	3,0608	3,5692	5,68	4,2616	6,0208	5,536	6,2772	4,5184	3,8412	3,6376	3,064	3,0124

Figure 5.5 : L_O_OS Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	6,994	9,554	31,542	15,194	15,162	29,662	29,647	16,788	9,3936	9,3812	6,4288	5,9688
Array	7,4872	10,431	35,675	15,198	15,022	28,124	28,305	17,697	10,002	10,315	6,922	6,3932

Figure 5.6 : L_O_OS Results for Containers with 10000 Primitives

5.3.2.2 about L_O_OS primitive and primitive container results

Some conclusions from the results are :

- It is seen from our results (not from the graphics above, but the 25 samples we have taken. The above results are the average of these 25 values) for each repetition that an observable difference exists between the first call and second call. The difference between the second call and following calls are very small. This conclusion applies to all of our local results and will not be mentioned again. The reason behind this could be that when the code is executed for the first time, the loading of code fragment is from main memory, at the best wish, and could get some time. After this first call, it could be taken to the cache memory and to reach it could get less time. The fluctuations appearing at the middle could be the result of operating system taking the code fragment away from cache for a while and using cache for some other work. After this, it could again be taken to cache when it is used again by the application.

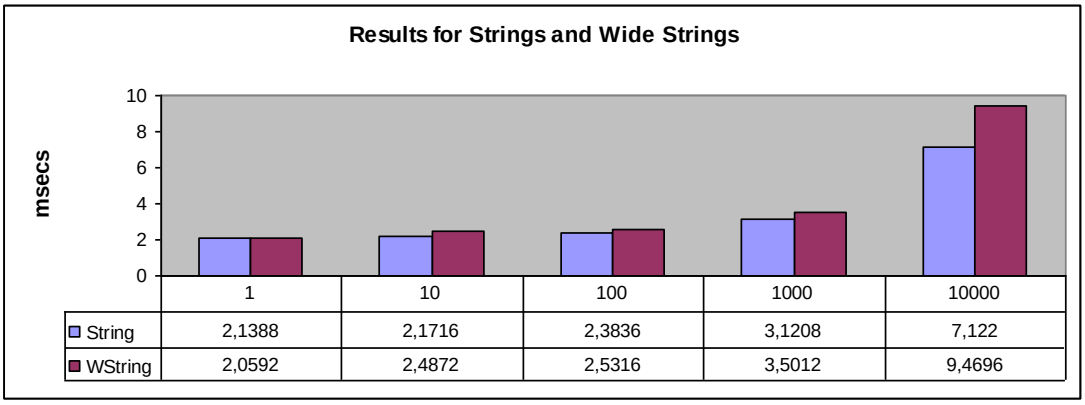
- If we send a primitive, a primitive within an array or a sequence, almost no difference appears. Our arrays and sequences are fixed size. According to the CDR rules, only values of array will be sent after marshalling. But for the sequence we could expect a little delay because of the preceding sending of unsigned long value which represents the size of the data to be sent. Our results do not seem to conform to this second conclusion. Since arrays and sequences are mapped to same Java type, and they have both fixed sizes our ORB seems to handle arrays and sequences in the same way.

- The difference among the results for containers with size 1, 10 and 100 are very little. But when we take the results for the size of 1000, we face with a sharp change. When we have 10000 elements, we have another sharp change. Indeed, we could expect that when we have doubled the size of the array, the time must have been doubled also. But it is not the case here. Its reason could be that when we have 1, 10 and 100 values they fix in a message. But when we have larger data, we have larger number of messages. So, our time is proportional to the number of messages sent, not only to the number of bytes sent (The proportion can be the ceiling of the ratio (number of bytes / message size)).

- wchar can be represented with two bytes (as unsigned shorts) or four bytes (as unsigned longs) according to the choosen character set. Our results show that wchar is represented with two bytes since it has nearly the same results with types having size of 2 bytes (e.g, short).

5.3.2.3 L_O_OS string and wide string results

Figure 5.7 shows the results obtained.



	1	10	100	1000	10000
String	2,1388	2,1716	2,3836	3,1208	7,122
WString	2,0592	2,4872	2,5316	3,5012	9,4696

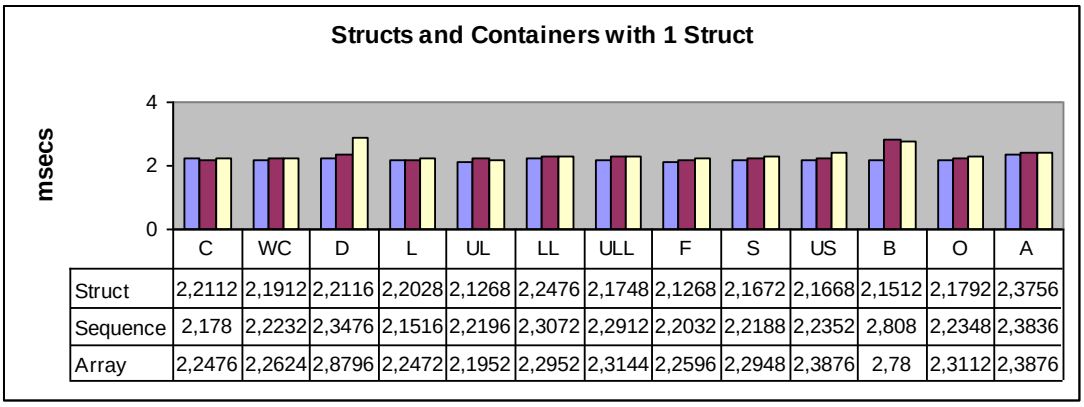
Figure 5.7 : L_O_OS Results for Strings and Wide Strings

5.3.2.4 about L_O_OS string and wide string results

Either we send character strings as array or sequence of characters (see results for primitve type of char and wchar at 5.3.3.1) or we send them as elements of a string, we see almost no difference.

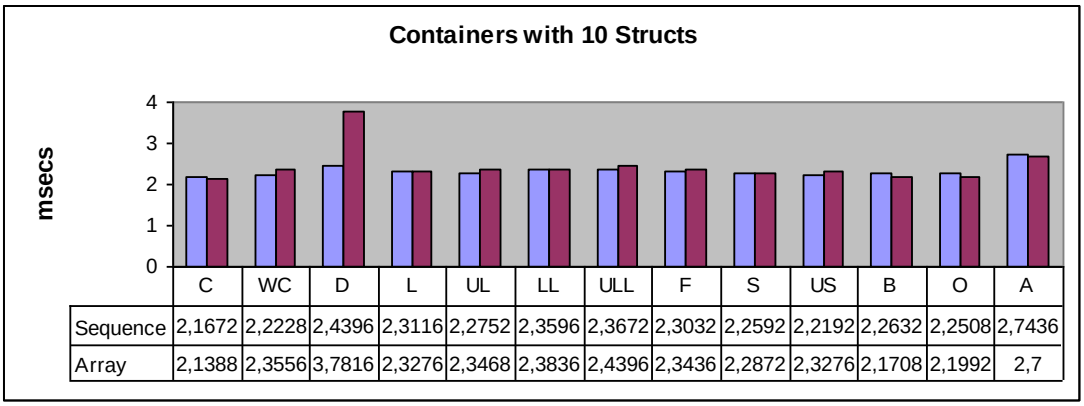
5.3.2.5 L_O_OS struct and struct container results

Figures 5.8 through 5.12 shows the results obtained.



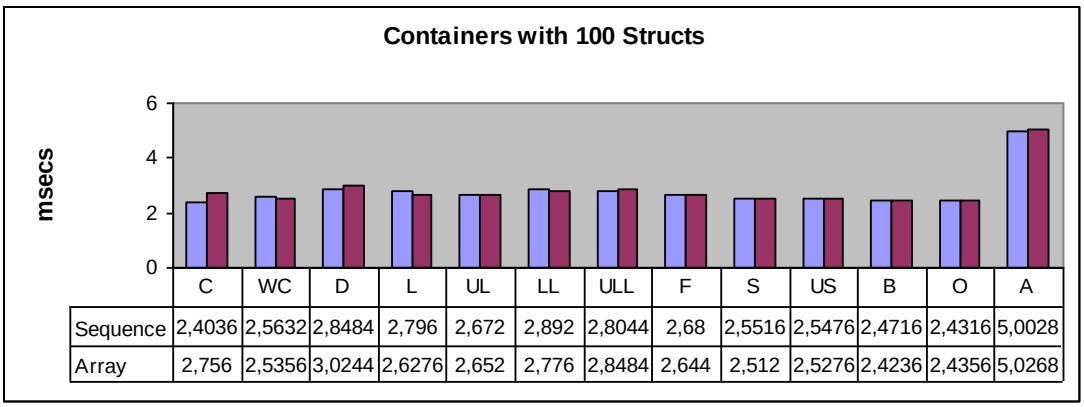
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Struct	2,2112	2,1912	2,2116	2,2028	2,1268	2,2476	2,1748	2,1268	2,1672	2,1668	2,1512	2,1792	2,3756
Sequence	2,178	2,2232	2,3476	2,1516	2,2196	2,3072	2,2912	2,2032	2,2188	2,2352	2,808	2,2348	2,3836
Array	2,2476	2,2624	2,8796	2,2472	2,1952	2,2952	2,3144	2,2596	2,2948	2,3876	2,78	2,3112	2,3876

Figure 5.8 : L_O_OS Results for Structs and Containers with 1 Struct



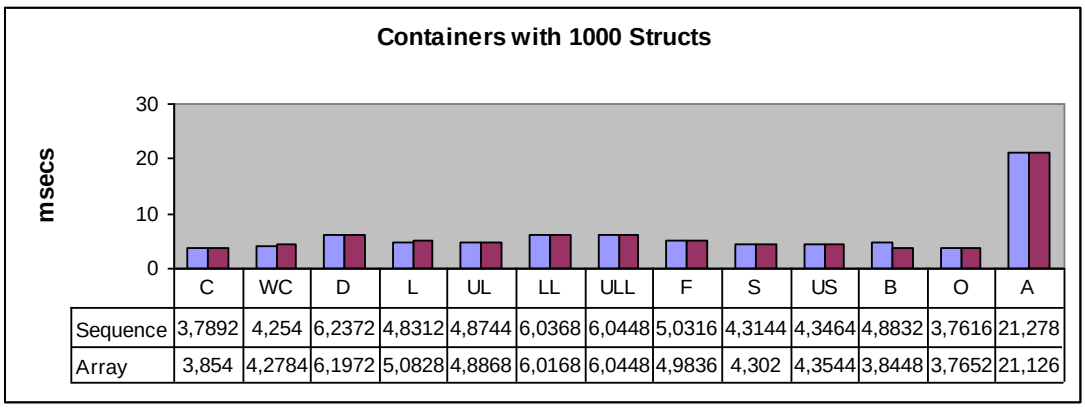
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,1672	2,2228	2,4396	2,3116	2,2752	2,3596	2,3672	2,3032	2,2592	2,2192	2,2632	2,2508	2,7436
Array	2,1388	2,3556	3,7816	2,3276	2,3468	2,3836	2,4396	2,3436	2,2872	2,3276	2,1708	2,1992	2,7

Figure 5.9 : L_O_OS Results for Containers with 10 Structs.



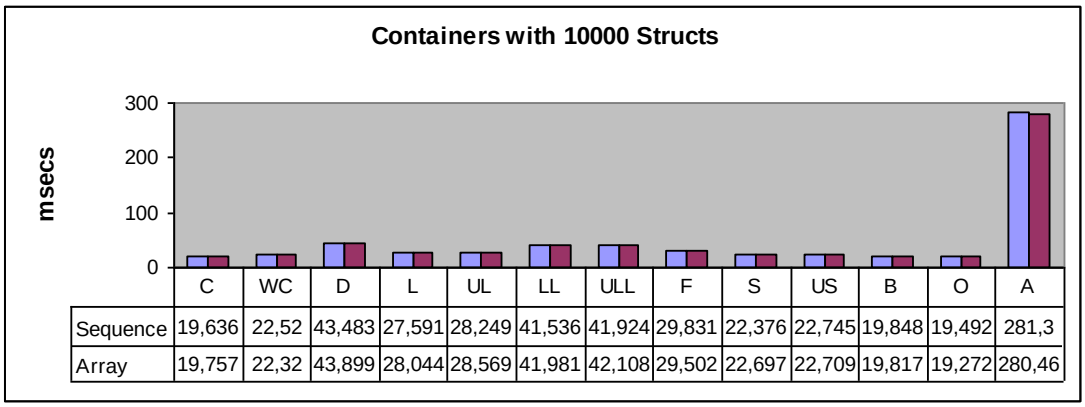
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,4036	2,5632	2,8484	2,796	2,672	2,892	2,8044	2,68	2,5516	2,5476	2,4716	2,4316	5,0028
Array	2,756	2,5356	3,0244	2,6276	2,652	2,776	2,8484	2,644	2,512	2,5276	2,4236	2,4356	5,0268

Figure 5.10 : L_O_OS Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,7892	4,254	6,2372	4,8312	4,8744	6,0368	6,0448	5,0316	4,3144	4,3464	4,8832	3,7616	21,278
Array	3,854	4,2784	6,1972	5,0828	4,8868	6,0168	6,0448	4,9836	4,302	4,3544	3,8448	3,7652	21,126

Figure 5.11 : L_O_OS Results for Containers with 1000 Structs.



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	19,636	22,52	43,483	27,591	28,249	41,536	41,924	29,831	22,376	22,745	19,848	19,492	281,3
Array	19,757	22,32	43,899	28,044	28,569	41,981	42,108	29,502	22,697	22,709	19,817	19,272	280,46

Figure 5.12 : L_O_OS Results for Containers with 10000 Structs

5.3.2.6 about L_O_OS struct and struct container results

Some conclusions from the results are :

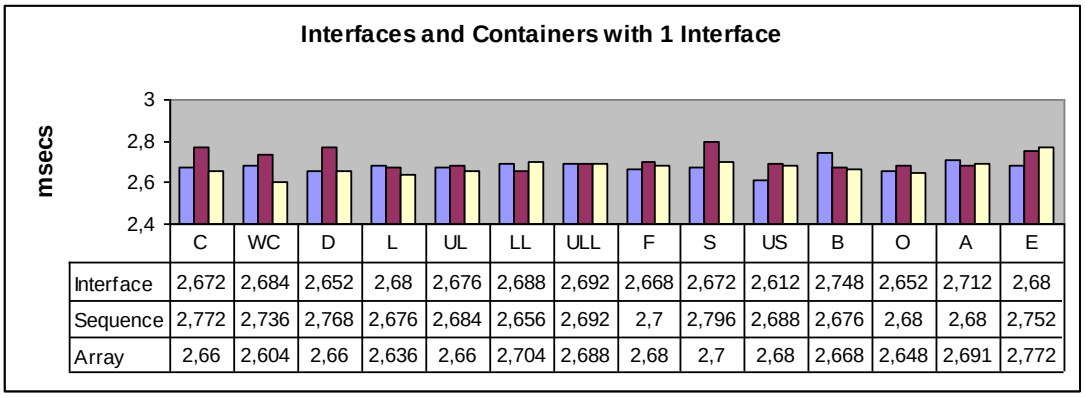
- If we consider the results obtained with only primitives and the results here we see that when we encapsulate a primitive within a struct, we see no observable difference between results for small sizes. But for the big-sized data, primitives perform better.

- Also it is true that placing the primitives within structs and then within containers make no difference with the stattements above..

- We have a special struct which consits of the fields for every primitive type. It gives almost same results with little sizes. But when size enlarges, the difference becomes apparent. Its reason is that according to CDR rules when handling the structs, fields of it must be sent in order of their declaration.

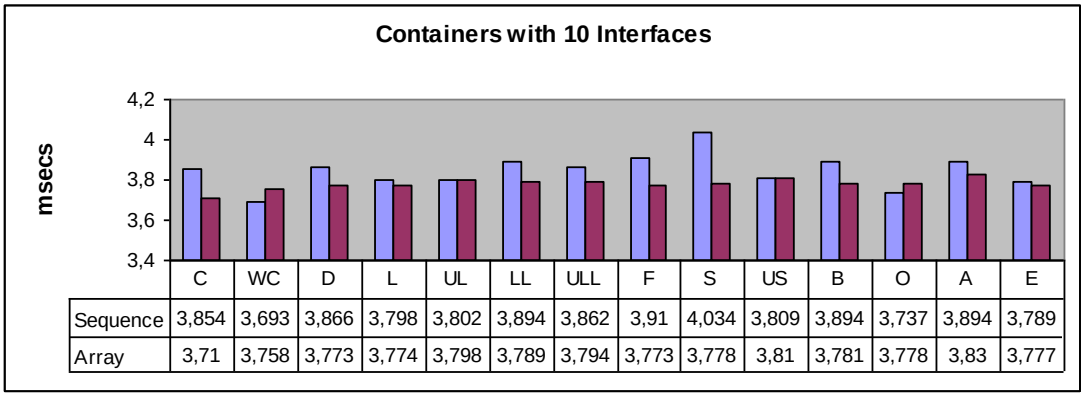
5.3.2.7 L_O_OS interface and interface container results

Figures 5.13 through 5.16 shows the results obtained.



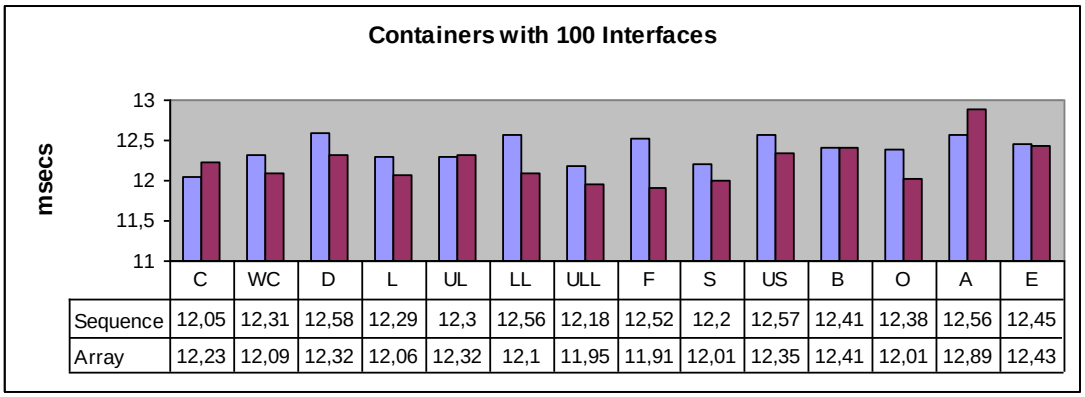
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	2,672	2,684	2,652	2,68	2,676	2,688	2,692	2,668	2,672	2,612	2,748	2,652	2,712	2,68
Sequence	2,772	2,736	2,768	2,676	2,684	2,656	2,692	2,7	2,796	2,688	2,676	2,68	2,68	2,752
Array	2,66	2,604	2,66	2,636	2,66	2,704	2,688	2,68	2,7	2,68	2,668	2,648	2,691	2,772

Figure 5.13 : L_O_OS Results for Interface and Containers with 1 Interface



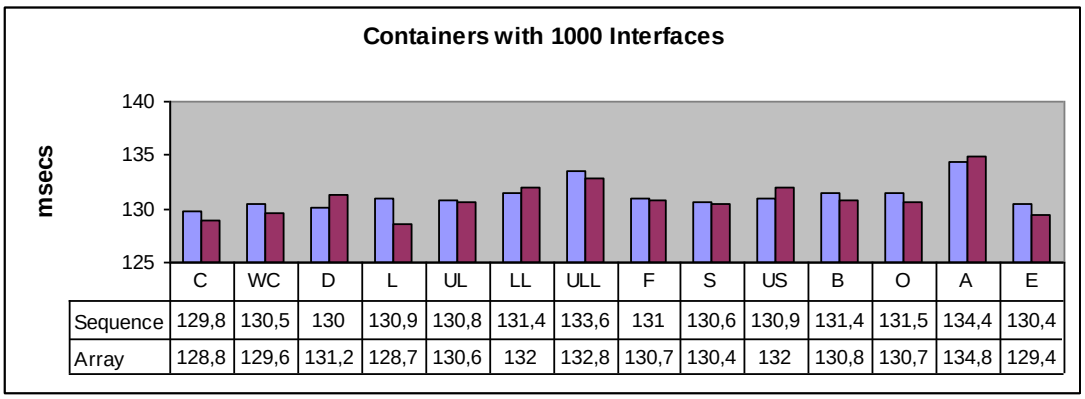
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	3,854	3,693	3,866	3,798	3,802	3,894	3,862	3,91	4,034	3,809	3,894	3,737	3,894	3,789
Array	3,71	3,758	3,773	3,774	3,798	3,789	3,794	3,773	3,778	3,81	3,781	3,778	3,83	3,777

Figure 5.14 : L_O_OS Results for Containers with 10 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	12,05	12,31	12,58	12,29	12,3	12,56	12,18	12,52	12,2	12,57	12,41	12,38	12,56	12,45
Array	12,23	12,09	12,32	12,06	12,32	12,1	11,95	11,91	12,01	12,35	12,41	12,01	12,89	12,43

Figure 5.15 : L_O_OS Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	129,8	130,5	130	130,9	130,8	131,4	133,6	131	130,6	130,9	131,4	131,5	134,4	130,4
Array	128,8	129,6	131,2	128,7	130,6	132	132,8	130,7	130,4	132	130,8	130,7	134,8	129,4

Figure 5.16 : L_O_OS Results for Containers with 1000 Interfaces

10000 interfaces produced the Out Of Memory error.

5.3.2.8 about L_O_OS interface and interface container results

Some conclusions from the results are :

- If we consider the results obtained with only primitives and the results here we see that when we encapsulate a primitive within an interface, the performance is reduced even with the small number of values. It could be the result of accessor and mutator functions created. But interface without members or functions (our empty interface) shows the same results. So, it is not a result of accessor or mutator functions but a result of type interface.

- The overhead introduced by interface type overwhelms the overhead of primitve types. So we have nearly the same results for all types.

- For 10000 elements we encountered the out of memory error at the server side.

5.3.2.9 L_O_OS union and enum results

Figures 5.17 through 5.21 shows the results obtained.

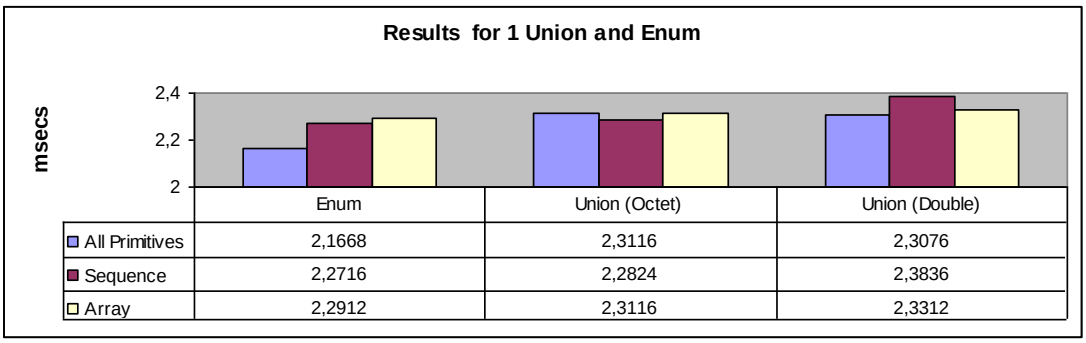


Figure 5.17 : L_O_OS Results for Union, Enum and Containers with 1 Union and Enum

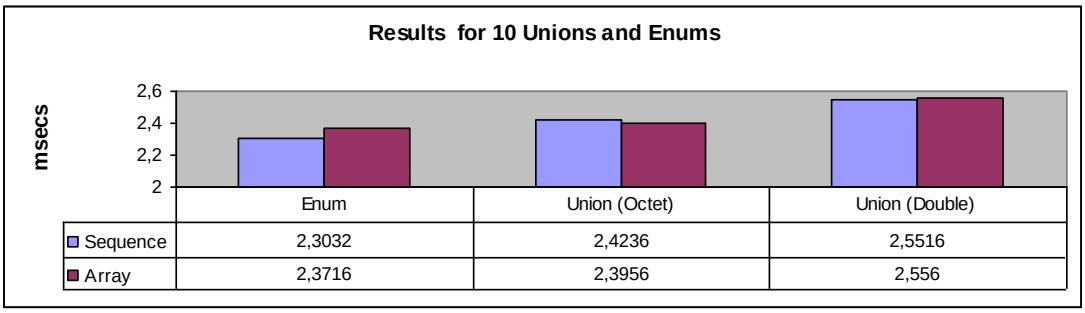


Figure 5.18 : L_O_OS Results for Containers with 10 Unions and Enums

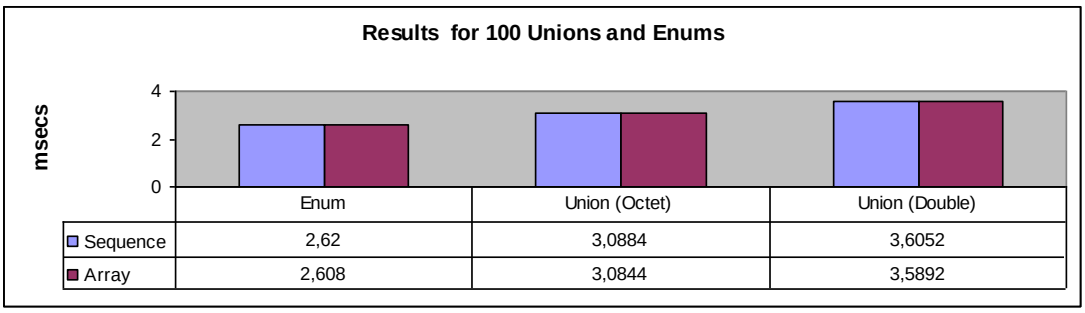


Figure 5.19 : L_O_OS Results for Containers with 100 Unions and Enums

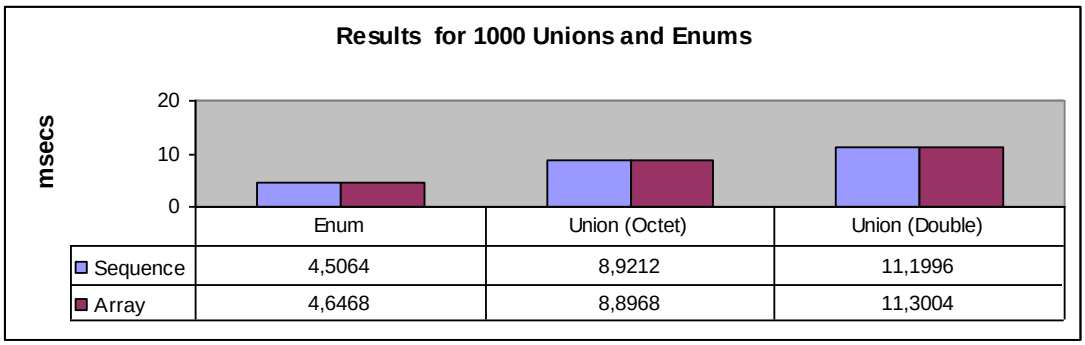


Figure 5.20 : L_O_OS Results for Containers with 1000 Unions and Enums

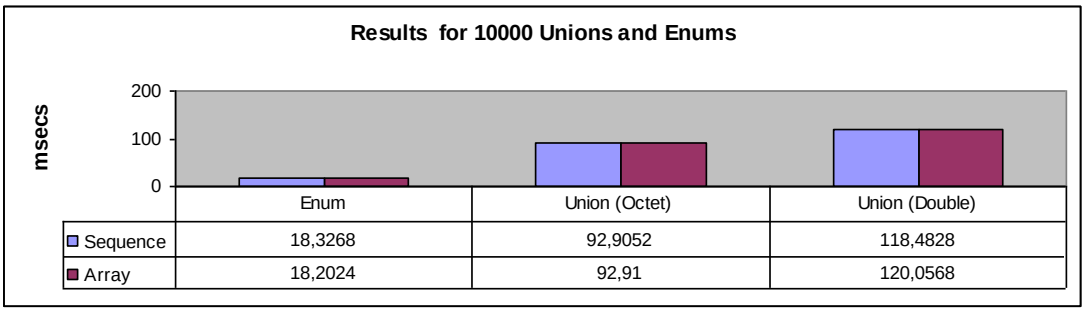


Figure 5.21 : L_O_OS Results for Containers with 10000 Unions and Enums

5.3.2.10 about L_O_OS union and enum results

Some conclusions from the results are :

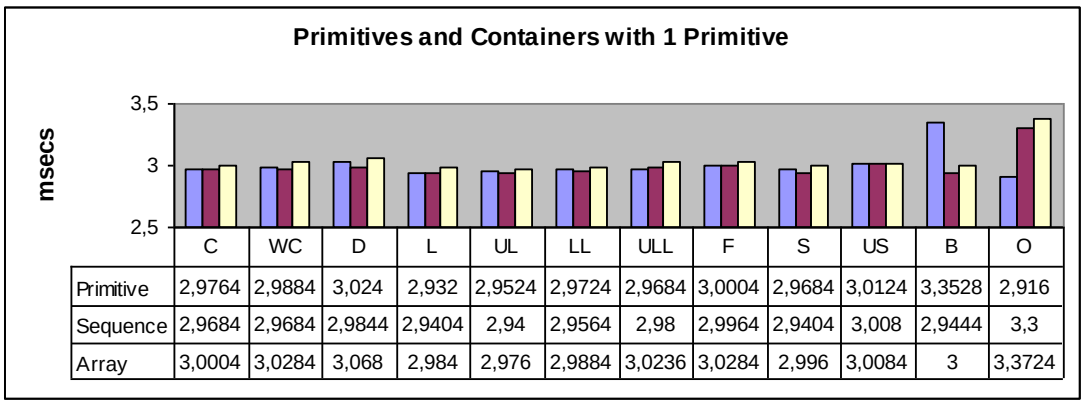
- CDR says that enum type is encoded as unsigned longs. Our enum results are close to unsigned longs. It could be said that enums have same performance with unsigned longs.
- Our results for unions, one carrying an octet and other a double, shows that for size 1, their performances are nearly the same. But when size increases, difference becomes apparent. Carrying double takes longer than carrying octet.

5.3.3 twoway – only send results

We will briefly refer to these results as L_T_OS (Local_Twoway_OnlySend) results.

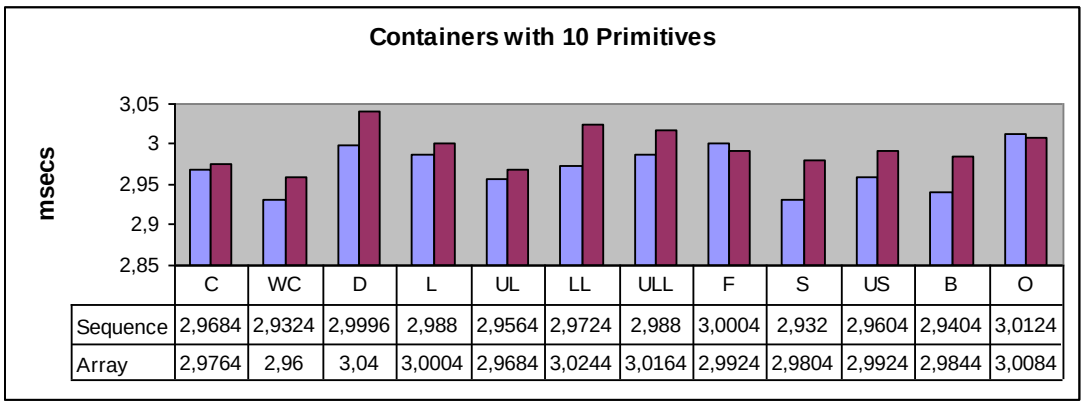
5.3.3.1 L_T_OS primitive and primitive container results

Figure 5.22 through 5.26 shows the results obtained for primitive types and containers with primitive types.



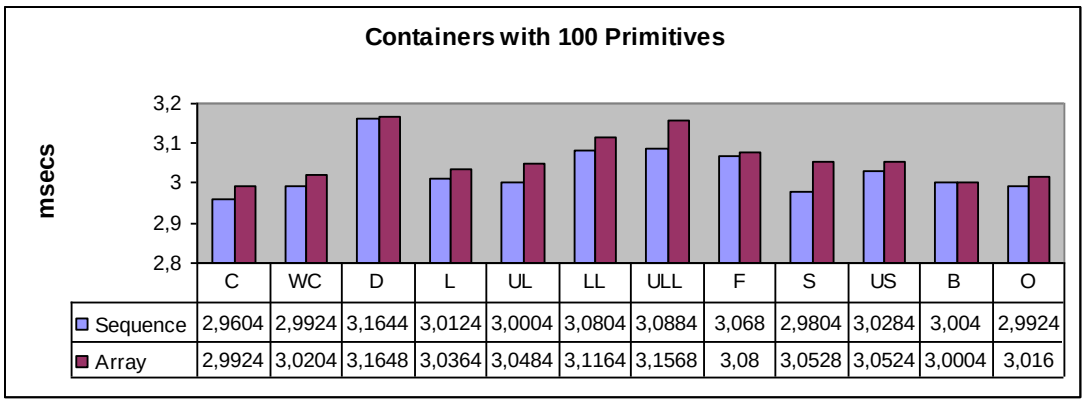
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Primitive	2,9764	2,9884	3,024	2,932	2,9524	2,9724	2,9684	3,0004	2,9684	3,0124	3,3528	2,916
Sequence	2,9684	2,9684	2,9844	2,9404	2,94	2,9564	2,98	2,9964	2,9404	3,008	2,9444	3,3
Array	3,0004	3,0284	3,068	2,984	2,976	2,9884	3,0236	3,0284	2,996	3,0084	3	3,3724

Figure 5.22 : L_T_OS Results for Primitives and Containers with 1 Primitive



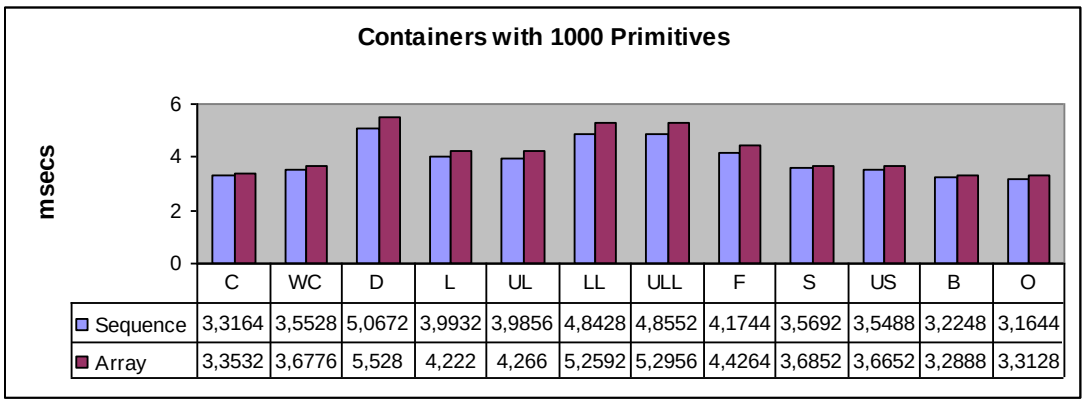
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,9684	2,9324	2,9996	2,988	2,9564	2,9724	2,988	3,0004	2,932	2,9604	2,9404	3,0124
Array	2,9764	2,96	3,04	3,0004	2,9684	3,0244	3,0164	2,9924	2,9804	2,9924	2,9844	3,0084

Figure 5.23 : L_T_OS Results for Containers with 10 Primitives



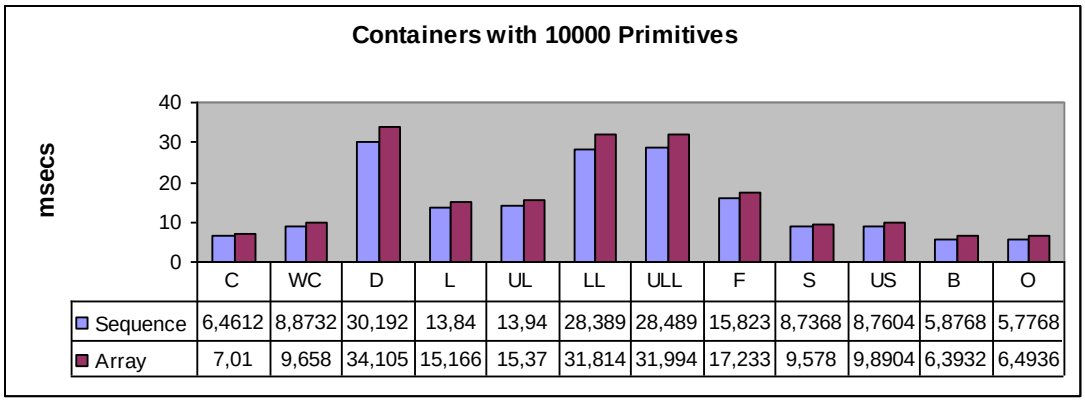
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,9604	2,9924	3,1644	3,0124	3,0004	3,0804	3,0884	3,068	2,9804	3,0284	3,004	2,9924
Array	2,9924	3,0204	3,1648	3,0364	3,0484	3,1164	3,1568	3,08	3,0528	3,0524	3,0004	3,016

Figure 5.24 : L_T_OS Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,3164	3,5528	5,0672	3,9932	3,9856	4,8428	4,8552	4,1744	3,5692	3,5488	3,2248	3,1644
Array	3,3532	3,6776	5,528	4,222	4,266	5,2592	5,2956	4,4264	3,6852	3,6652	3,2888	3,3128

Figure 5.25 : L_T_OS Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	6,4612	8,8732	30,192	13,84	13,94	28,389	28,489	15,823	8,7368	8,7604	5,8768	5,7768
Array	7,01	9,658	34,105	15,166	15,37	31,814	31,994	17,233	9,578	9,8904	6,3932	6,4936

Figure 5.26 : L_T_OS Results for Containers with 10000 Primitives

5.3.3.2 about L_T_OS primitive and primitive container results

Some conclusions from the results are :

- We have nearly same results for arrays and sequences (with possible 10% deviation).
- If we compare with L_O_OS results of same category we see that oneway results are faster for small sizes, but when we come to size 1000 and 10000 we get the nearly same results.

5.3.3.3 L_T_OS string and wide string results

Figure 5.27 shows the results obtained.

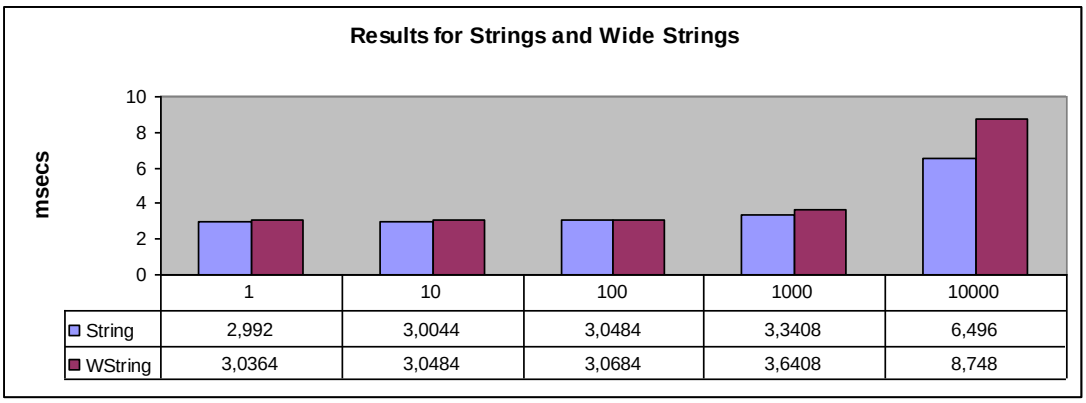


Figure 5.27 : L_T_OS Results for Strings and Wide Strings

5.3.3.4 about L_T_OS string and wide string results

Some conclusions from the results are :

- For strings and wstrings we have nearly the same results with primitive arrays and sequences of type char and wchar, respectively.
- If we compare with L_O_OS results, we see that for the small lengths, oneway calls are faster. But when size increases, the difference decreases and for the length of 10000, twoway shows better performance than oneway.

5.3.3.5 L_T_OS struct and struct container results

Figures 5.28 through 5.32 shows the results obtained.

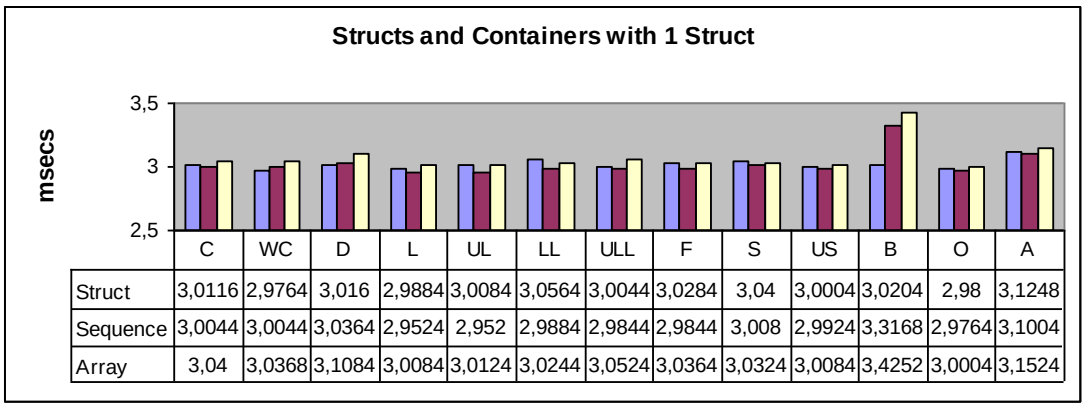
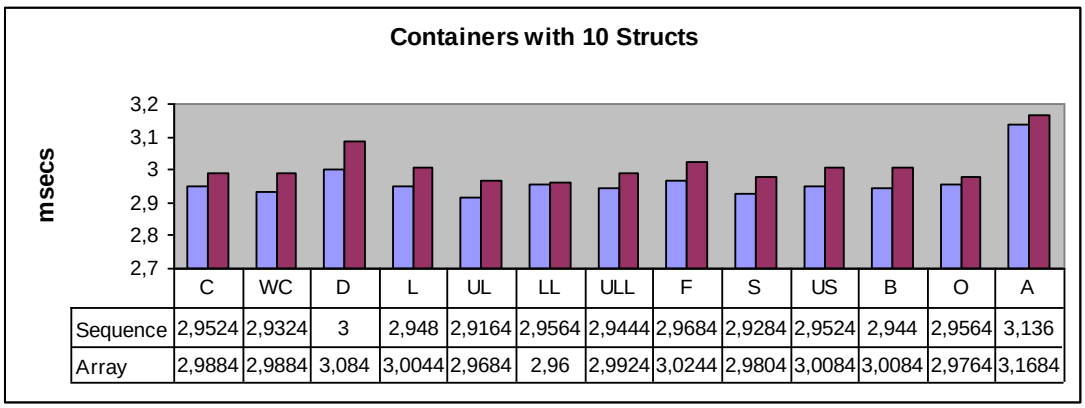
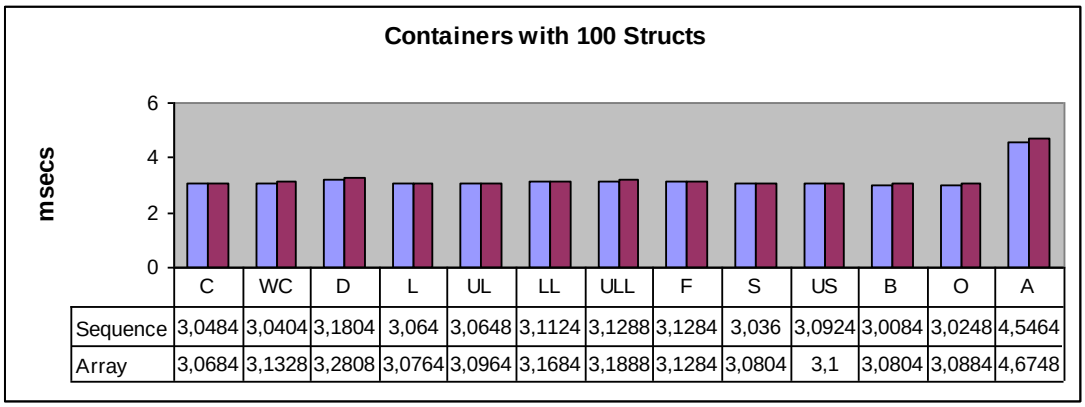


Figure 5.28 : L_T_OS Results for Structs and Containers with 1 Struct



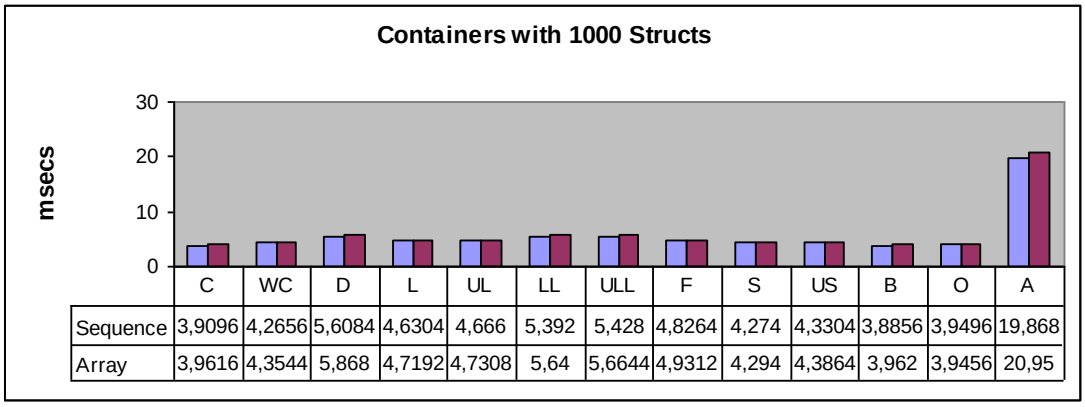
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,9524	2,9324	3	2,948	2,9164	2,9564	2,9444	2,9684	2,9284	2,9524	2,944	2,9564	3,136
Array	2,9884	2,9884	3,084	3,0044	2,9684	2,96	2,9924	3,0244	2,9804	3,0084	3,0084	2,9764	3,1684

Figure 5.29 : L_T_OS Results for Containers with 10 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,0484	3,0404	3,1804	3,064	3,0648	3,1124	3,1288	3,1284	3,036	3,0924	3,0084	3,0248	4,5464
Array	3,0684	3,1328	3,2808	3,0764	3,0964	3,1684	3,1888	3,1284	3,0804	3,1	3,0804	3,0884	4,6748

Figure 5.30 : L_T_OS Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,9096	4,2656	5,6084	4,6304	4,666	5,392	5,428	4,8264	4,274	4,3304	3,8856	3,9496	19,868
Array	3,9616	4,3544	5,868	4,7192	4,7308	5,64	5,6644	4,9312	4,294	4,3864	3,962	3,9456	20,95

Figure 5.31 : L_T_OS Results for Containers with 1000 Structs

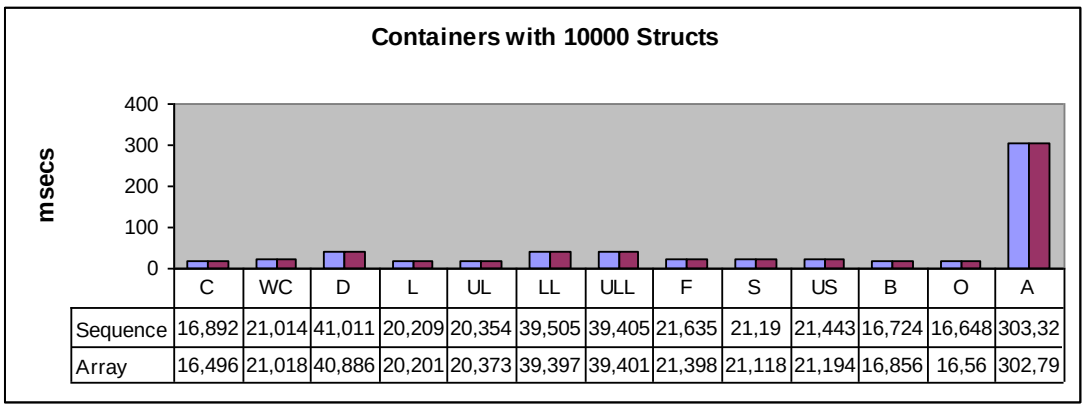


Figure 5.32 : L_T_OS Results for Containers with 10000 Structs

5.3.3.6 about L_T_OS struct and struct container results

Some conclusions from the results are :

- We have the nearly same results for sequences and arrays.
- If we compare the results with results of L_O_OS we see that for the small sizes (1,10, 100) oneway results are better than twoways. For the larger sizes they are nearly equal (size 1000 except our private all struct). Then for size 10000 it has two features in it : for primitive structs twoway is faster, and for our private all struct oneway is faster.

5.3.3.7 L_T_OS interface and interface container results

Figures 5.33 through 5.37 shows the results obtained.

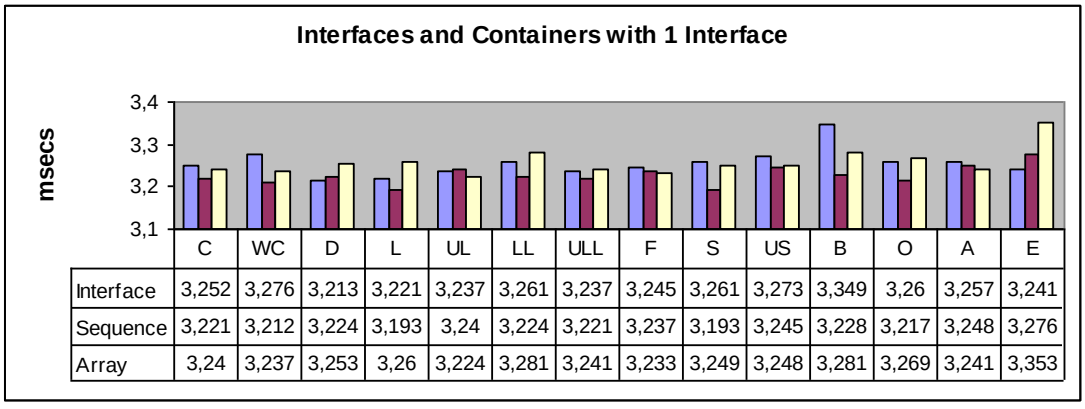
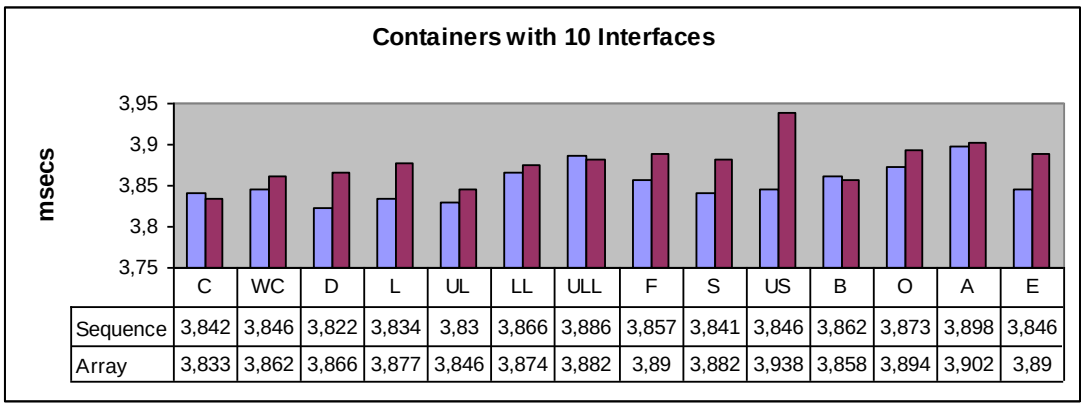
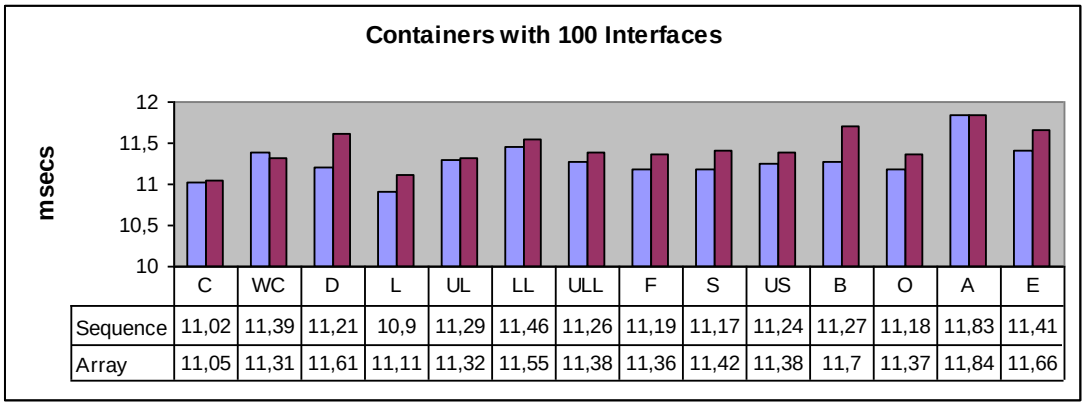


Figure 5.33 : L_T_OS Results for Interface and Containers with 1 Interface



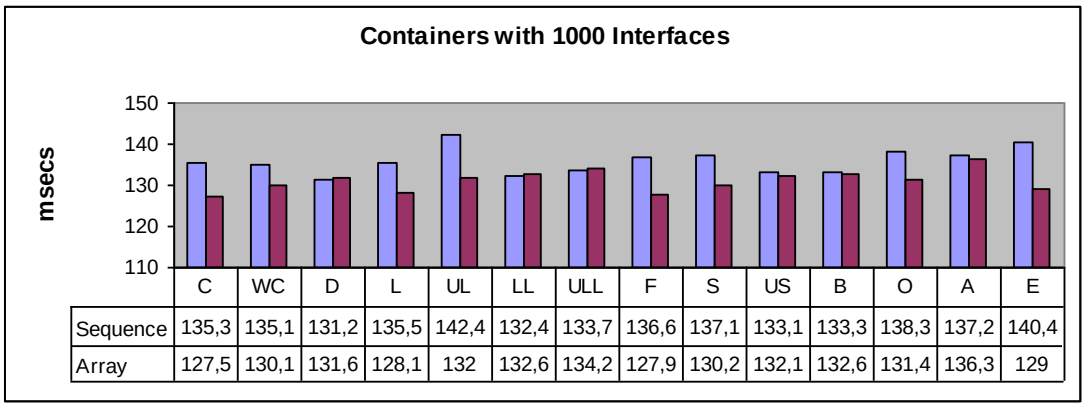
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	3,842	3,846	3,822	3,834	3,83	3,866	3,886	3,857	3,841	3,846	3,862	3,873	3,898	3,846
Array	3,833	3,862	3,866	3,877	3,846	3,874	3,882	3,89	3,882	3,938	3,858	3,894	3,902	3,89

Figure 5.34 : L_T_OS Results for Containers with 10 Interfaces



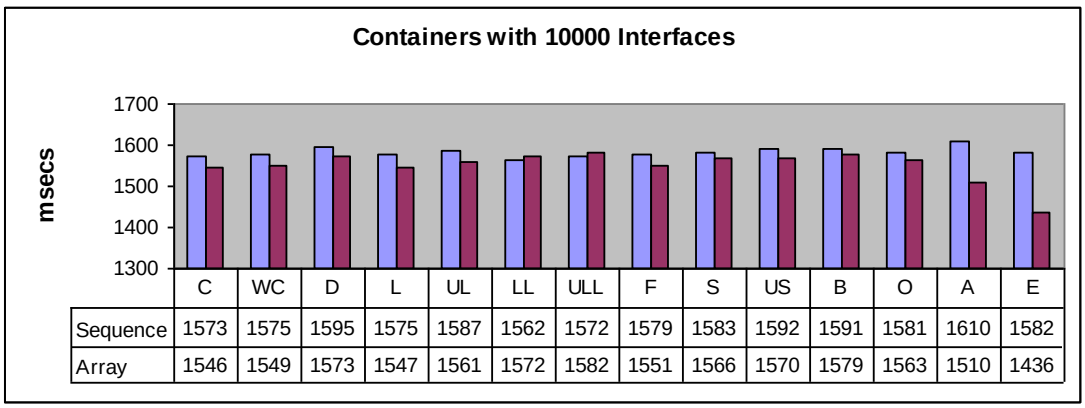
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	11,02	11,39	11,21	10,9	11,29	11,46	11,26	11,19	11,17	11,24	11,27	11,18	11,83	11,41
Array	11,05	11,31	11,61	11,11	11,32	11,55	11,38	11,36	11,42	11,38	11,7	11,37	11,84	11,66

Figure 5.35 : L_T_OS Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	135,3	135,1	131,2	135,5	142,4	132,4	133,7	136,6	137,1	133,1	133,3	138,3	137,2	140,4
Array	127,5	130,1	131,6	128,1	132	132,6	134,2	127,9	130,2	132,1	132,6	131,4	136,3	129

Figure 5.36 : L_T_OS Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	1573	1575	1595	1575	1587	1562	1572	1579	1583	1592	1591	1581	1610	1582
Array	1546	1549	1573	1547	1561	1572	1582	1551	1566	1570	1579	1563	1510	1436

Figure 5.37 : L_T_OS Results for Containers with 10000 Interfaces

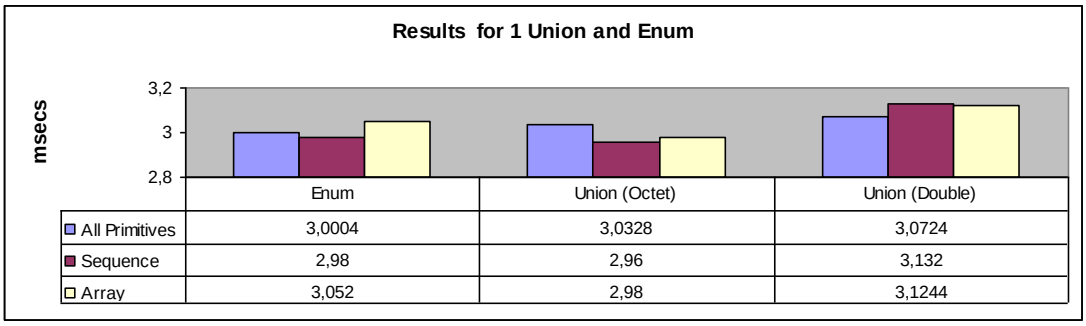
5.3.3.8 about L_T_OS interface and interface container results

Some conclusions from the results are :

- Arrays and sequences performs nearly the same
- If we compare with L_O_OS results we see that for the size of 100, twoway is faster than oneway, but at size 1, 10, 1000 oneway is faster.
- L_O_OS with size 10000 could not complete the test. But with twoway we could completed it here.

5.3.3.9 union and enum results

Figures 5.38 through 5.42 shows the results obtained.



	Enum	Union (Octet)	Union (Double)
All Primitives	3,0004	3,0328	3,0724
Sequence	2,98	2,96	3,132
Array	3,052	2,98	3,1244

Figure 5.38 : L_T_OS Results for Union, Enum and Containers with 1 Union and Enum

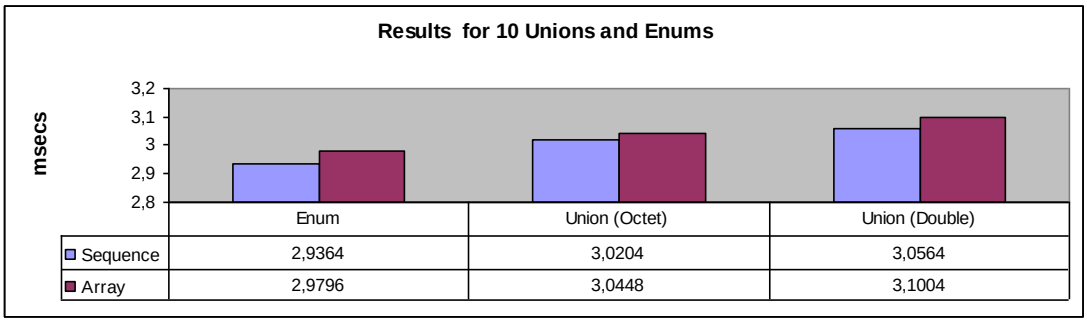


Figure 5.39 : L_T_OS Results for Containers with 10 Unions and Enums

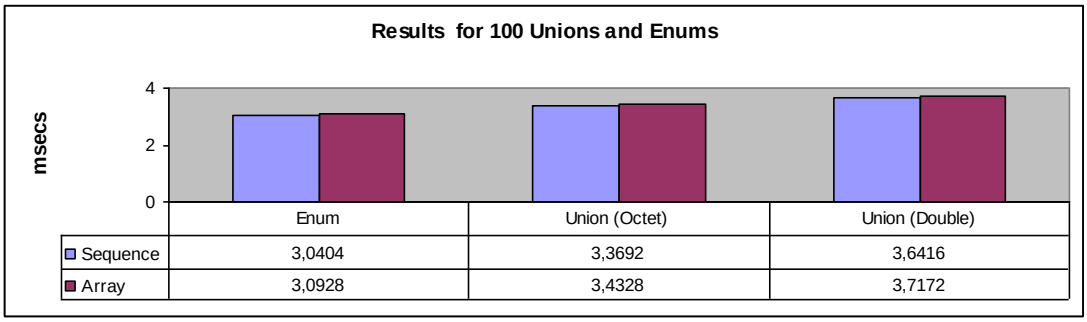


Figure 5.40 : L_T_OS Results for Containers with 100 Unions and Enums

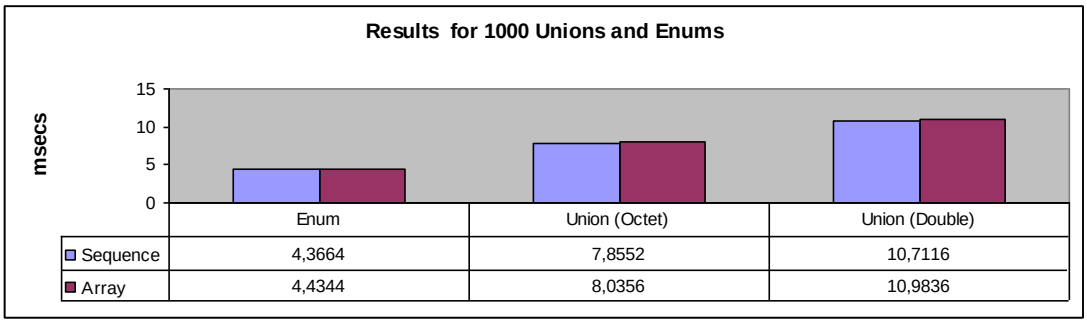


Figure 5.41 : L_T_OS Results for Containers with 1000 Unions and Enums

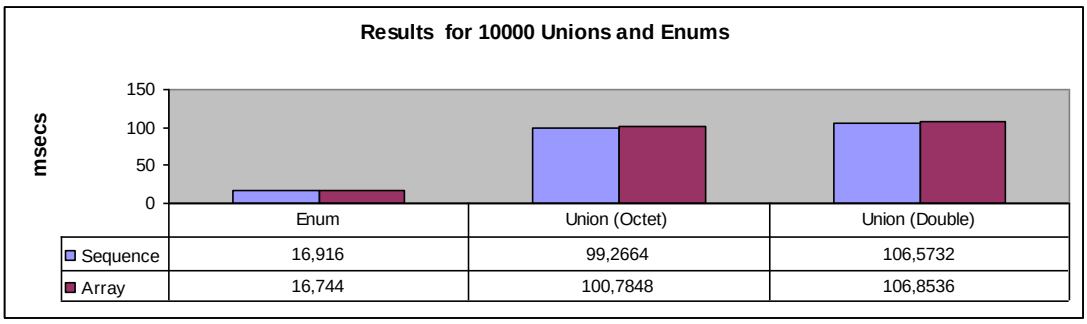


Figure 5.42 : L_T_OS Results for Containers with 10000 Unions and Enums

5.3.3.10 about L_T_OS union and enum results

Some conclusions from the results are :

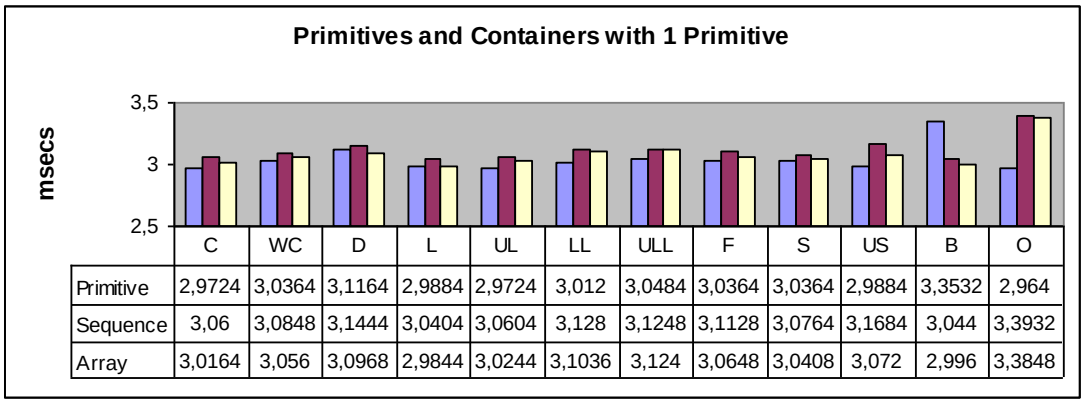
- Arrays and sequences performs nearly the same
- If we compare with L_O_OS results we see that for the size of 100, twoway is faster than oneway, but at size 1, 10, 1000 oneway is faster.

5.3.4 twoway – send get results

We will briefly refer to these results as L_T_SG (Local_Twoway_SendGet) results.

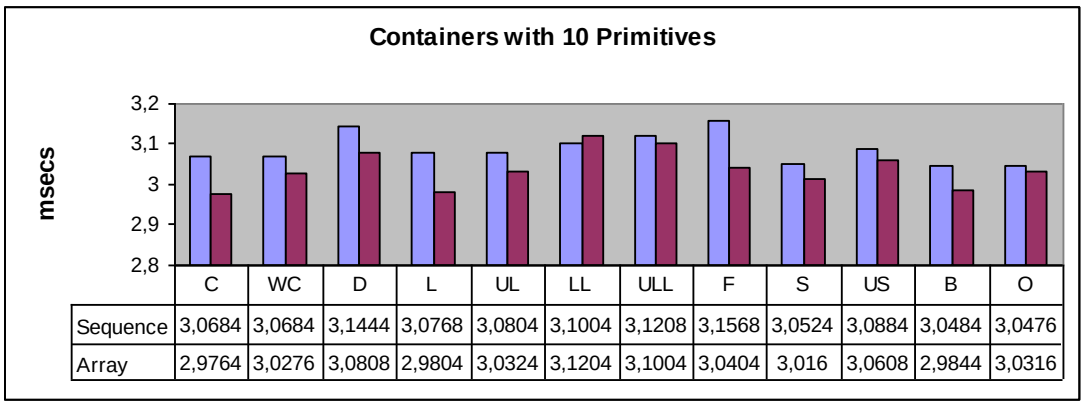
5.3.4.1 L_T_SG primitive and primitive container results

Figure 5.43 through 5.47 shows the results obtained for primitive types and containers with primitive types.



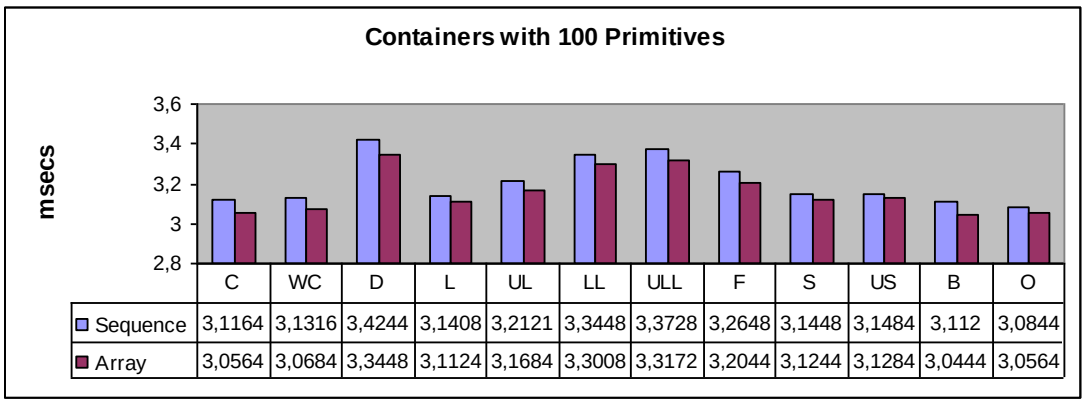
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Primitive	2,9724	3,0364	3,1164	2,9884	2,9724	3,012	3,0484	3,0364	3,0364	2,9884	3,3532	2,964
Sequence	3,06	3,0848	3,1444	3,0404	3,0604	3,128	3,1248	3,1128	3,0764	3,1684	3,044	3,3932
Array	3,0164	3,056	3,0968	2,9844	3,0244	3,1036	3,124	3,0648	3,0408	3,072	2,996	3,3848

Figure 5.43 : L_T_SG Results for Primitives and Containers with 1 Primitive



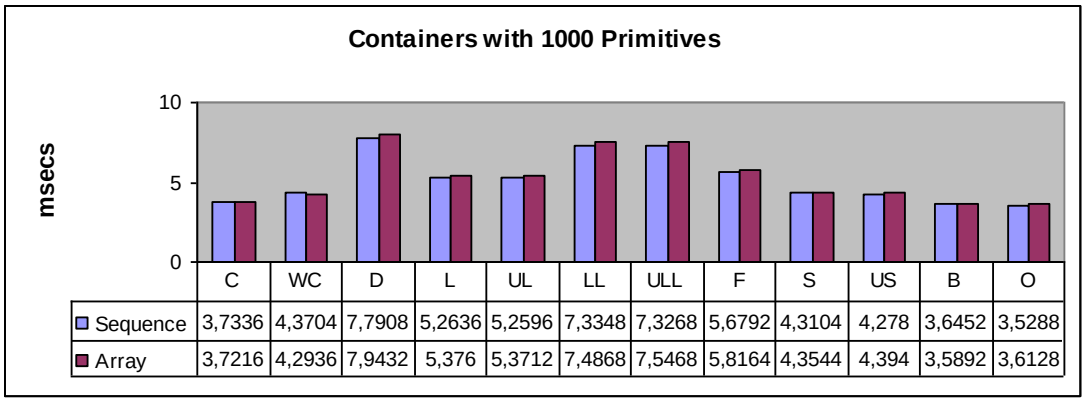
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,0684	3,0684	3,1444	3,0768	3,0804	3,1004	3,1208	3,1568	3,0524	3,0884	3,0484	3,0476
Array	2,9764	3,0276	3,0808	2,9804	3,0324	3,1204	3,1004	3,0404	3,016	3,0608	2,9844	3,0316

Figure 5.44 : L_T_SG Results for Containers with 10 Primitives



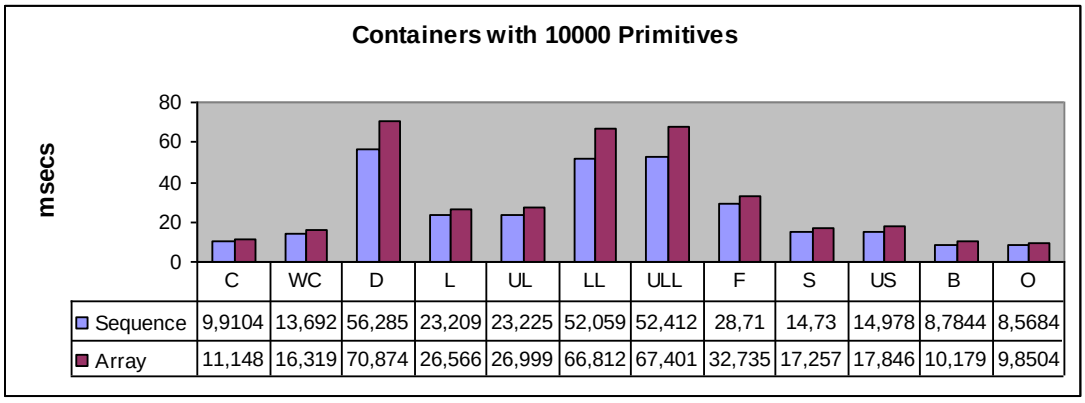
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,1164	3,1316	3,4244	3,1408	3,2121	3,3448	3,3728	3,2648	3,1448	3,1484	3,112	3,0844
Array	3,0564	3,0684	3,3448	3,1124	3,1684	3,3008	3,3172	3,2044	3,1244	3,1284	3,0444	3,0564

Figure 5.45 : L_T_SG Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,7336	4,3704	7,7908	5,2636	5,2596	7,3348	7,3268	5,6792	4,3104	4,278	3,6452	3,5288
Array	3,7216	4,2936	7,9432	5,376	5,3712	7,4868	7,5468	5,8164	4,3544	4,394	3,5892	3,6128

Figure 5.46 : L_T_SG Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	9,9104	13,692	56,285	23,209	23,225	52,059	52,412	28,71	14,73	14,978	8,7844	8,5684
Array	11,148	16,319	70,874	26,566	26,999	66,812	67,401	32,735	17,257	17,846	10,179	9,8504

Figure 5.47 : L_T_SG Results for Containers with 10000 Primitives

5.3.4.2 about L_T_SG primitive and primitive container results

Some conclusions from the results are :

- One primitive, primitive within a sequence and primitive within an array performs the same.

- Sequences are a bit faster than the arrays for the size 10000. I can't deduce anything from this result. In the worst case, arrays must have been faster than sequences.

- If we compare with L_T_OS results, we see that sending a type is nearly the same with L_T_SG for small sizes (1, 10, 100) and becomes apparent when size increases (1000, 10000). We would expect the L_T_SG to performt slower all the times from L_T_OS since there is two paths of data flow. But it is not the case with small sizes. I guess this is because of the fact that small sized data can be carried in a packet and the acknowledgement can be piggybacked with this packet. Twoway calls always block for an acknowledgement that specifies the successive end of the call. If the time consumed on the server side to copy data and send back is small, which is the case for small-sized data, then these results can be very close. For the big data we lose our time in coping with more than one packet and copying of return results to be sent back from the server.

5.3.4.3 L_T_SG string and wide string results

Figure 5.48 shows the results obtained.

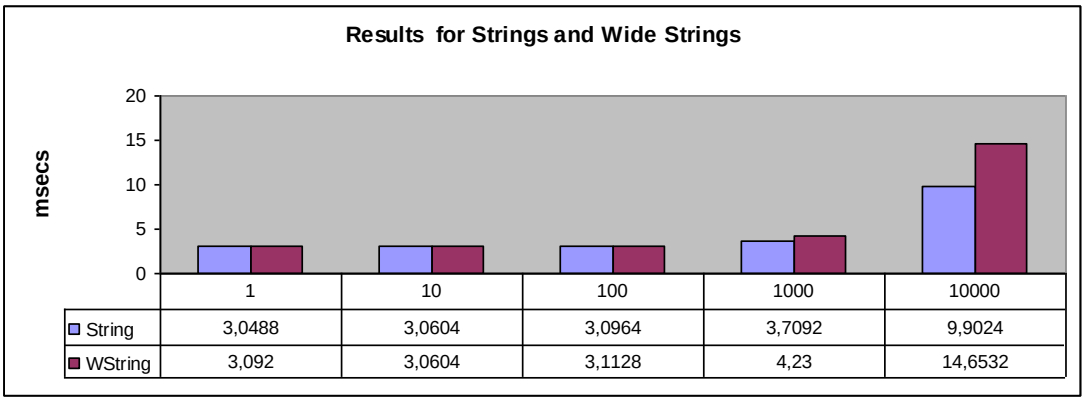


Figure 5.48 : L_T_SG Results for Strings and Wide Strings

5.3.4.4 about L_T_SG string and wide string results

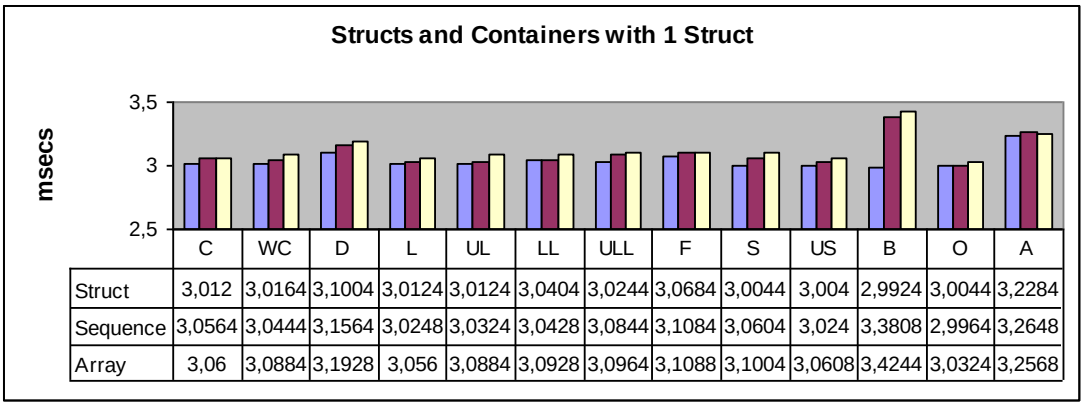
Some conclusions from the results are :

- Strings and WStrings exhibits the same performance characteristics with char and wchar sequences, respectively. And they are faster than the arrays of their respective types.

- If we compare with L_T_OS results we see that the difference between sending and sending/getting the (w)strings becomes apparent after size 1000.

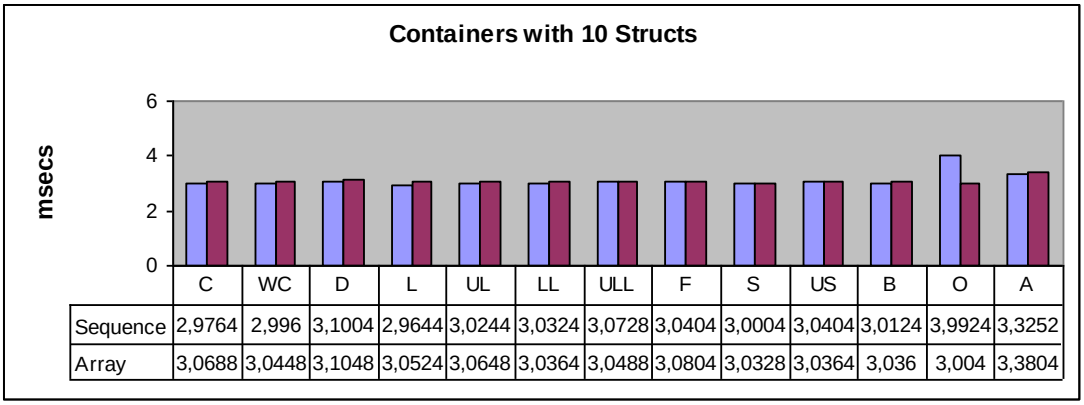
5.3.4.5 L_T_SG struct and struct container results

Figures 5.49 through 5.53 shows the results obtained.



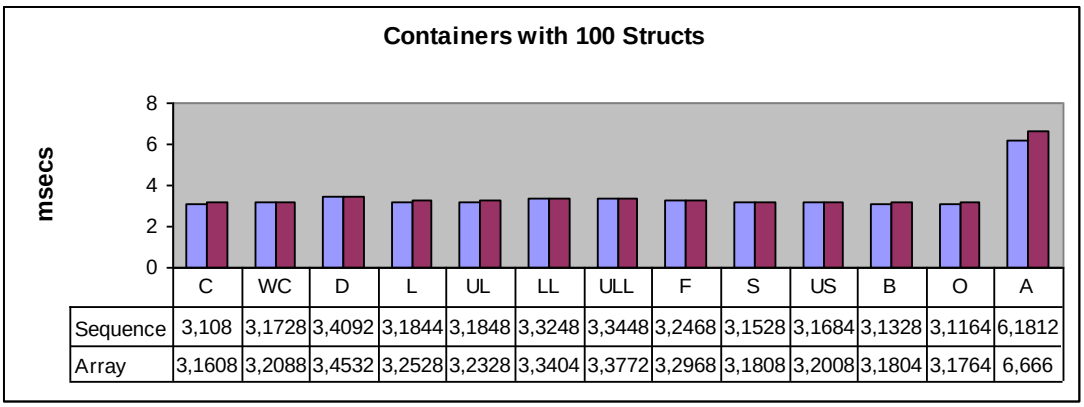
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Struct	3,012	3,0164	3,1004	3,0124	3,0124	3,0404	3,0244	3,0684	3,0044	3,004	2,9924	3,0044	3,2284
Sequence	3,0564	3,0444	3,1564	3,0248	3,0324	3,0428	3,0844	3,1084	3,0604	3,024	3,3808	2,9964	3,2648
Array	3,06	3,0884	3,1928	3,056	3,0884	3,0928	3,0964	3,1088	3,1004	3,0608	3,4244	3,0324	3,2568

Figure 5.49 : L_T_SG Results for Structs and Containers with 1 Struct



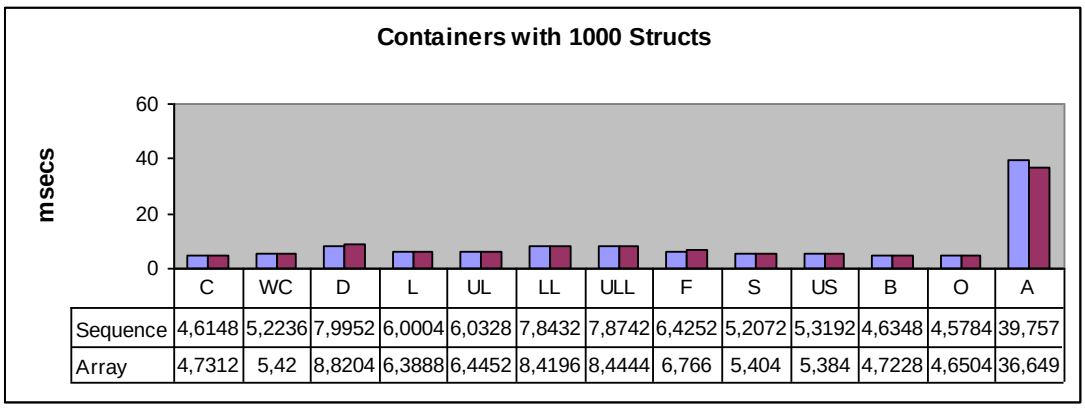
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,9764	2,996	3,1004	2,9644	3,0244	3,0324	3,0728	3,0404	3,0004	3,0404	3,0124	3,9924	3,3252
Array	3,0688	3,0448	3,1048	3,0524	3,0648	3,0364	3,0488	3,0804	3,0328	3,0364	3,036	3,004	3,3804

Figure 5.50 : L_T_SG Results for Containers with 10 Structs



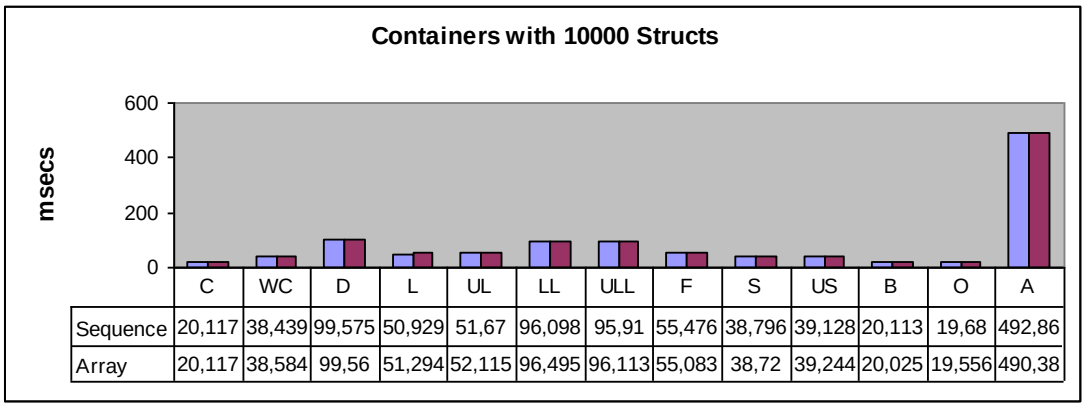
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,108	3,1728	3,4092	3,1844	3,1848	3,3248	3,3448	3,2468	3,1528	3,1684	3,1328	3,1164	6,1812
Array	3,1608	3,2088	3,4532	3,2528	3,2328	3,3404	3,3772	3,2968	3,1808	3,2008	3,1804	3,1764	6,666

Figure 5.51 : L_T_SG Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	4,6148	5,2236	7,9952	6,0004	6,0328	7,8432	7,8742	6,4252	5,2072	5,3192	4,6348	4,5784	39,757
Array	4,7312	5,42	8,8204	6,3888	6,4452	8,4196	8,4444	6,766	5,404	5,384	4,7228	4,6504	36,649

Figure 5.52 : L_T_SG Results for Containers with 1000 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	20,117	38,439	99,575	50,929	51,67	96,098	95,91	55,476	38,796	39,128	20,113	19,68	492,86
Array	20,117	38,584	99,56	51,294	52,115	96,495	96,113	55,083	38,72	39,244	20,025	19,556	490,38

Figure 5.53 : L_T_SG Results for Containers with 10000 Structs

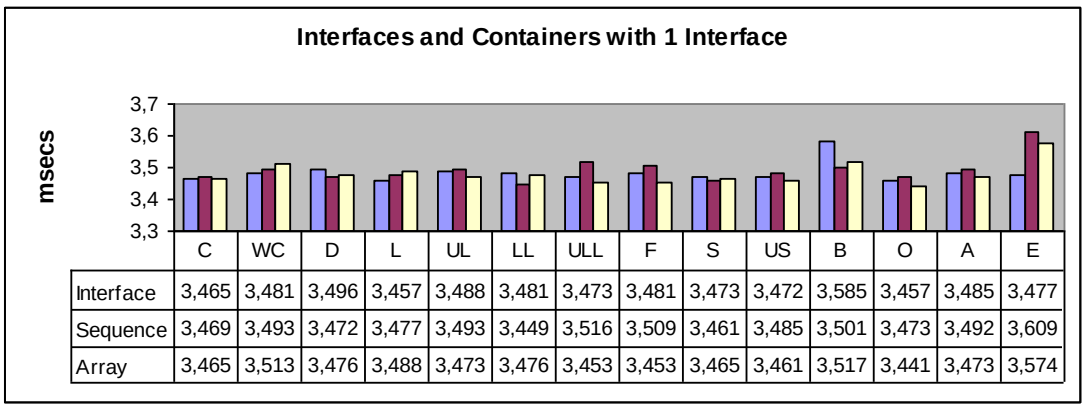
5.3.4.6 about L_T_SG struct and struct container results

Some conclusions from the results are :

- To carry a primitive alone, within a struct or within a container with 1 struct is same.
- There is no difference between carrying the structs with arrays or sequences.
- If we compare with L_T_OS results we see that there is difference for sizes 1, 10 and 100. After size 1000 difference becomes to appear.

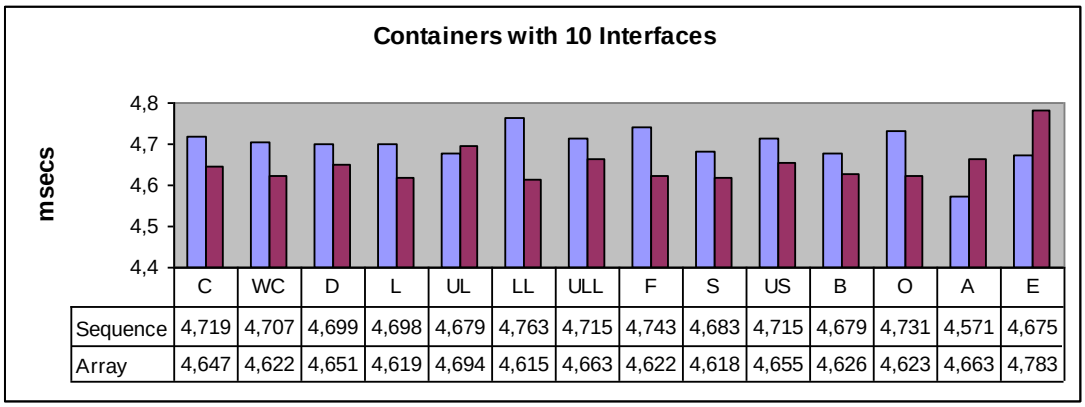
5.3.4.7 L_T_SG interface and interface container results

Figures 5.54 through 5.58 shows the results obtained.



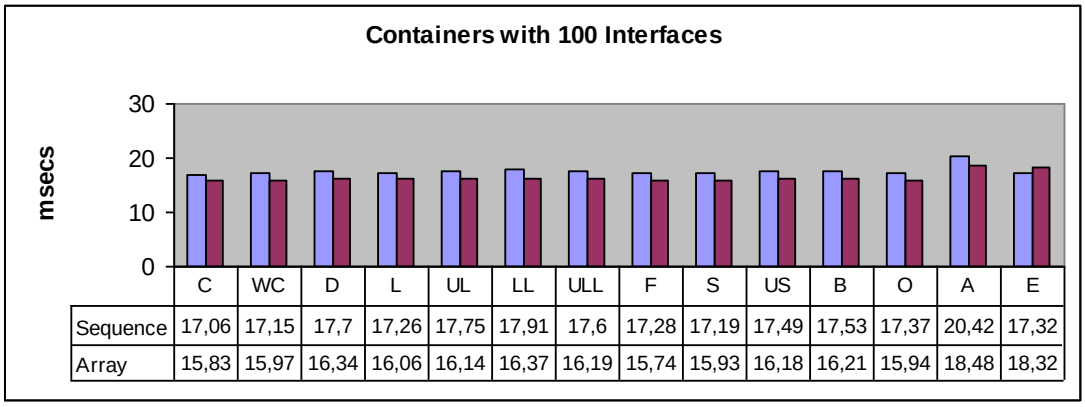
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	3,465	3,481	3,496	3,457	3,488	3,481	3,473	3,481	3,473	3,472	3,585	3,457	3,485	3,477
Sequence	3,469	3,493	3,472	3,477	3,493	3,449	3,516	3,509	3,461	3,485	3,501	3,473	3,492	3,609
Array	3,465	3,513	3,476	3,488	3,473	3,476	3,453	3,453	3,465	3,461	3,517	3,441	3,473	3,574

Figure 5.54 : L_T_SG Results for Interface and Containers with 1 Interface



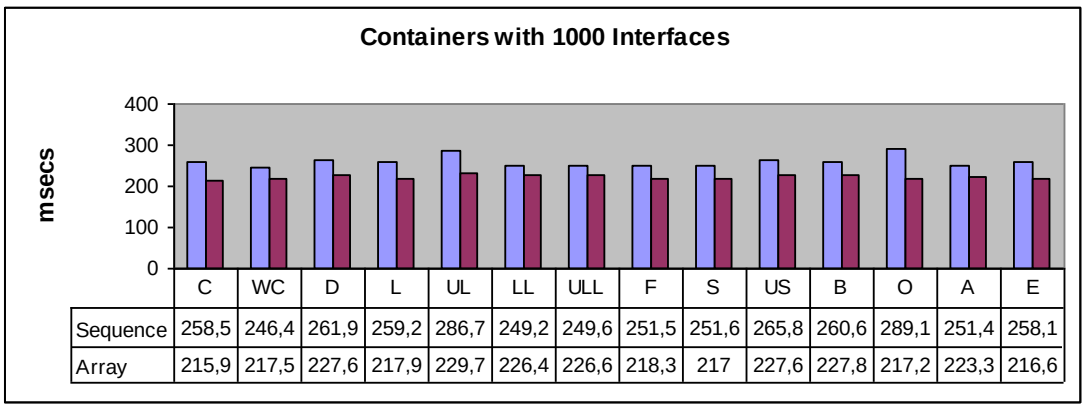
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	4,719	4,707	4,699	4,698	4,679	4,763	4,715	4,743	4,683	4,715	4,679	4,731	4,571	4,675
Array	4,647	4,622	4,651	4,619	4,694	4,615	4,663	4,622	4,618	4,655	4,626	4,623	4,663	4,783

Figure 5.55 : L_T_SG Results for Containers with 10 Interfaces



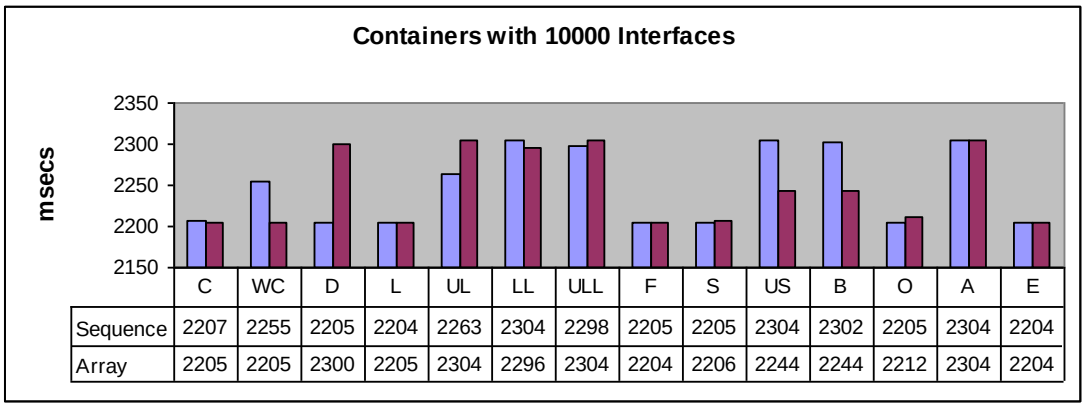
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	17,06	17,15	17,7	17,26	17,75	17,91	17,6	17,28	17,19	17,49	17,53	17,37	20,42	17,32
Array	15,83	15,97	16,34	16,06	16,14	16,37	16,19	15,74	15,93	16,18	16,21	15,94	18,48	18,32

Figure 5.56 : L_T_SG Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	258,5	246,4	261,9	259,2	286,7	249,2	249,6	251,5	251,6	265,8	260,6	289,1	251,4	258,1
Array	215,9	217,5	227,6	217,9	229,7	226,4	226,6	218,3	217	227,6	227,8	217,2	223,3	216,6

Figure 5.57 : L_T_SG Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	2207	2255	2205	2204	2263	2304	2298	2205	2205	2304	2302	2205	2304	2204
Array	2205	2205	2300	2205	2304	2296	2304	2204	2206	2244	2244	2212	2304	2204

Figure 5.58 : L_T_SG Results for Containers with 10000 Interfaces

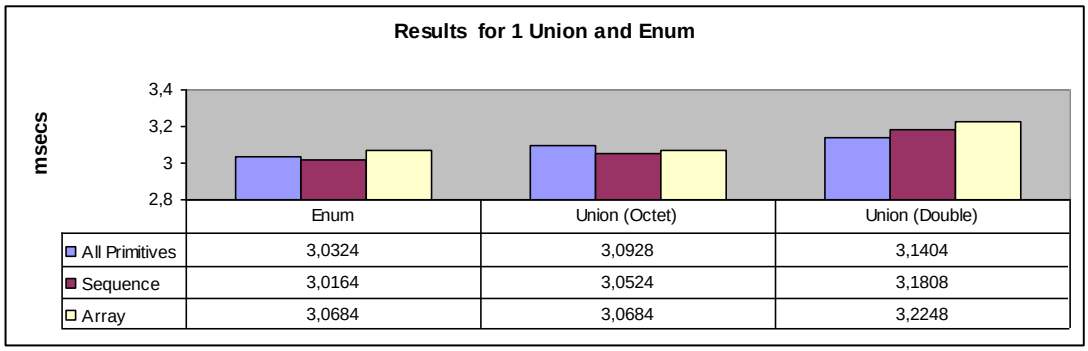
5.3.4.8 about L_T_SG interface and interface container results

Some conclusions from the results are :

- To carry a primitive within an interface is slower than carrying it alone.
- There is no difference between sequences and arrays.
- Sharp changes between results begin very early, with size 10. This shows that interfaces requires big memory areas.
- If we compare with L_T_OS results we see that even for the only one interface there is difference between only sending and sending/getting the interfaces.

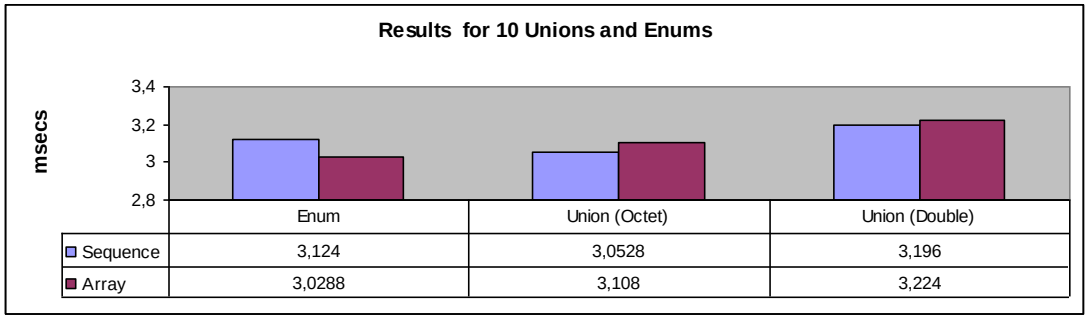
5.3.4.9 L_T_SG union and enum results

Figures 5.59 through 5.63 shows the results obtained.



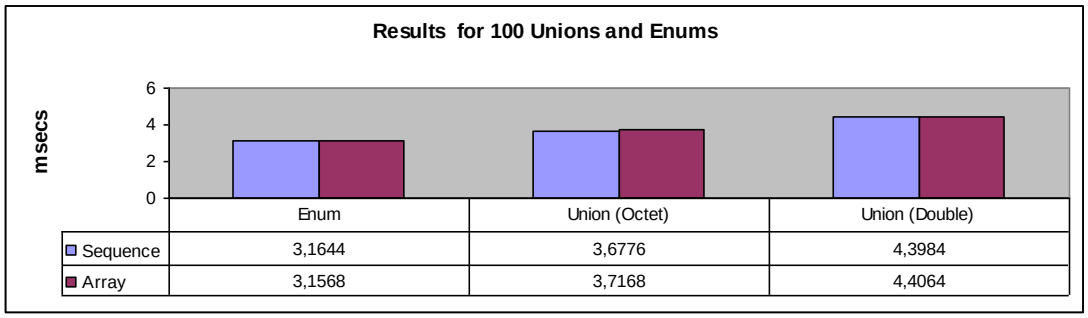
	Enum	Union (Octet)	Union (Double)
All Primitives	3,0324	3,0928	3,1404
Sequence	3,0164	3,0524	3,1808
Array	3,0684	3,0684	3,2248

Figure 5.59 : L_T_SG Results for Union, Enum and Containers with 1 Union and Enum



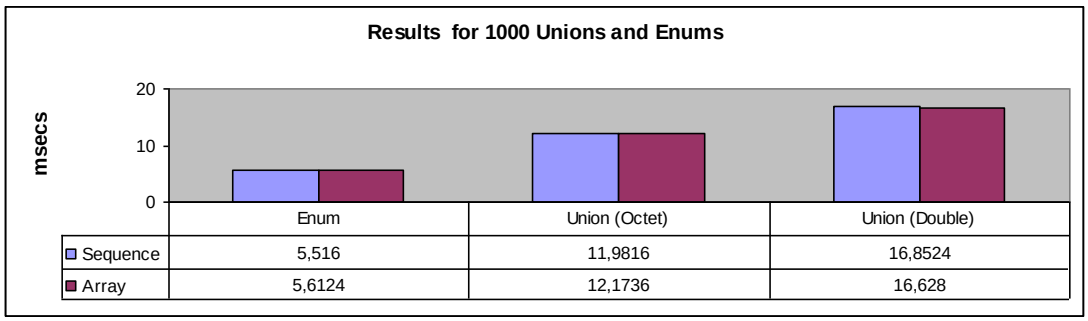
	Enum	Union (Octet)	Union (Double)
Sequence	3,124	3,0528	3,196
Array	3,0288	3,108	3,224

Figure 5.60 : L_T_SG Results for Containers with 10 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	3,1644	3,6776	4,3984
Array	3,1568	3,7168	4,4064

Figure 5.61 : L_T_SG Results for Containers with 100 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	5,516	11,9816	16,8524
Array	5,6124	12,1736	16,628

Figure 5.62 : L_T_SG Results for Containers with 1000 Unions and Enums

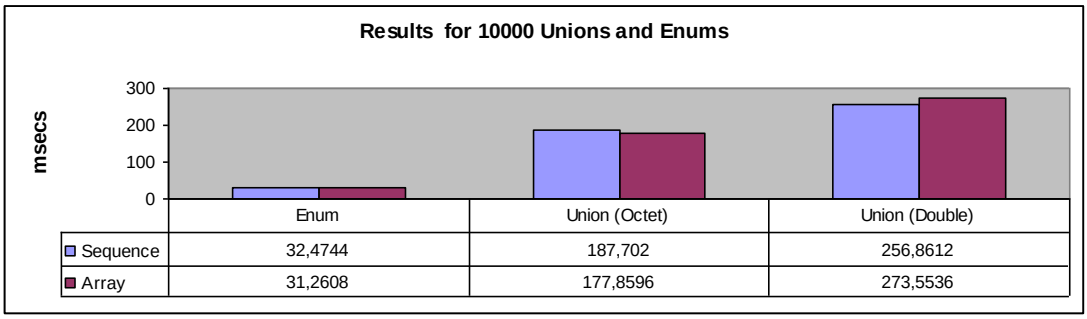


Figure 5.63 : L_T_SG Results for Containers with 10000 Unions and Enums

5.3.4.10 about L_T_SG union and enum results

Some conclusions from the results are :

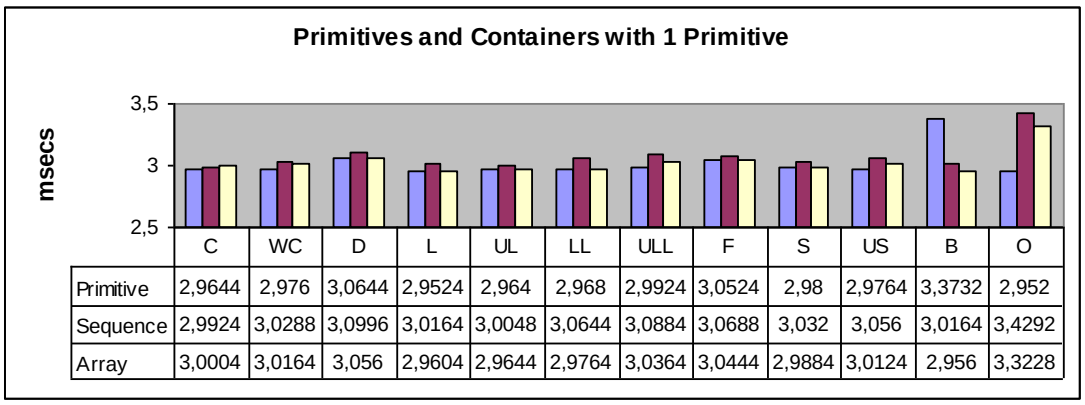
- Enum shows the same characteristics with unsigned longs generally. But it is faster for size 10000 from unsigned longs.
- Enum and union Arrays and sequences perform nearly the same as expected.
- Difeerence between unions with doubles and unions with octets are obvious for sizes bigger than 10.
- If we compare with L_T_OS results we see that for the enum they almost show the same performance up to size 1000, and for the union up to size 100.

5.3.5 twoway – only get results

We will briefly refer to these results as L_T_OG (Local_Twoway_OnlyGet) results.

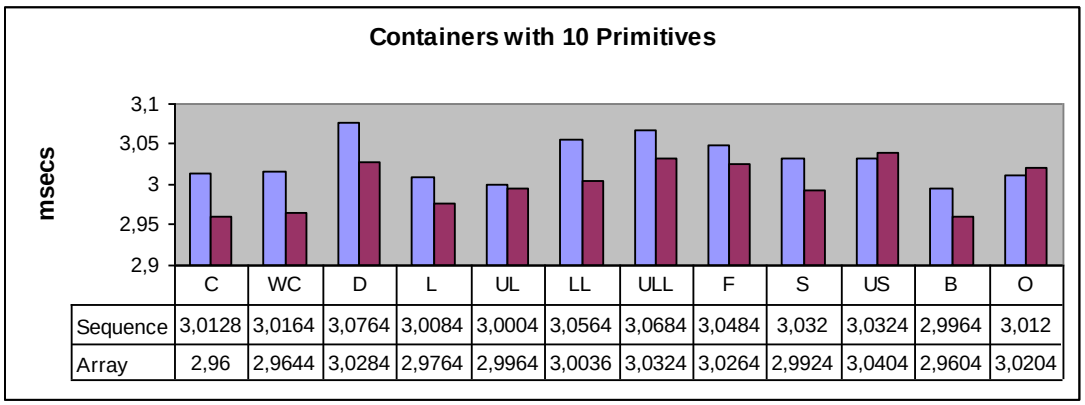
5.3.5.1 L_T_OG primitive and primitive container results

Figure 5.64 through 5.68 shows the results obtained for primitive types and containers with primitive types.



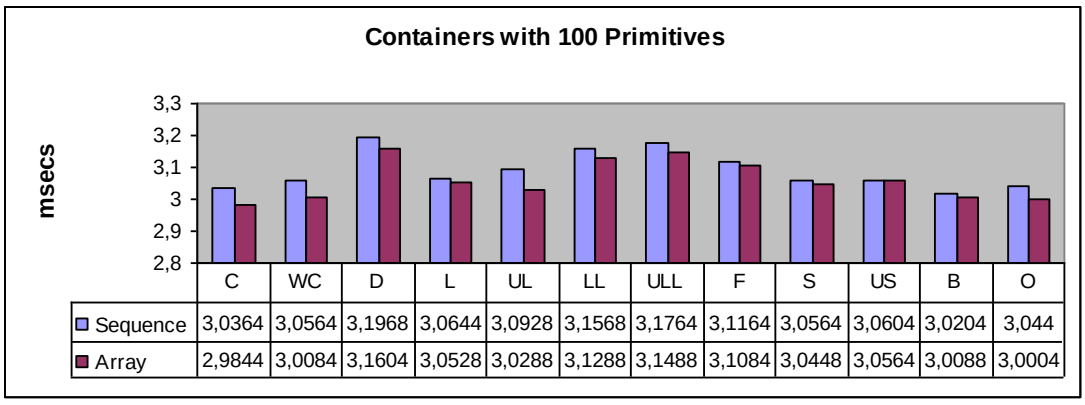
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Primitive	2,9644	2,976	3,0644	2,9524	2,964	2,968	2,9924	3,0524	2,98	2,9764	3,3732	2,952
Sequence	2,9924	3,0288	3,0996	3,0164	3,0048	3,0644	3,0884	3,0688	3,032	3,056	3,0164	3,4292
Array	3,0004	3,0164	3,056	2,9604	2,9644	2,9764	3,0364	3,0444	2,9884	3,0124	2,956	3,3228

Figure 5.64 : L_T_OG Results for Primitives and Containers with 1 Primitive



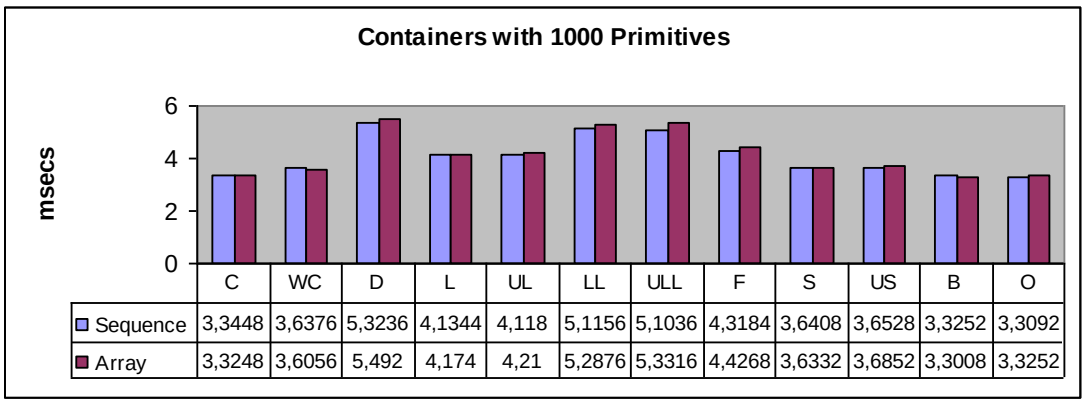
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,0128	3,0164	3,0764	3,0084	3,0004	3,0564	3,0684	3,0484	3,032	3,0324	2,9964	3,012
Array	2,96	2,9644	3,0284	2,9764	2,9964	3,0036	3,0324	3,0264	2,9924	3,0404	2,9604	3,0204

Figure 5.65 : L_T_OG Results for Containers with 10 Primitives



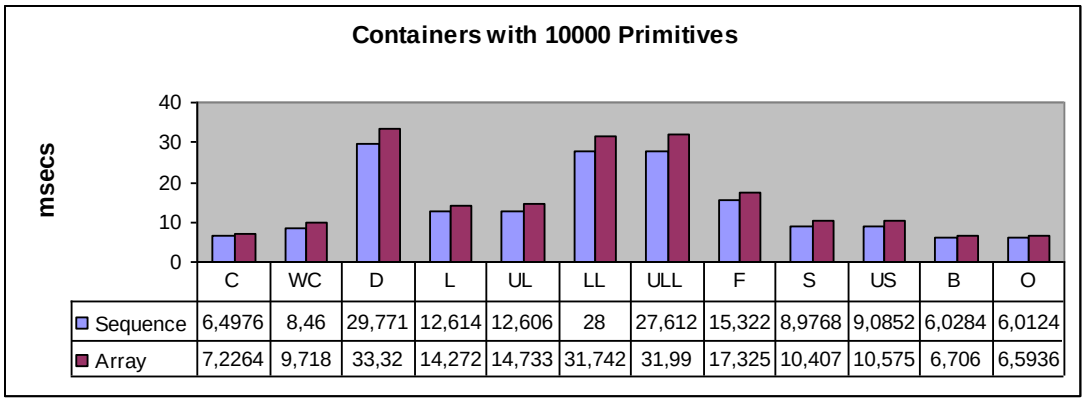
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,0364	3,0564	3,1968	3,0644	3,0928	3,1568	3,1764	3,1164	3,0564	3,0604	3,0204	3,044
Array	2,9844	3,0084	3,1604	3,0528	3,0288	3,1288	3,1488	3,1084	3,0448	3,0564	3,0088	3,0004

Figure 5.66 : L_T_OG Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,3448	3,6376	5,3236	4,1344	4,118	5,1156	5,1036	4,3184	3,6408	3,6528	3,3252	3,3092
Array	3,3248	3,6056	5,492	4,174	4,21	5,2876	5,3316	4,4268	3,6332	3,6852	3,3008	3,3252

Figure 5.67 : L_T_OG Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	6,4976	8,46	29,771	12,614	12,606	28	27,612	15,322	8,9768	9,0852	6,0284	6,0124
Array	7,2264	9,718	33,32	14,272	14,733	31,742	31,99	17,325	10,407	10,575	6,706	6,5936

Figure 5.68 : L_T_OG Results for Containers with 10000 Primitives

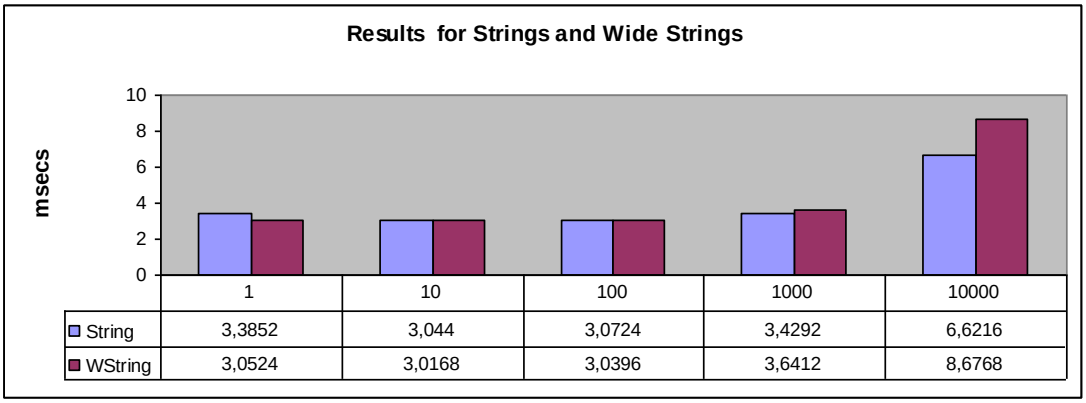
5.3.5.2 about L_T_OG primitive and primitive container results

Some conclusions from the results are :

- Primitives alone and primitives in a container performs same.
- Arrays and sequences perform nearly the same. Arrays are a bit better than sequences, but it can be ignored.
- If we compare the results with L_O_SG results we see that for the small sizes they ara nearly equal. When calling a method we pass to the ORB (or more truely to the Object Adapter) the name and parameters of the method. So, when we have small sizes of parameters, we can fit them into a packet and can send the method name and parameters within a packet. But when parameter sizes increases the number of packets also increases.
- If we compare the results with L_T_OS, we see that they perform almost the same.

5.3.5.3 L_T_OG string and wide string results

Figure 5.69 shows the results obtained.



	1	10	100	1000	10000
String	3,3852	3,044	3,0724	3,4292	6,6216
WString	3,0524	3,0168	3,0396	3,6412	8,6768

Figure 5.69 : L_T_OG Results for Strings and Wide Strings

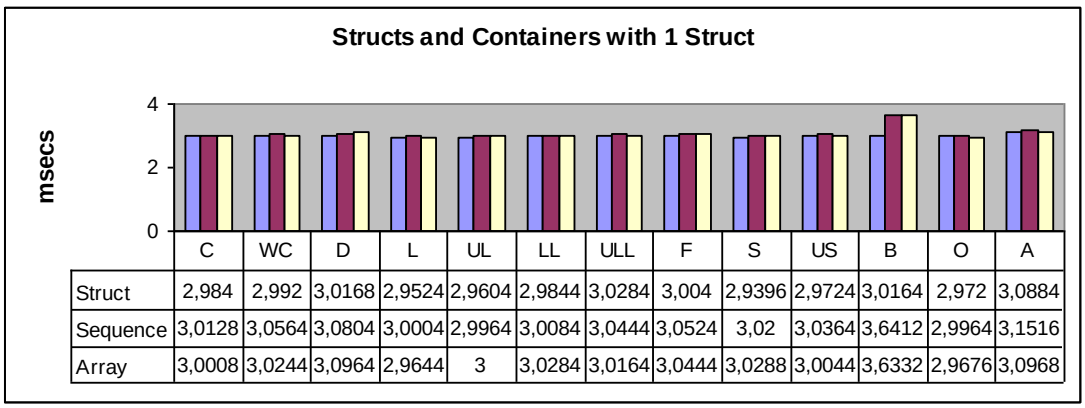
5.3.5.4 about L_T_OG string and wide string results

Some conclusions from the results are :

- Encapsulating char and wchar within containers and strings performs the same.
- If we compare with L_T_OS results we see that the results are nearly same.
- If we compare with L_T_SG results we see that they are nearly the same for sizes 1, 10 and 100 and L_T_SG becomes slower and slower for sizes 1000 and 10000.

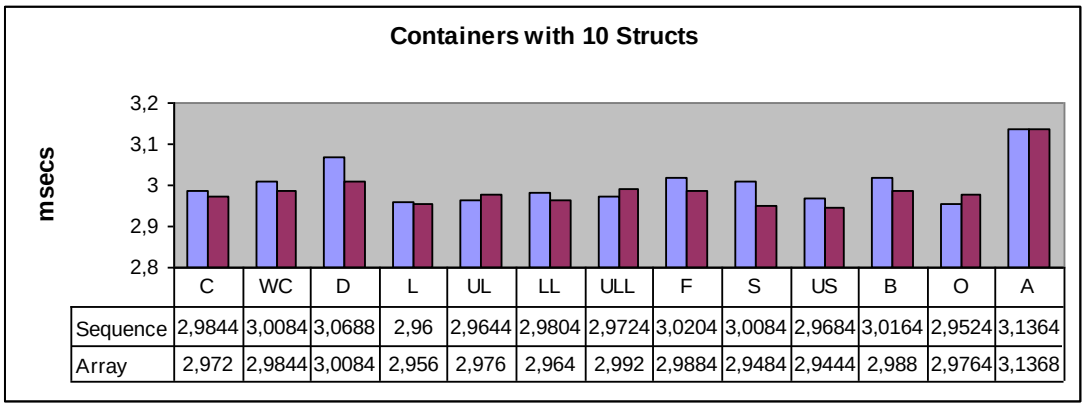
5.3.5.5 L_T_OG struct and struct container results

Figures 5.70 through 5.74 shows the results obtained.



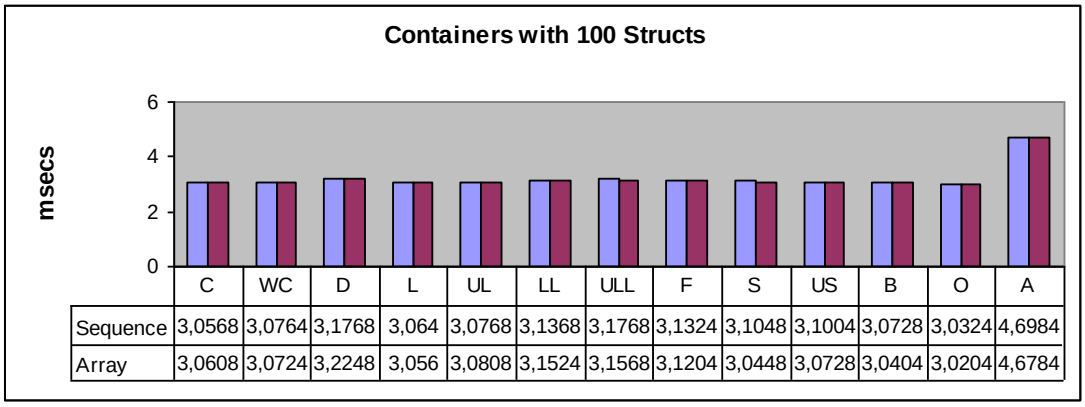
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Struct	2,984	2,992	3,0168	2,9524	2,9604	2,9844	3,0284	3,004	2,9396	2,9724	3,0164	2,972	3,0884
Sequence	3,0128	3,0564	3,0804	3,0004	2,9964	3,0084	3,0444	3,0524	3,02	3,0364	3,6412	2,9964	3,1516
Array	3,0008	3,0244	3,0964	2,9644	3	3,0284	3,0164	3,0444	3,0288	3,0044	3,6332	2,9676	3,0968

Figure 5.70 : L_T_OG Results for Structs and Containers with 1 Struct



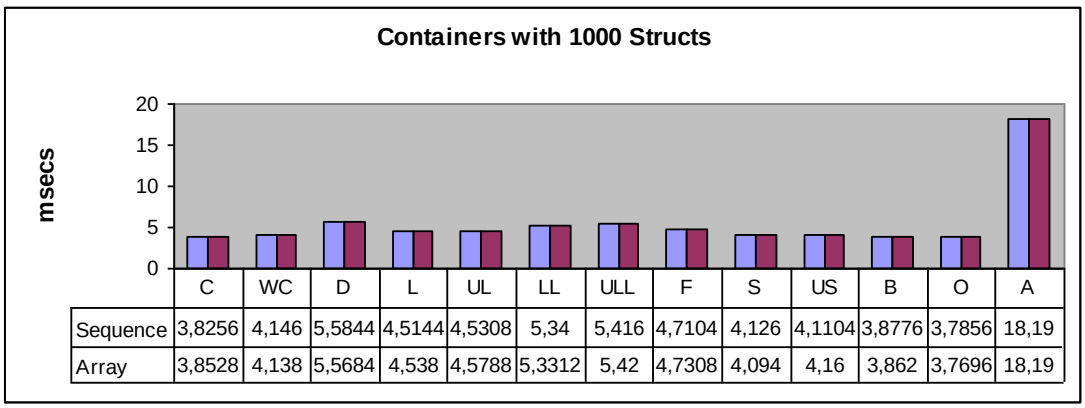
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,9844	3,0084	3,0688	2,96	2,9644	2,9804	2,9724	3,0204	3,0084	2,9684	3,0164	2,9524	3,1364
Array	2,972	2,9844	3,0084	2,956	2,976	2,964	2,992	2,9884	2,9484	2,9444	2,988	2,9764	3,1368

Figure 5.71 : L_T_OG Results for Containers with 10 Structs.



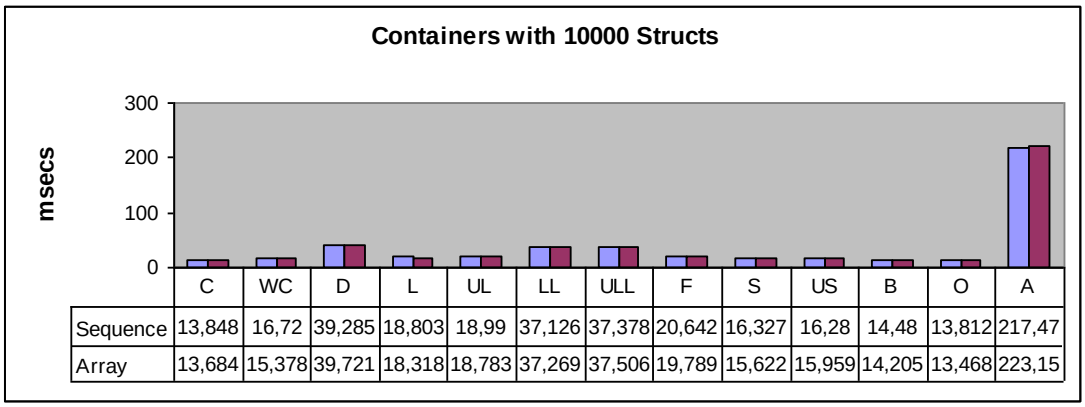
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,0568	3,0764	3,1768	3,064	3,0768	3,1368	3,1768	3,1324	3,1048	3,1004	3,0728	3,0324	4,6984
Array	3,0608	3,0724	3,2248	3,056	3,0808	3,1524	3,1568	3,1204	3,0448	3,0728	3,0404	3,0204	4,6784

Figure 5.72 : L_T_OG Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,8256	4,146	5,5844	4,5144	4,5308	5,34	5,416	4,7104	4,126	4,1104	3,8776	3,7856	18,19
Array	3,8528	4,138	5,5684	4,538	4,5788	5,3312	5,42	4,7308	4,094	4,16	3,862	3,7696	18,19

Figure 5.73 : L_T_OG Results for Containers with 1000 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	13,848	16,72	39,285	18,803	18,99	37,126	37,378	20,642	16,327	16,28	14,48	13,812	217,47
Array	13,684	15,378	39,721	18,318	18,783	37,269	37,506	19,789	15,622	15,959	14,205	13,468	223,15

Figure 5.74 : L_T_OG Results for Containers with 10000 Structs

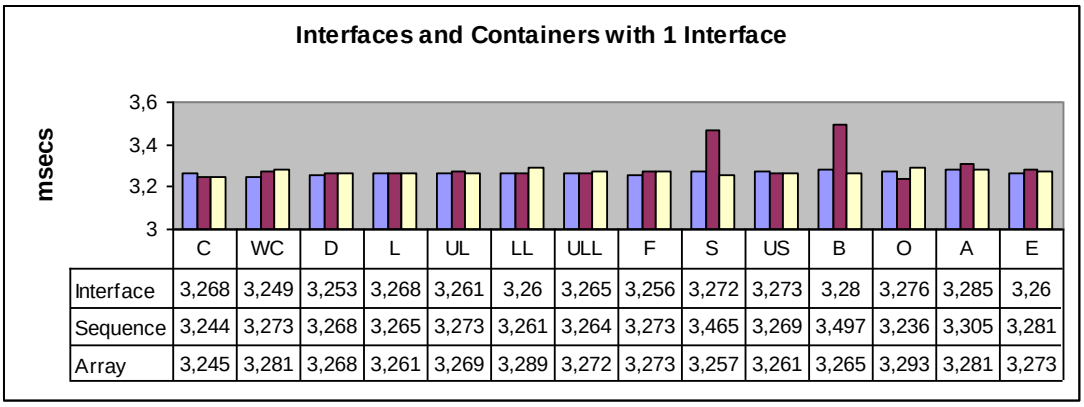
5.3.5.6 about L_T_OG struct and struct container results

Some conclusions from the results are :

- To carry a primitive alone, within a struct or within a container with 1 struct is same.
- There is no difference between carrying the structs within arrays or sequences.
- If we compare with L_T_OS results we see that we have the same performance for small sizes. But L_T_OG is faster as obviously seen from our special struct with size 10000, and slightly seen from the other results.
- If we compare our results with L_T_SG we see that it is slower than L_T_OG results especially for greater sizes.

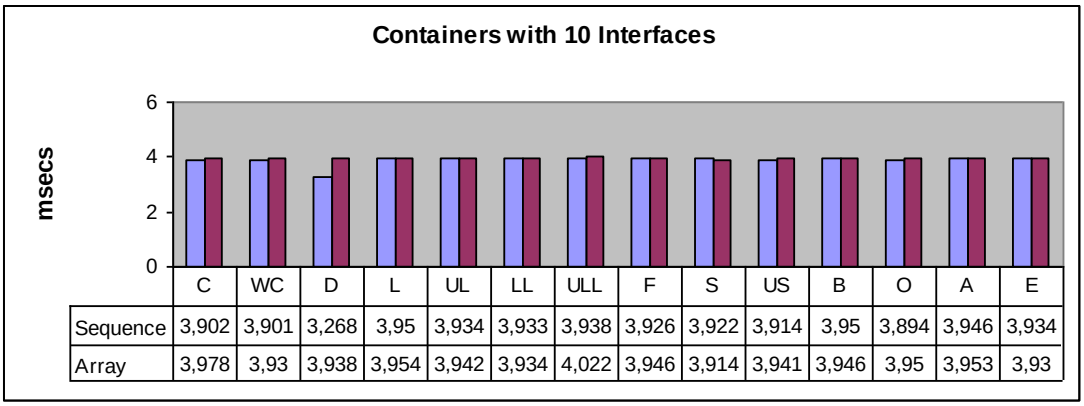
5.3.5.7 L_T_OG interface and interface container results

Figures 5.75 through 5.79 shows the results obtained.



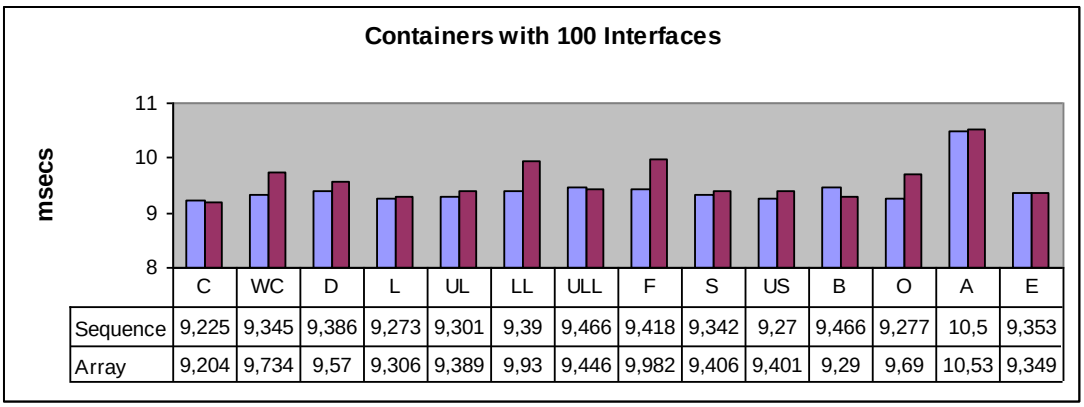
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	3,268	3,249	3,253	3,268	3,261	3,26	3,265	3,256	3,272	3,273	3,28	3,276	3,285	3,26
Sequence	3,244	3,273	3,268	3,265	3,273	3,261	3,264	3,273	3,465	3,269	3,497	3,236	3,305	3,281
Array	3,245	3,281	3,268	3,261	3,269	3,289	3,272	3,273	3,257	3,261	3,265	3,293	3,281	3,273

Figure 5.75 : L_T_OG Results for Interface and Containers with 1 Interface



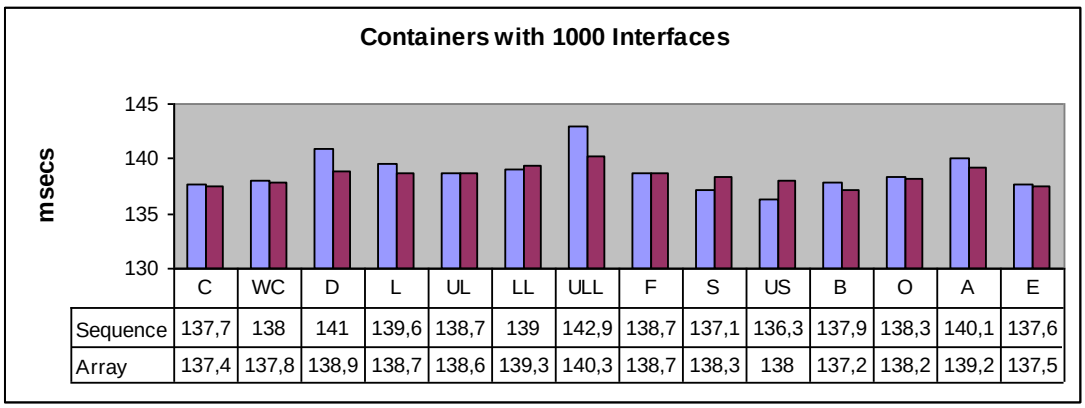
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	3,902	3,901	3,268	3,95	3,934	3,933	3,938	3,926	3,922	3,914	3,95	3,894	3,946	3,934
Array	3,978	3,93	3,938	3,954	3,942	3,934	4,022	3,946	3,914	3,941	3,946	3,95	3,953	3,93

Figure 5.76 : L_T_OG Results for Containers with 10 Interfaces



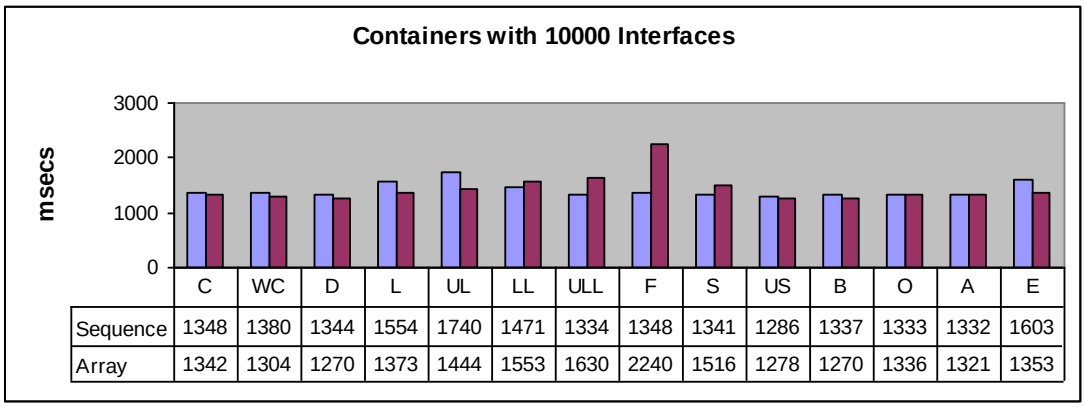
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	9,225	9,345	9,386	9,273	9,301	9,39	9,466	9,418	9,342	9,27	9,466	9,277	10,5	9,353
Array	9,204	9,734	9,57	9,306	9,389	9,93	9,446	9,982	9,406	9,401	9,29	9,69	10,53	9,349

Figure 5.77 : L_T_OG Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	137,7	138	141	139,6	138,7	139	142,9	138,7	137,1	136,3	137,9	138,3	140,1	137,6
Array	137,4	137,8	138,9	138,7	138,6	139,3	140,3	138,7	138,3	138	137,2	138,2	139,2	137,5

Figure 5.78 : L_T_OG Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	1348	1380	1344	1554	1740	1471	1334	1348	1341	1286	1337	1333	1332	1603
Array	1342	1304	1270	1373	1444	1553	1630	2240	1516	1278	1270	1336	1321	1353

Figure 5.79 : L_T_OG Results for Containers with 10000 Interfaces

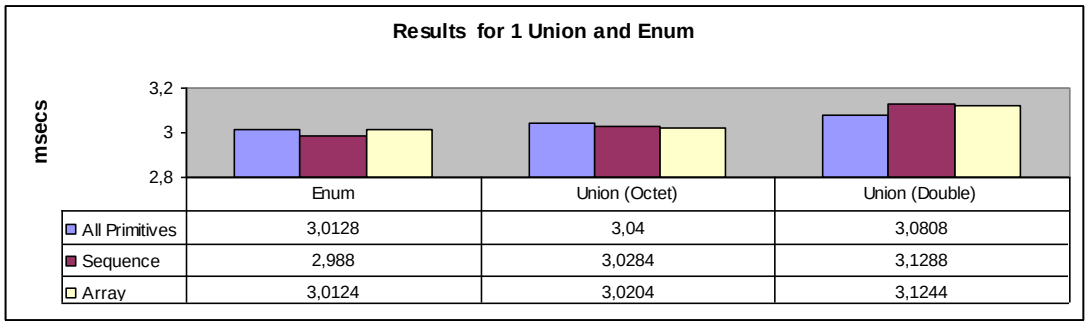
5.3.5.8 about L_T_OG interface and interface container results

Some conclusions from the results are :

- To carry a primitive within an interface or a container with 1 interface is the slowest of all constructed types and have the same performances.
- If we compare with L_T_OS results we see that they have the same performance for sizes 1, 10, 100, L_T_OS is better for size 1000 and L_T_OG is better for size 10000.
- If we compare with L_T_SG we see that L_T_OG is faster for even size 1 and difference grows radically with growing size.

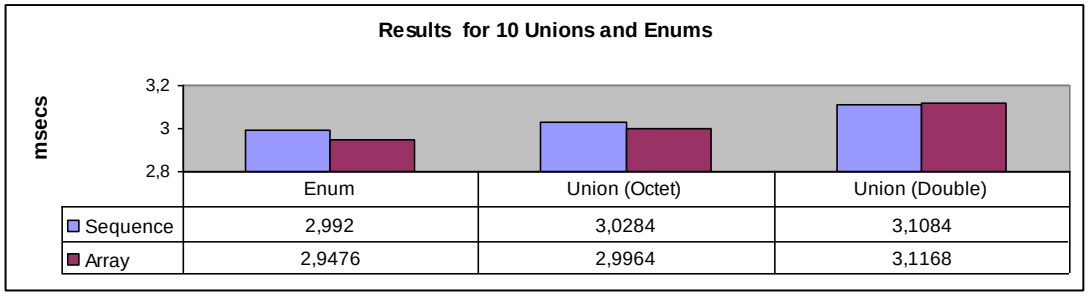
5.3.5.9 L_T_OG union and enum results

Figures 5.80 through 5.84 shows the results obtained.



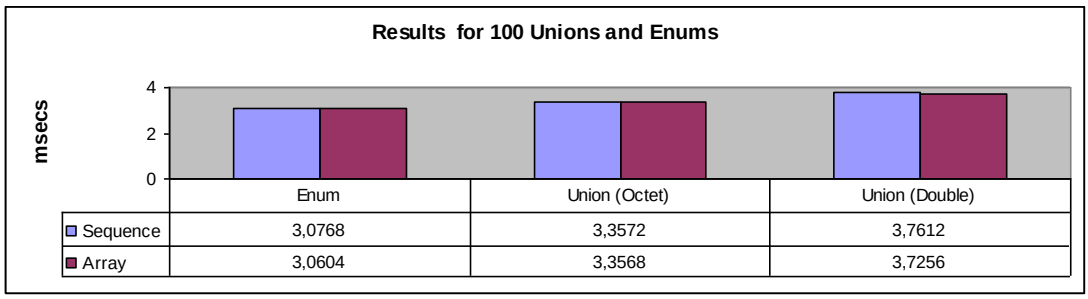
	Enum	Union (Octet)	Union (Double)
All Primitives	3,0128	3,04	3,0808
Sequence	2,988	3,0284	3,1288
Array	3,0124	3,0204	3,1244

Figure 5.80 : L_T_OG Results for Union, Enum and Containers with 1 Union and Enum



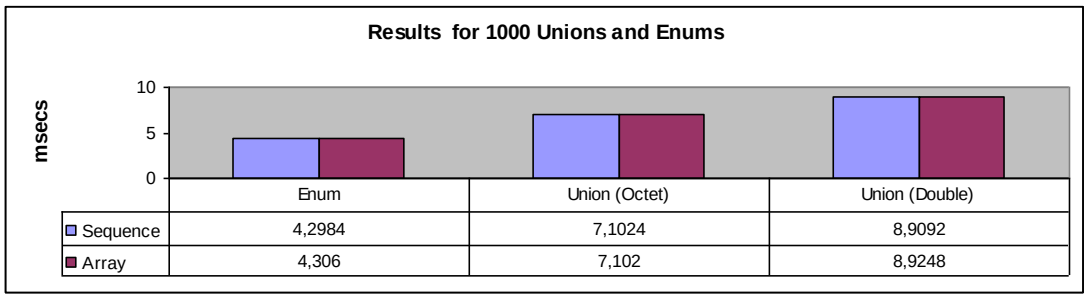
	Enum	Union (Octet)	Union (Double)
Sequence	2,992	3,0284	3,1084
Array	2,9476	2,9964	3,1168

Figure 5.81 : L_T_OG Results for Containers with 10 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	3,0768	3,3572	3,7612
Array	3,0604	3,3568	3,7256

Figure 5.82 : L_T_OG Results for Containers with 100 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	4,2984	7,1024	8,9092
Array	4,306	7,102	8,9248

Figure 5.83 : L_T_OG Results for Containers with 1000 Unions and Enums

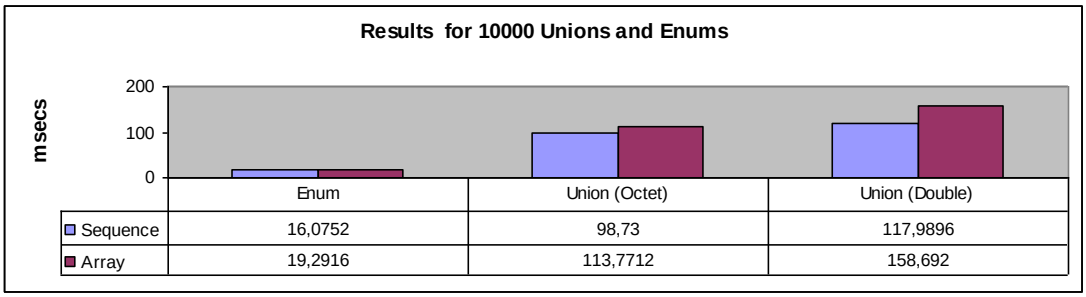


Figure 5.84 : L_T_OG Results for Containers with 10000 Unions and Enums

5.3.5.10 about L_T_OG union and enum results

Some conclusions from the results are :

- Enum shows the same characteristics with unsigned longs generally. But it is slower for size 10000 than unsigned longs.
- Enum and union arrays and sequences performs nearly the same as expected. But for the size 10000 arrays are slower.
- Difference between unions with doubles and unions with octets are obvious for sizes bigger than 100.
- If we compare with L_T_OS results we see that for the enum and union sequences they almost shows the same performance. L_T_OS arrays are a bit slower.
- If we compare with L_T_SG results we see that for the enums there is a clear difference for sizes greater than 1000. Octet unions start to show this difference with size 1000 and double unions with size 100.

5.4 Results for Remote Calls

Following are the results for our remote calls taken as described at section 4.5.3. We must point out the followings, before passing to the results :

- ❖ We mentioned at the local results that we observe a sharp difference between the first call and the second. This sharpness is so prominent for some remote calls (and these anomalies occur randomly) that it affects the average value of results greatly. So, we do not take the first calls' time for the abnormal results for remote calls. You can expect a long time with respect to

others for first calls (our results show that it is about 20,000 msecs slower for our configuration) and the other calls have nearly the same times.

- We have two different hardware configurations for the server and client sides. So, the comparisons between local and remote calls could be unhealthy. But it gives us an idea about the remote calls.

- We took the results mainly to measure the effects of remote calls with respect to the local ones. So, we will generally give the conclusions regarding to local/remote changes. Other aspects will be mentioned only if they deviate from local ones greatly. So, we have a lot of results here, but very few comments on them.

5.4.1 remote oneway and twoway invoke results

Figure 5.85 shows the results for oneway and twoway functions which take no arguments and return nothing.

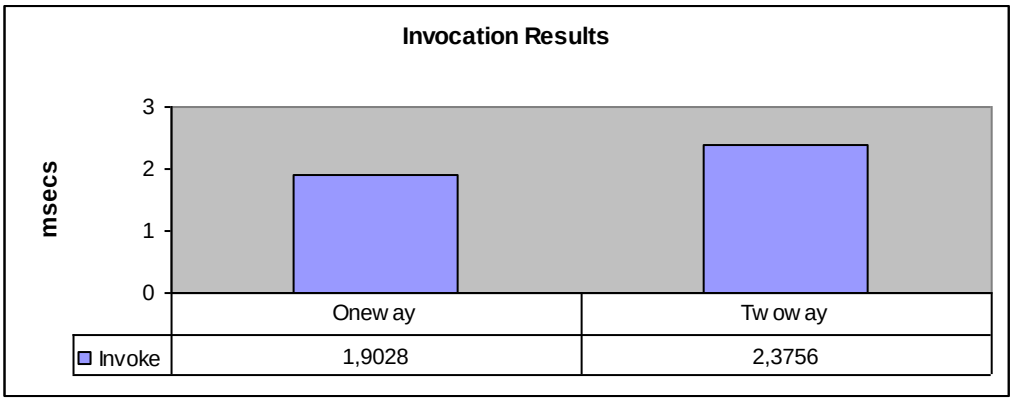


Figure 5.85 : Remote results for Oneway and Twoway Invocations

5.4.2 about oneway and twoway invoke results

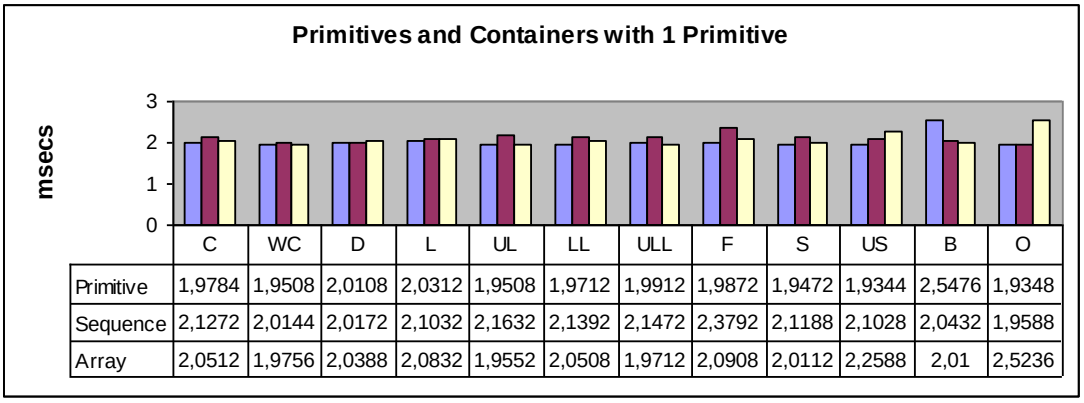
Remote results are faster than local ones and oneway is faster than twoway.

5.4.3 oneway – only send results

We will briefly refer to these results as R_O_OS (Remote_Oneway_OnlySend) results.

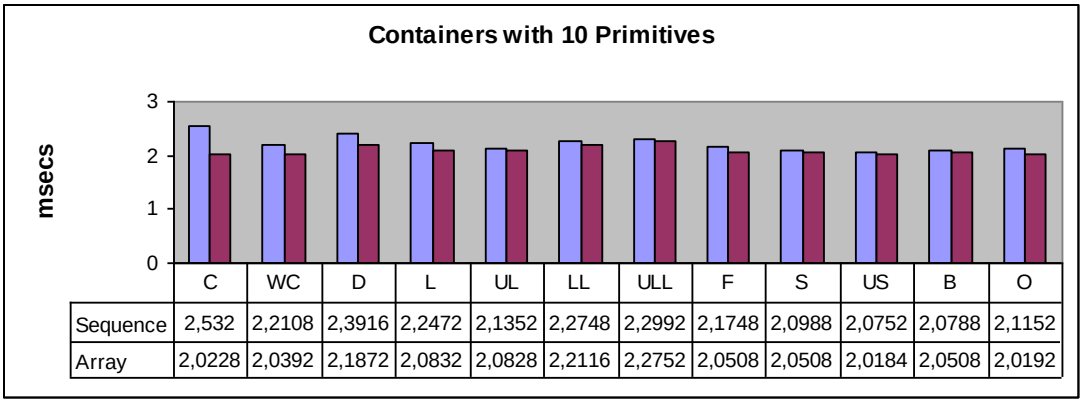
5.4.3.1 R_O_OS primitive and primitive container results

Figure 5.86 through 5.90 shows the results obtained for primitive types and containers with primitive types.



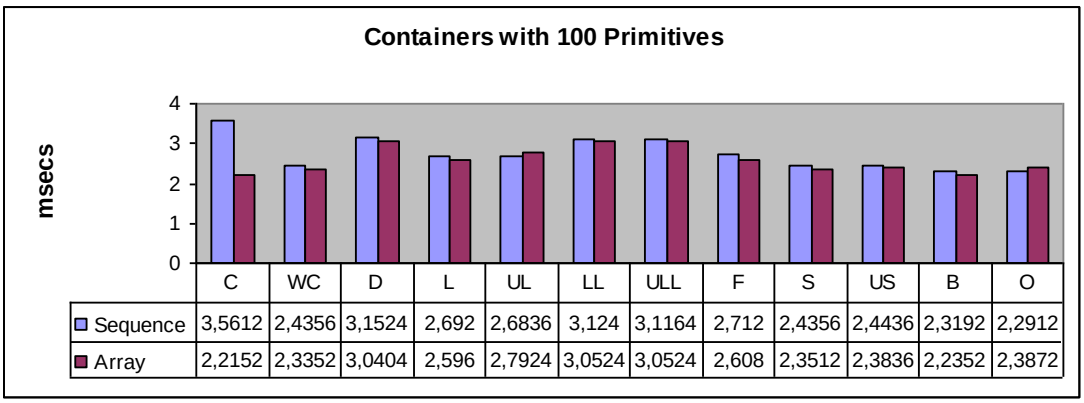
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Primitive	1,9784	1,9508	2,0108	2,0312	1,9508	1,9712	1,9912	1,9872	1,9472	1,9344	2,5476	1,9348
Sequence	2,1272	2,0144	2,0172	2,1032	2,1632	2,1392	2,1472	2,3792	2,1188	2,1028	2,0432	1,9588
Array	2,0512	1,9756	2,0388	2,0832	1,9552	2,0508	1,9712	2,0908	2,0112	2,2588	2,01	2,5236

Figure 5.86 : R_O_OS Results for Primitives and Containers with 1 Primitive



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,532	2,2108	2,3916	2,2472	2,1352	2,2748	2,2992	2,1748	2,0988	2,0752	2,0788	2,1152
Array	2,0228	2,0392	2,1872	2,0832	2,0828	2,2116	2,2752	2,0508	2,0508	2,0184	2,0508	2,0192

Figure 5.87 : R_O_OS Results for Containers with 10 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	3,5612	2,4356	3,1524	2,692	2,6836	3,124	3,1164	2,712	2,4356	2,4436	2,3192	2,2912
Array	2,2152	2,3352	3,0404	2,596	2,7924	3,0524	3,0524	2,608	2,3512	2,3836	2,2352	2,3872

Figure 5.88 : R_O_OS Results for Containers with 100 Primitives

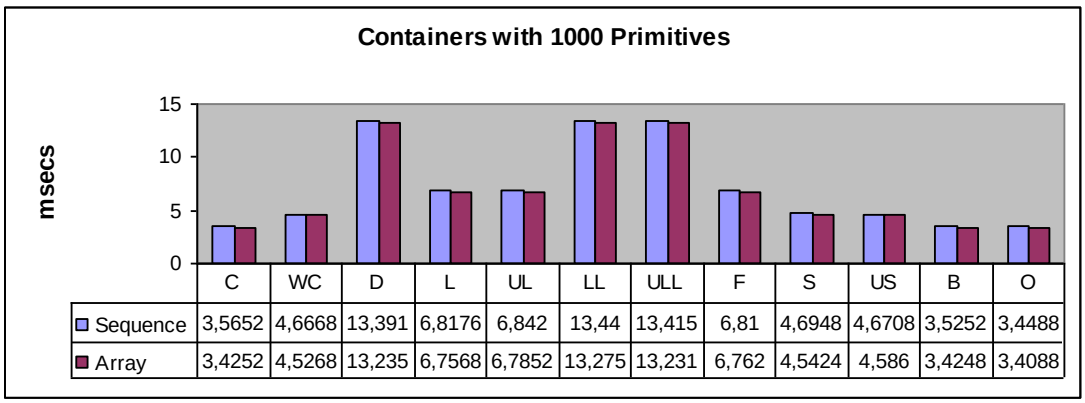


Figure 5.89 : R_O_OS Results for Containers with 1000 Primitives

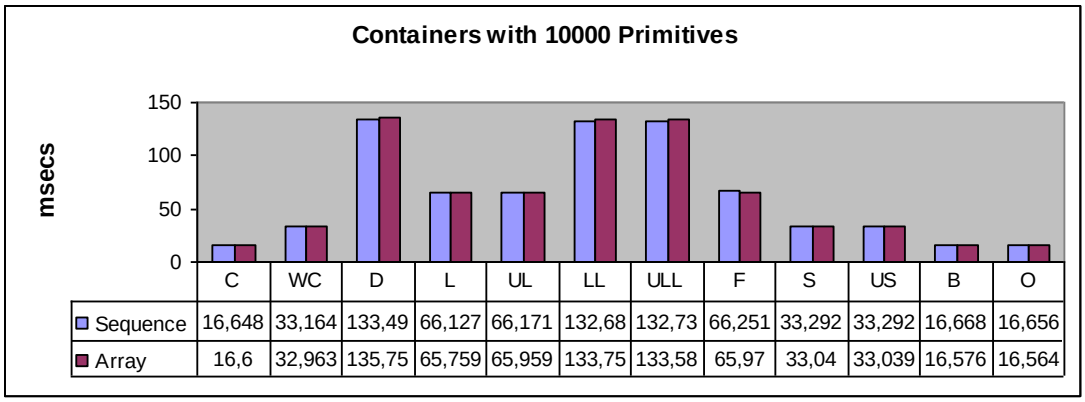


Figure 5.90 : R_O_OS Results for Containers with 10000 Primitives

5.4.3.2 about R_O_OS primitive and primitive container results

If we compare with L_O_OS results we see that local calls are faster than remote calls.

5.4.3.3 R_O_OS string and wide string results

Figure 5.91 shows the results obtained.

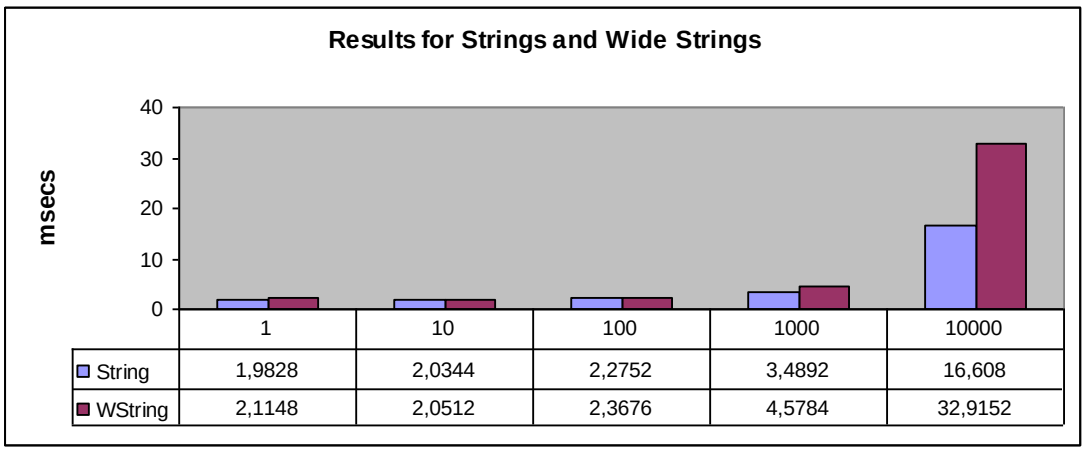


Figure 5.91 : R_O_OS Results for Strings and Wide Strings

5.4.3.4 about R_O_OS string and wide string results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- At the size 10000 the ratio of wstrings/strings nearly reach to the value of 2, which is the ratio of sizes of these two types.
- Comparison with L_O_OS results shows that the local calls are very fast with respect to remote ones.

5.4.3.5 R_O_OS struct and struct container results

Figures 5.92 through 5.96 shows the results obtained.

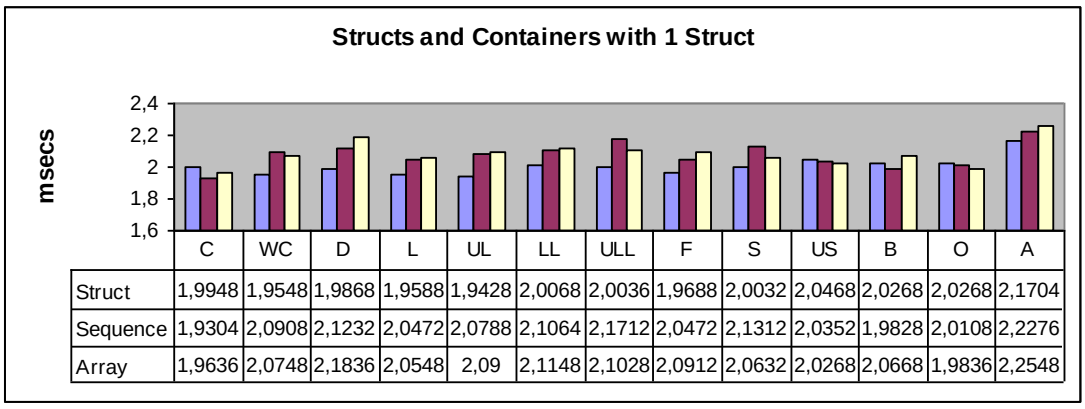
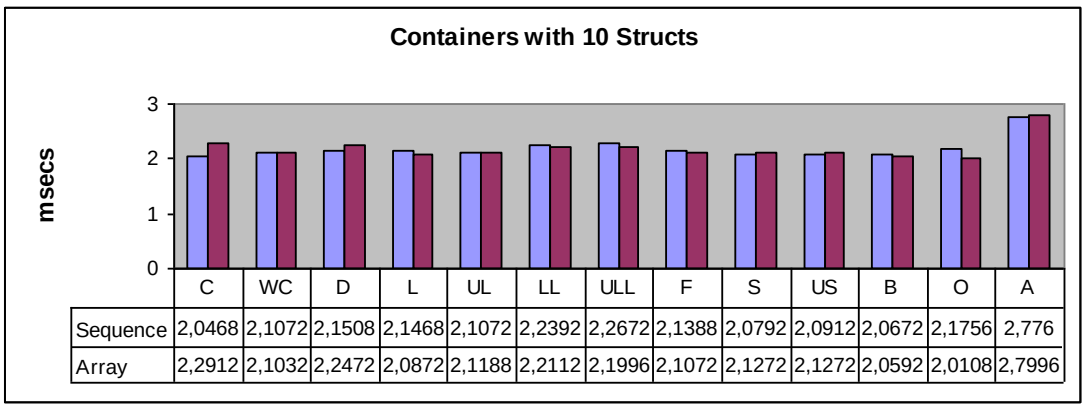
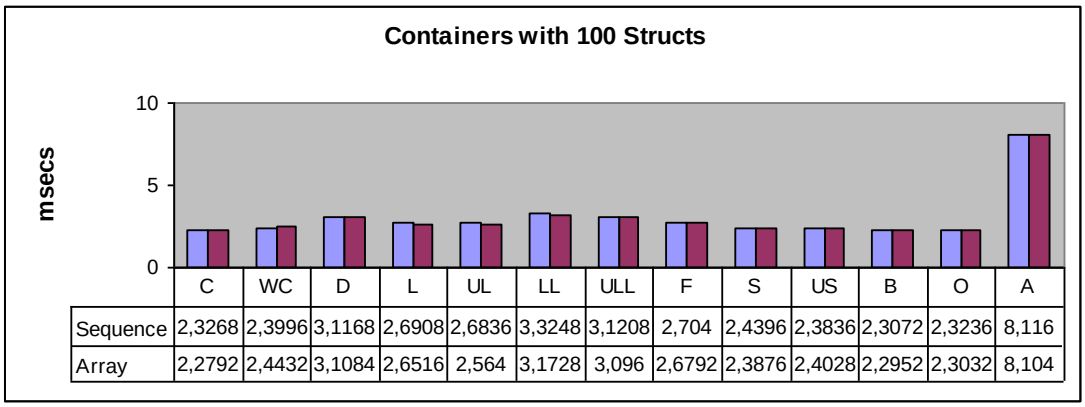


Figure 5.92 : R_O_OS Results for Structs and Containers with 1 Struct



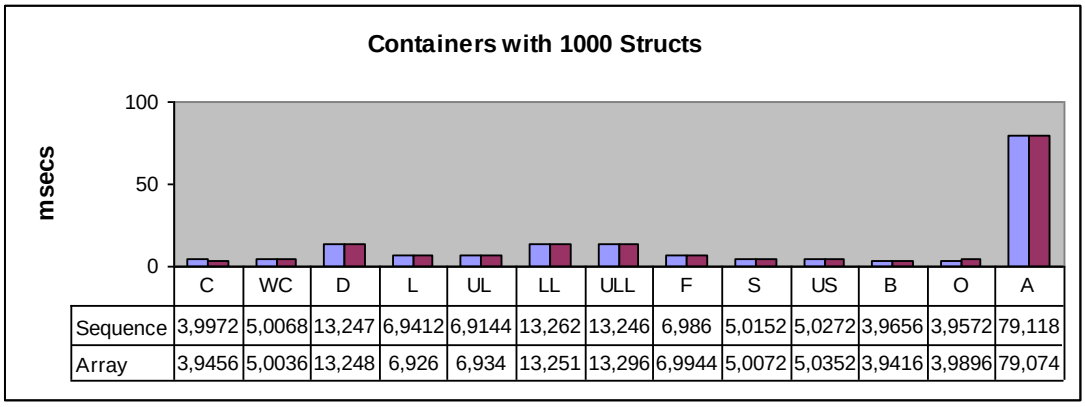
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,0468	2,1072	2,1508	2,1468	2,1072	2,2392	2,2672	2,1388	2,0792	2,0912	2,0672	2,1756	2,776
Array	2,2912	2,1032	2,2472	2,0872	2,1188	2,2112	2,1996	2,1072	2,1272	2,1272	2,0592	2,0108	2,7996

Figure 5.93 : R_O_OS Results for Containers with 10 Structs.



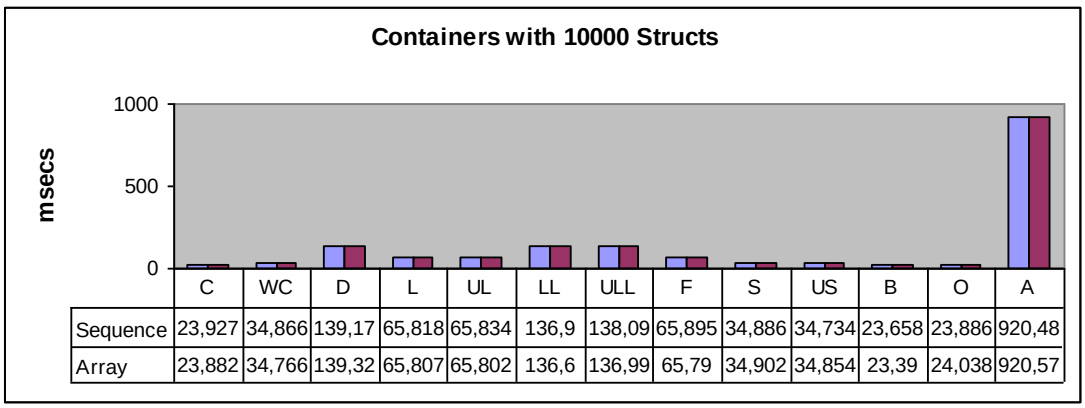
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,3268	2,3996	3,1168	2,6908	2,6836	3,3248	3,1208	2,704	2,4396	2,3836	2,3072	2,3236	8,116
Array	2,2792	2,4432	3,1084	2,6516	2,564	3,1728	3,096	2,6792	2,3876	2,4028	2,2952	2,3032	8,104

Figure 5.94 : R_O_OS Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,9972	5,0068	13,247	6,9412	6,9144	13,262	13,246	6,986	5,0152	5,0272	3,9656	3,9572	79,118
Array	3,9456	5,0036	13,248	6,926	6,934	13,251	13,296	6,9944	5,0072	5,0352	3,9416	3,9896	79,074

Figure 5.95 : R_O_OS Results for Containers with 1000 Structs.



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	23,927	34,866	139,17	65,818	65,834	136,9	138,09	65,895	34,886	34,734	23,658	23,886	920,48
Array	23,882	34,766	139,32	65,807	65,802	136,6	136,99	65,79	34,902	34,854	23,39	24,038	920,57

Figure 5.96 : R_O_OS Results for Containers with 10000 Structs

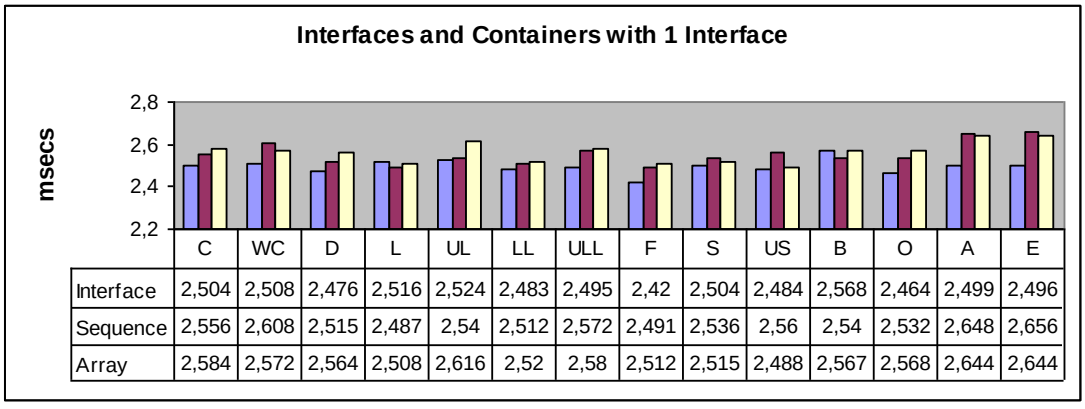
5.4.3.6 about R_O_OS struct and struct container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_O_OS results we see that local results are generally faster than remote ones especially for large-sized structs, For example, it takes about 280 msecs for local calls to send ten thousands of our special struct and and about 920 msecs for remote calls; locals are 3,5 times faster.

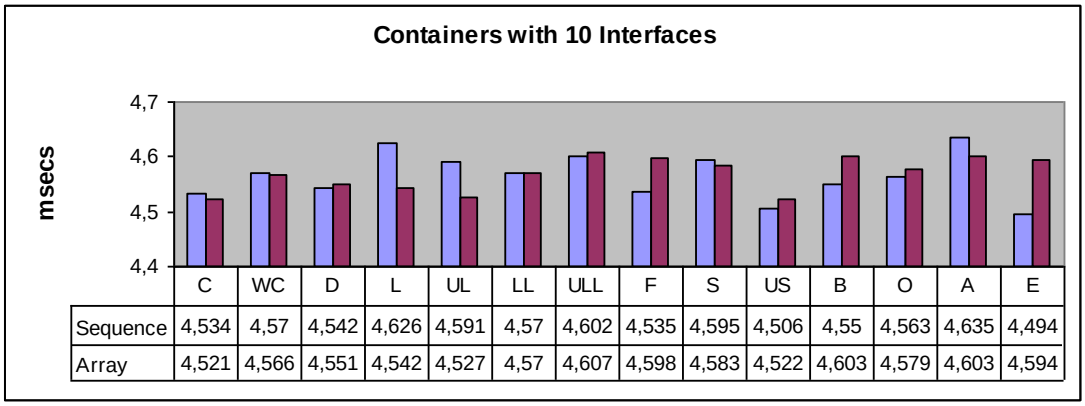
5.4.3.7 R_O_OS interface and interface container results

Figures 5.97 through 5.101 shows the results obtained.



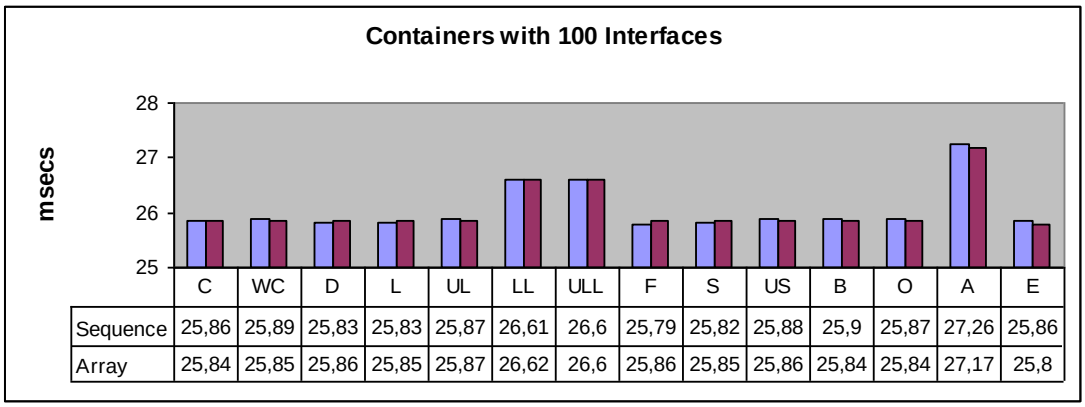
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	2,504	2,508	2,476	2,516	2,524	2,483	2,495	2,42	2,504	2,484	2,568	2,464	2,499	2,496
Sequence	2,556	2,608	2,515	2,487	2,54	2,512	2,572	2,491	2,536	2,56	2,54	2,532	2,648	2,656
Array	2,584	2,572	2,564	2,508	2,616	2,52	2,58	2,512	2,515	2,488	2,567	2,568	2,644	2,644

Figure 5.97 : R_O_OS Results for Interface and Containers with 1 Interface



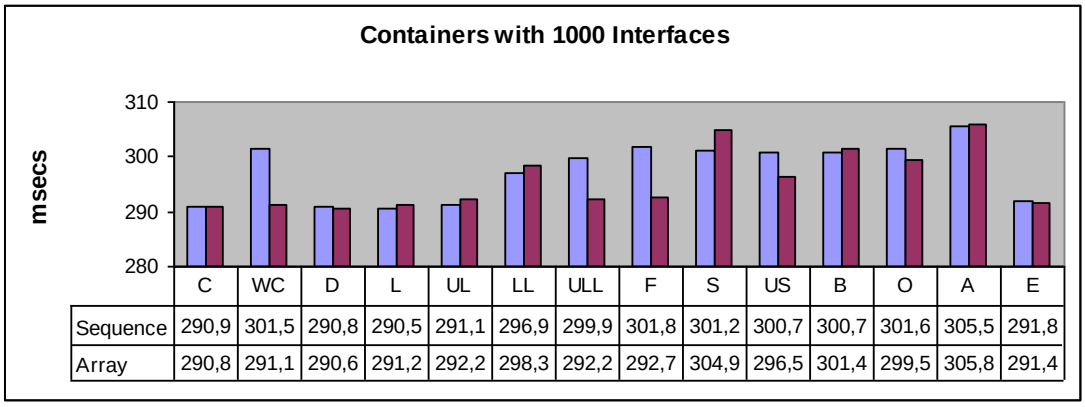
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	4,534	4,57	4,542	4,626	4,591	4,57	4,602	4,535	4,595	4,506	4,55	4,563	4,635	4,494
Array	4,521	4,566	4,551	4,542	4,527	4,57	4,607	4,598	4,583	4,522	4,603	4,579	4,603	4,594

Figure 5.98 : R_O_OS Results for Containers with 10 Interfaces



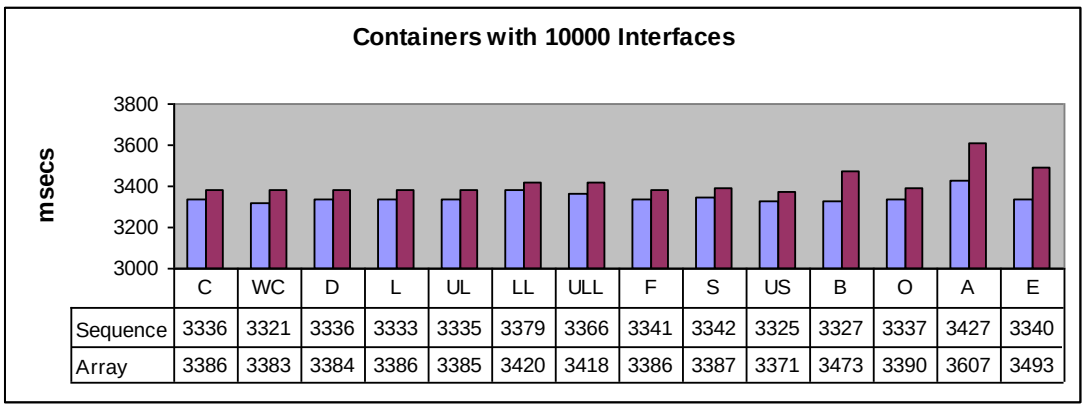
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	25,86	25,89	25,83	25,83	25,87	26,61	26,6	25,79	25,82	25,88	25,9	25,87	27,26	25,86
Array	25,84	25,85	25,86	25,85	25,87	26,62	26,6	25,86	25,85	25,86	25,84	25,84	27,17	25,8

Figure 5.99 : R_O_OS Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	290,9	301,5	290,8	290,5	291,1	296,9	299,9	301,8	301,2	300,7	300,7	301,6	305,5	291,8
Array	290,8	291,1	290,6	291,2	292,2	298,3	292,2	292,7	304,9	296,5	301,4	299,5	305,8	291,4

Figure 5.100 : R_O_OS Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	3336	3321	3336	3333	3335	3379	3366	3341	3342	3325	3327	3337	3427	3340
Array	3386	3383	3384	3386	3385	3420	3418	3386	3387	3371	3473	3390	3607	3493

Figure 5.101 : R_O_OS Results for Containers with 10000 Interfaces

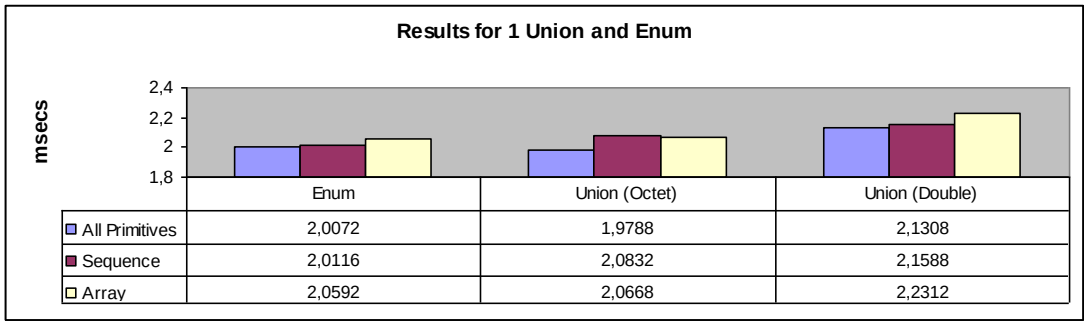
5.4.3.8 about R_O_OS interface and interface container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_O_OS results we see that local calls are faster than remote calls for sizes greater than 1.
- We got the out of memory error at L_O_OS for size 10000. Here we could completed the test for this size.

5.4.3.9 R_O_OS union and enum results

Figures 5.102 through 5.106 show the results obtained.



	Enum	Union (Octet)	Union (Double)
All Primitives	2,0072	1,9788	2,1308
Sequence	2,0116	2,0832	2,1588
Array	2,0592	2,0668	2,2312

Figure 5.102 : R_O_OS Results for Union, Enum and Containers with 1 Union and Enum

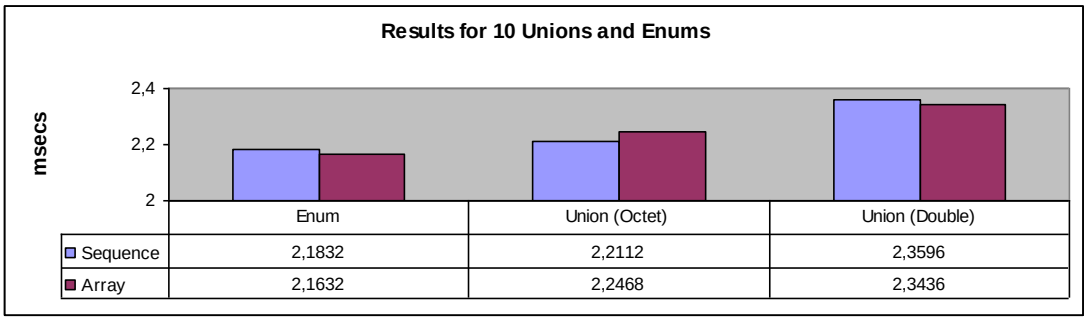


Figure 5.103 : R_O_OS Results for Containers with 10 Unions and Enums

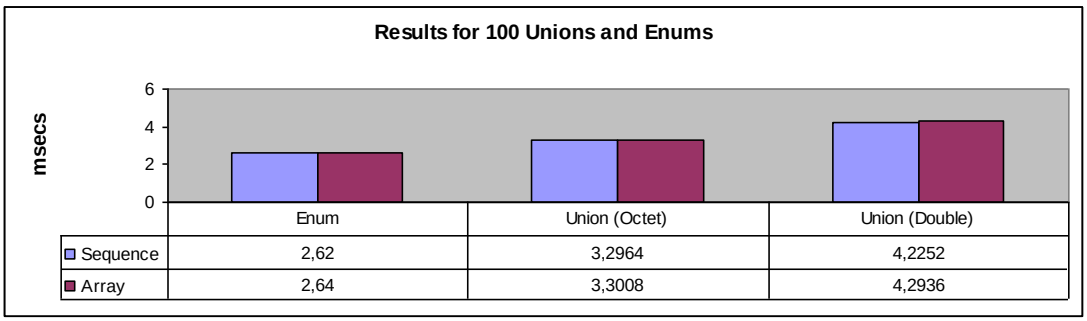


Figure 5.104 : R_O_OS Results for Containers with 100 Unions and Enums

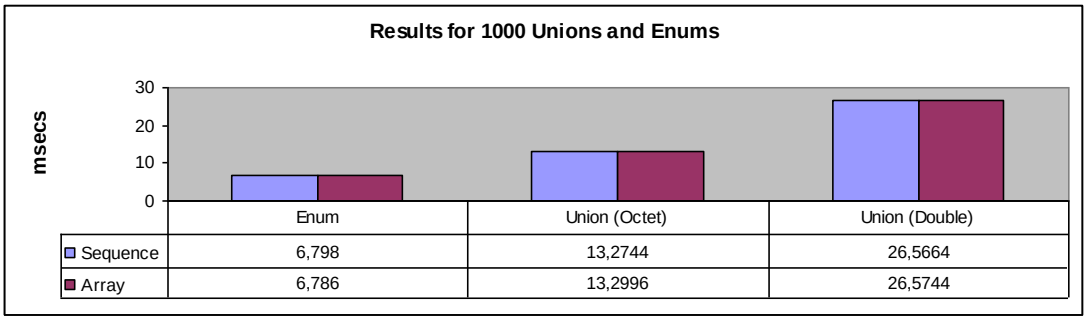


Figure 5.105 : R_O_OS Results for Containers with 1000 Unions and Enums

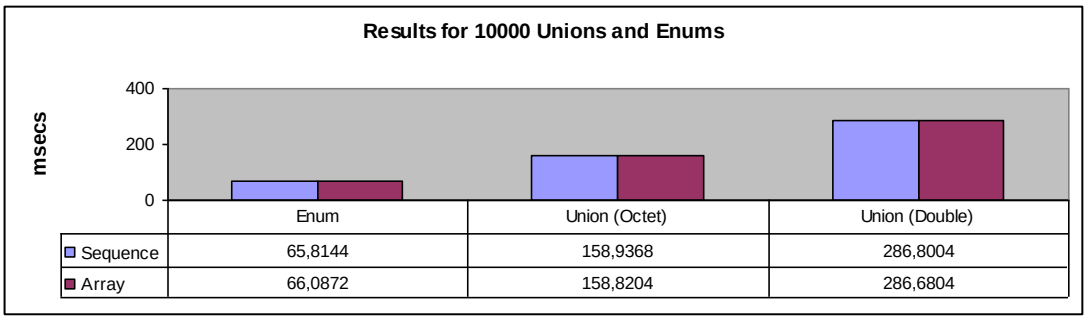


Figure 5.106 : R_O_OS Results for Containers with 10000 Unions and Enums

5.4.3.10 about R_O_OS union and enum results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

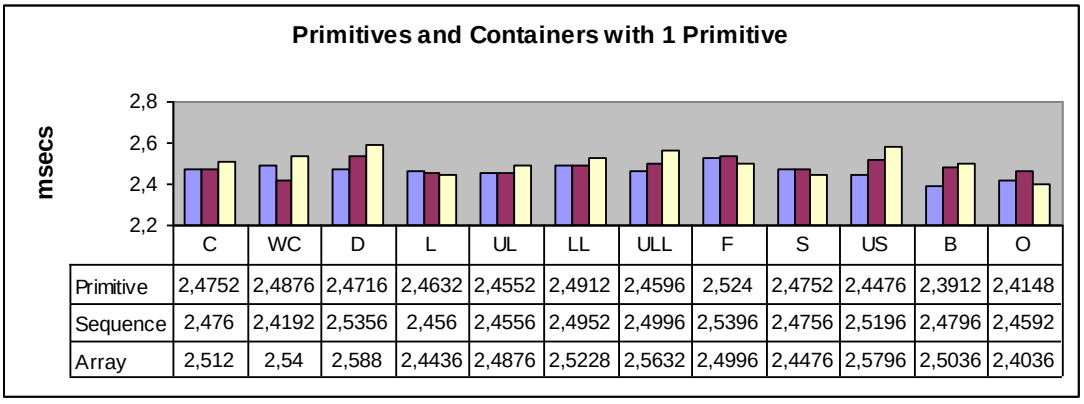
- If we compare with L_O_OS results we see that local calls are faster than remote calls for sizes greater than about 100.

5.4.4 twoway – only send results

We will briefly refer to these results as R_T_OS (Remote_Twoway_OnlySend) results.

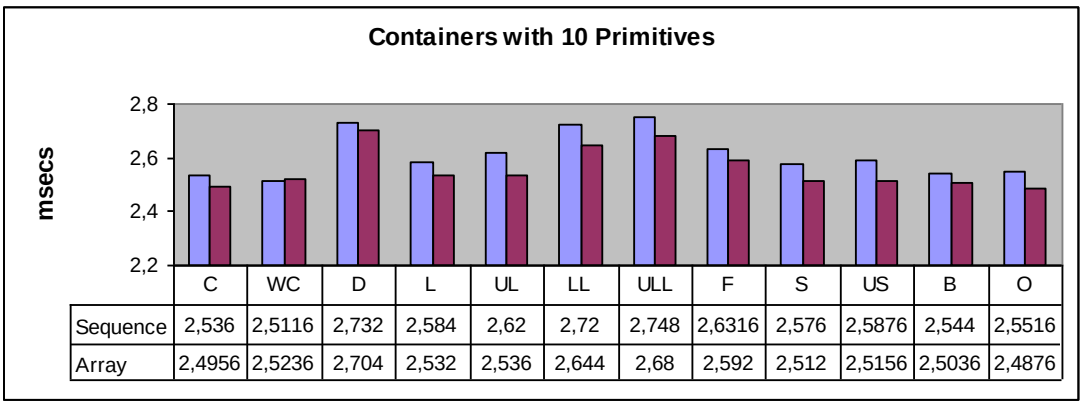
5.4.4.1 R_T_OS primitive and primitive container results

Figure 5.107 through 5.111 shows the results obtained for primitive types and containers with primitive types.



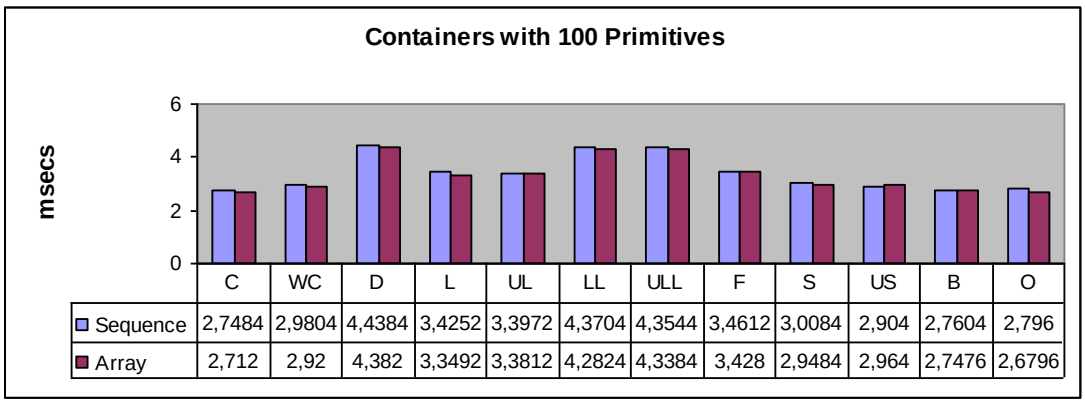
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Primitive	2,4752	2,4876	2,4716	2,4632	2,4552	2,4912	2,4596	2,524	2,4752	2,4476	2,3912	2,4148
Sequence	2,476	2,4192	2,5356	2,456	2,4556	2,4952	2,4996	2,5396	2,4756	2,5196	2,4796	2,4592
Array	2,512	2,54	2,588	2,4436	2,4876	2,5228	2,5632	2,4996	2,4476	2,5796	2,5036	2,4036

Figure 5.107 : R_T_OS Results for Primitives and Containers with 1 Primitive



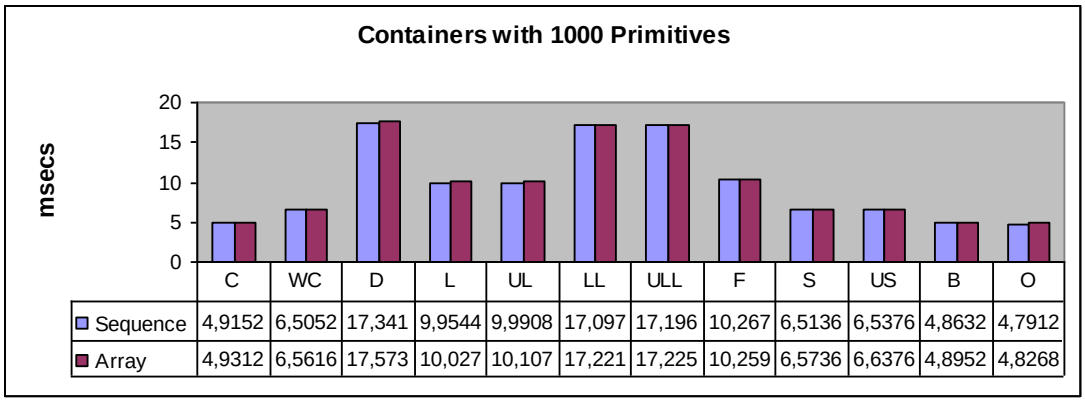
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,536	2,5116	2,732	2,584	2,62	2,72	2,748	2,6316	2,576	2,5876	2,544	2,5516
Array	2,4956	2,5236	2,704	2,532	2,536	2,644	2,68	2,592	2,512	2,5156	2,5036	2,4876

Figure 5.108 : R_T_OS Results for Containers with 10 Primitives



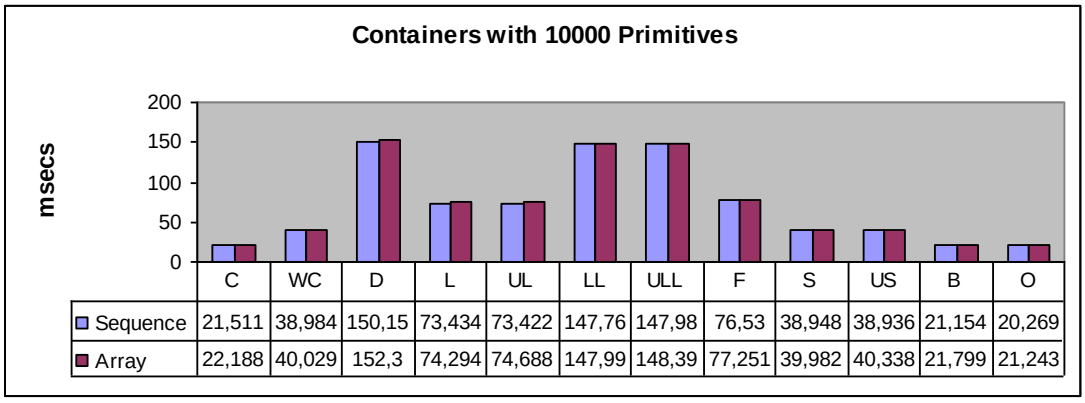
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,7484	2,9804	4,4384	3,4252	3,3972	4,3704	4,3544	3,4612	3,0084	2,904	2,7604	2,796
Array	2,712	2,92	4,382	3,3492	3,3812	4,2824	4,3384	3,428	2,9484	2,964	2,7476	2,6796

Figure 5.109 : R_T_OS Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	4,9152	6,5052	17,341	9,9544	9,9908	17,097	17,196	10,267	6,5136	6,5376	4,8632	4,7912
Array	4,9312	6,5616	17,573	10,027	10,107	17,221	17,225	10,259	6,5736	6,6376	4,8952	4,8268

Figure 5.110 : R_T_OS Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	21,511	38,984	150,15	73,434	73,422	147,76	147,98	76,53	38,948	38,936	21,154	20,269
Array	22,188	40,029	152,3	74,294	74,688	147,99	148,39	77,251	39,982	40,338	21,799	21,243

Figure 5.111 : R_T_OS Results for Containers with 10000 Primitives

5.4.4.2 about R_T_OS primitive and primitive container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

- If we compare with L_T_OS results we see that local calls are faster than remote calls.

5.4.4.3 R_T_OS string and wide string results

Figure 5.112 shows the results obtained.

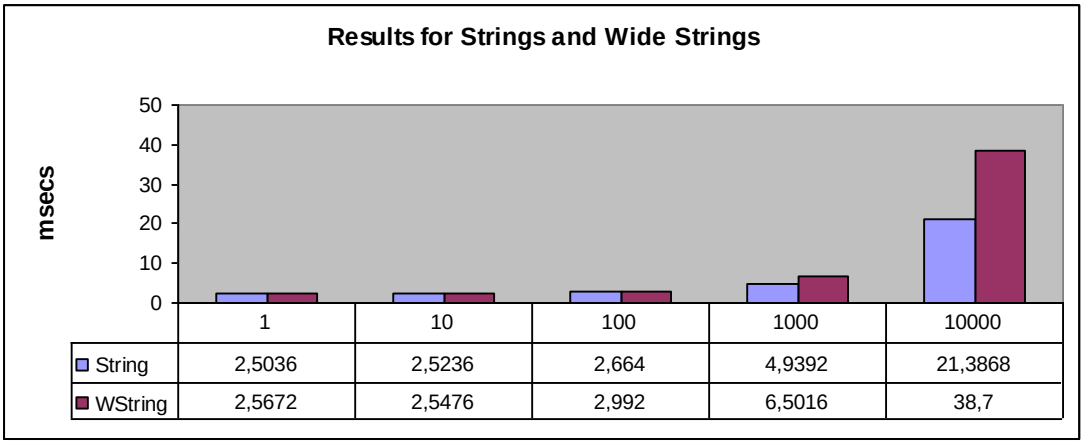


Figure 5.112 : R_T_OS Results for Strings and Wide Strings

5.4.4.4 about R_T_OS string and wide string results

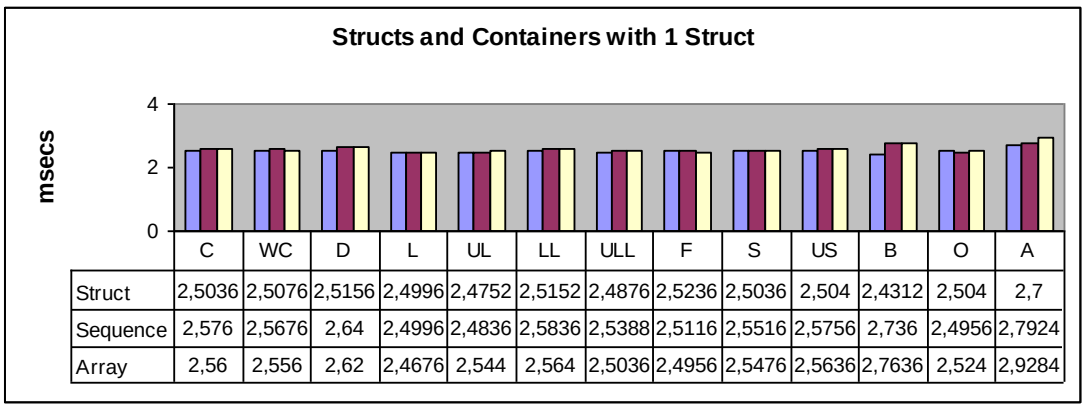
Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

- If we compare with L_T_OS results we see that local calls are faster than remote calls.

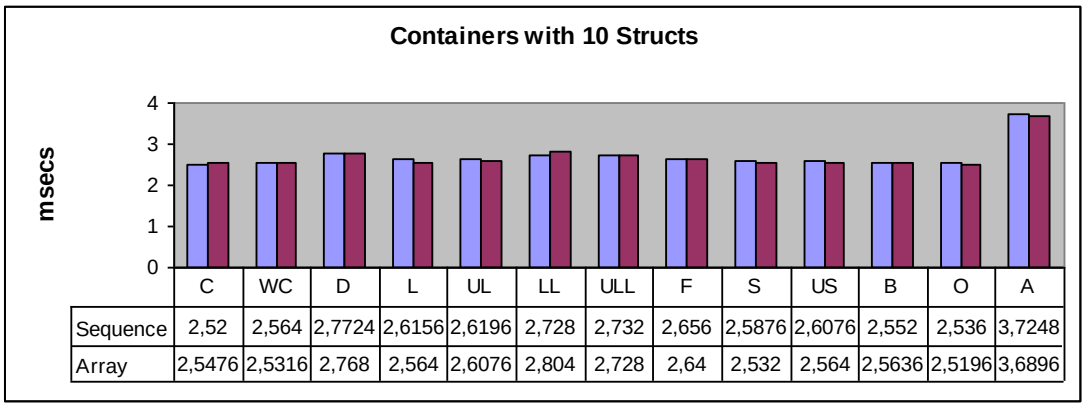
5.4.4.5 R_T_OS struct and struct container results

Figures 5.113 through 5.117 shows the results obtained.



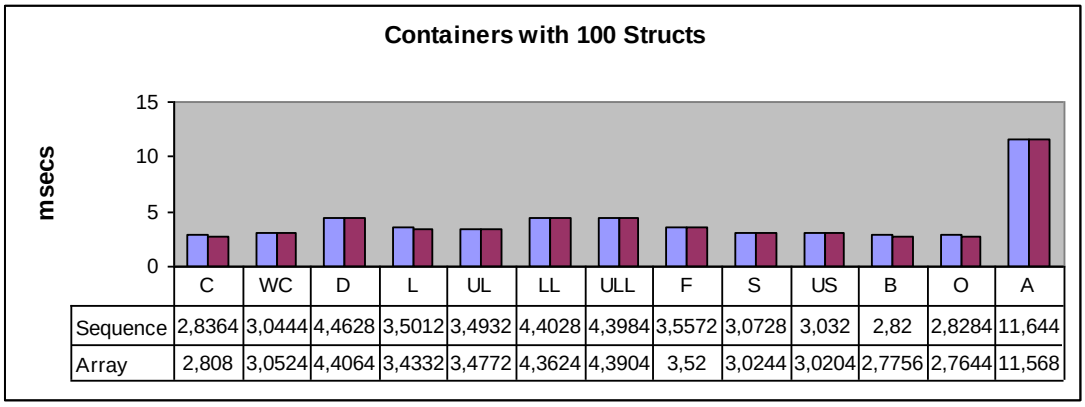
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Struct	2,5036	2,5076	2,5156	2,4996	2,4752	2,5152	2,4876	2,5236	2,5036	2,504	2,4312	2,504	2,7
Sequence	2,576	2,5676	2,64	2,4996	2,4836	2,5836	2,5388	2,5116	2,5516	2,5756	2,736	2,4956	2,7924
Array	2,56	2,556	2,62	2,4676	2,544	2,564	2,5036	2,4956	2,5476	2,5636	2,7636	2,524	2,9284

Figure 5.113 : R_T_OS Results for Structs and Containers with 1 Struct



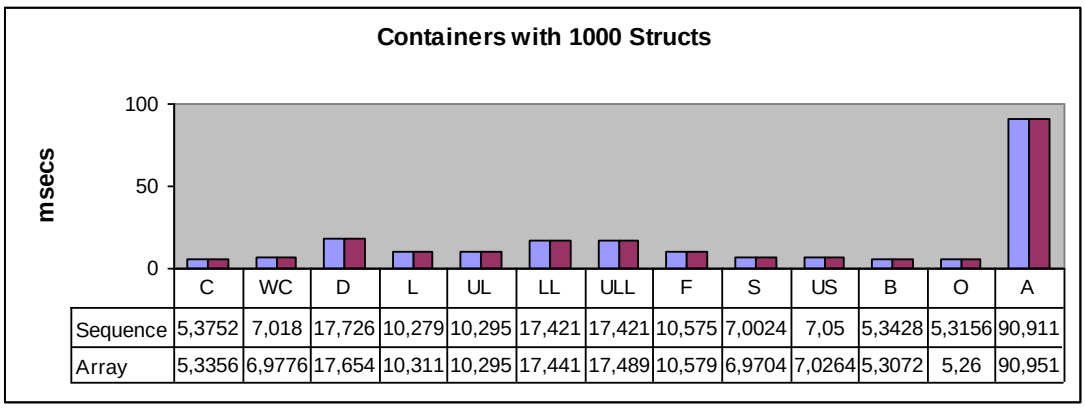
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,52	2,564	2,7724	2,6156	2,6196	2,728	2,732	2,656	2,5876	2,6076	2,552	2,536	3,7248
Array	2,5476	2,5316	2,768	2,564	2,6076	2,804	2,728	2,64	2,532	2,564	2,5636	2,5196	3,6896

Figure 5.114 : R_T_OS Results for Containers with 10 Structs.



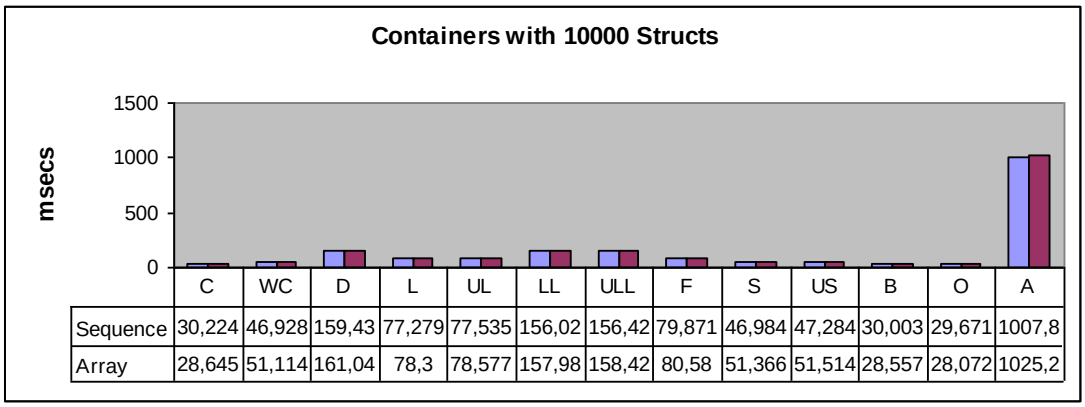
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,8364	3,0444	4,4628	3,5012	3,4932	4,4028	4,3984	3,5572	3,0728	3,032	2,82	2,8284	11,644
Array	2,808	3,0524	4,4064	3,4332	3,4772	4,3624	4,3904	3,52	3,0244	3,0204	2,7756	2,7644	11,568

Figure 5.115 : R_T_OS Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	5,3752	7,018	17,726	10,279	10,295	17,421	17,421	10,575	7,0024	7,05	5,3428	5,3156	90,911
Array	5,3356	6,9776	17,654	10,311	10,295	17,441	17,489	10,579	6,9704	7,0264	5,3072	5,26	90,951

Figure 5.116 : R_T_OS Results for Containers with 1000 Structs.



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	30,224	46,928	159,43	77,279	77,535	156,02	156,42	79,871	46,984	47,284	30,003	29,671	1007,8
Array	28,645	51,114	161,04	78,3	78,577	157,98	158,42	80,58	51,366	51,514	28,557	28,072	1025,2

Figure 5.117 : R_T_OS Results for Containers with 10000 Structs

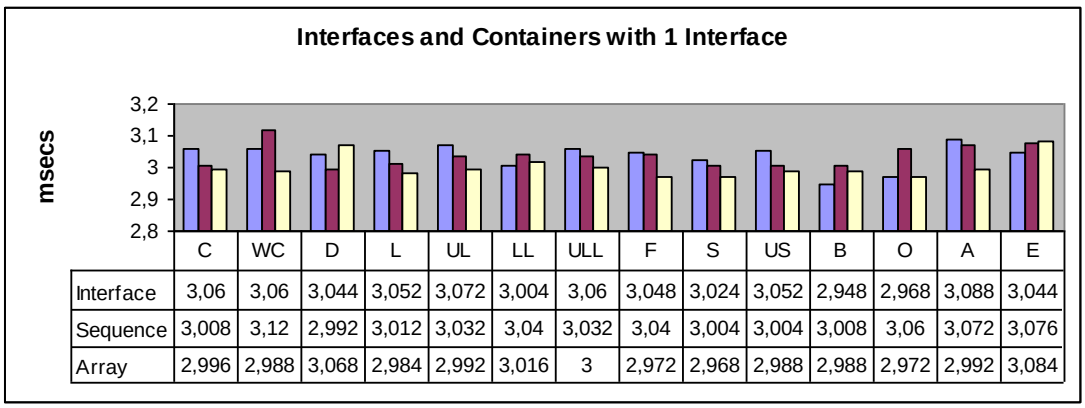
5.4.4.6 about R_T_OS struct and struct container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_OS results we see that local calls are faster than remote calls.

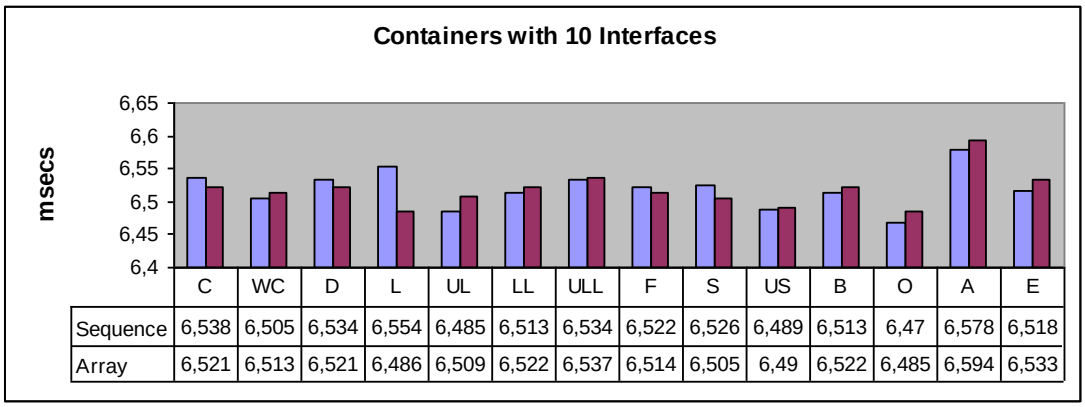
5.4.4.7 R_T_OS interface and interface container results

Figures 5.118 through 5.122 shows the results obtained.



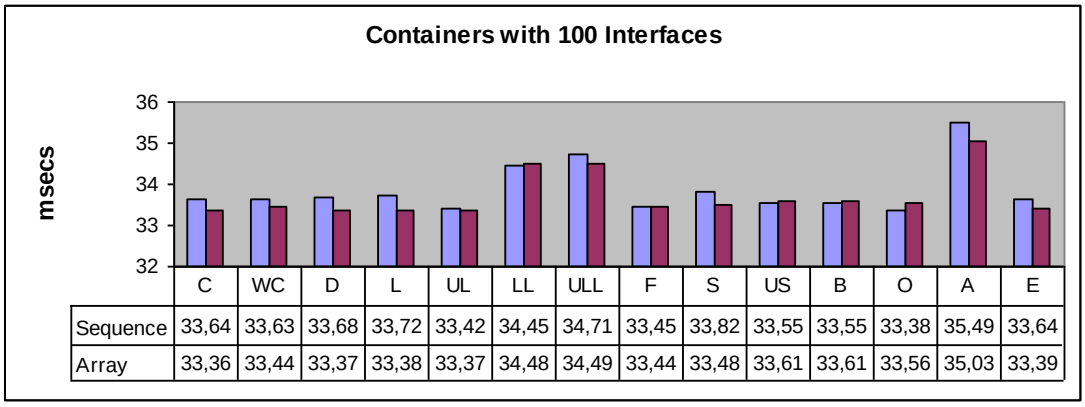
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	3,06	3,06	3,044	3,052	3,072	3,004	3,06	3,048	3,024	3,052	2,948	2,968	3,088	3,044
Sequence	3,008	3,12	2,992	3,012	3,032	3,04	3,032	3,04	3,004	3,004	3,008	3,06	3,072	3,076
Array	2,996	2,988	3,068	2,984	2,992	3,016	3	2,972	2,968	2,988	2,988	2,972	2,992	3,084

Figure 5.118 : R_T_OS Results for Interface and Containers with 1 Interface



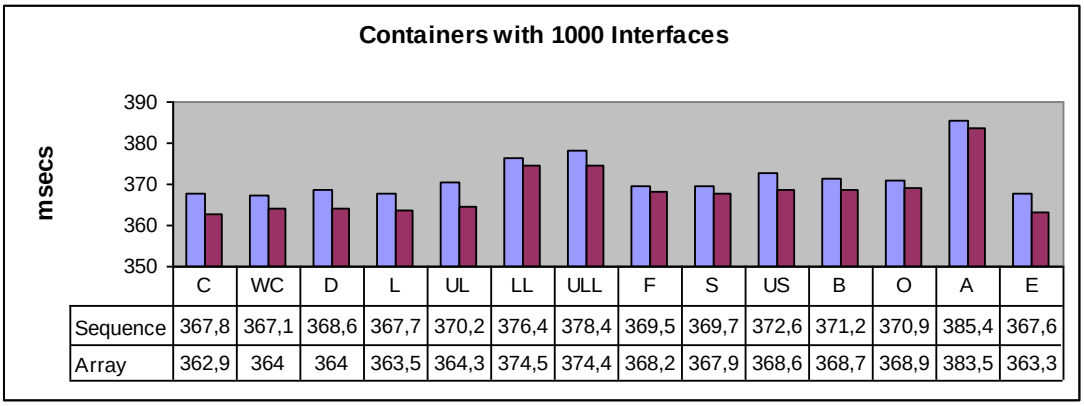
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	6,538	6,505	6,534	6,554	6,485	6,513	6,534	6,522	6,526	6,489	6,513	6,47	6,578	6,518
Array	6,521	6,513	6,521	6,486	6,509	6,522	6,537	6,514	6,505	6,49	6,522	6,485	6,594	6,533

Figure 5.119 : R_T_OS Results for Containers with 10 Interfaces



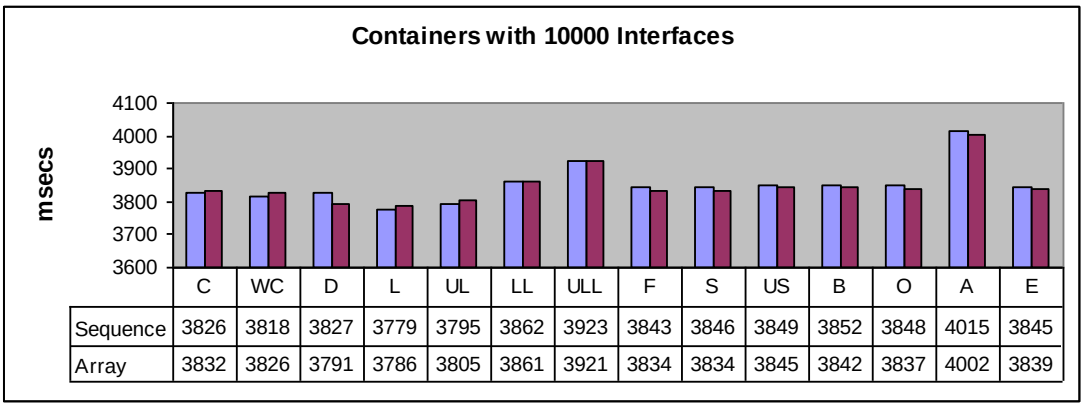
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	33,64	33,63	33,68	33,72	33,42	34,45	34,71	33,45	33,82	33,55	33,55	33,38	35,49	33,64
Array	33,36	33,44	33,37	33,38	33,37	34,48	34,49	33,44	33,48	33,61	33,61	33,56	35,03	33,39

Figure 5.120 : R_T_OS Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	367,8	367,1	368,6	367,7	370,2	376,4	378,4	369,5	369,7	372,6	371,2	370,9	385,4	367,6
Array	362,9	364	364	363,5	364,3	374,5	374,4	368,2	367,9	368,6	368,7	368,9	383,5	363,3

Figure 5.121 : R_T_OS Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	3826	3818	3827	3779	3795	3862	3923	3843	3846	3849	3852	3848	4015	3845
Array	3832	3826	3791	3786	3805	3861	3921	3834	3834	3845	3842	3837	4002	3839

Figure 5.122 : R_T_OS Results for Containers with 10000 Interfaces

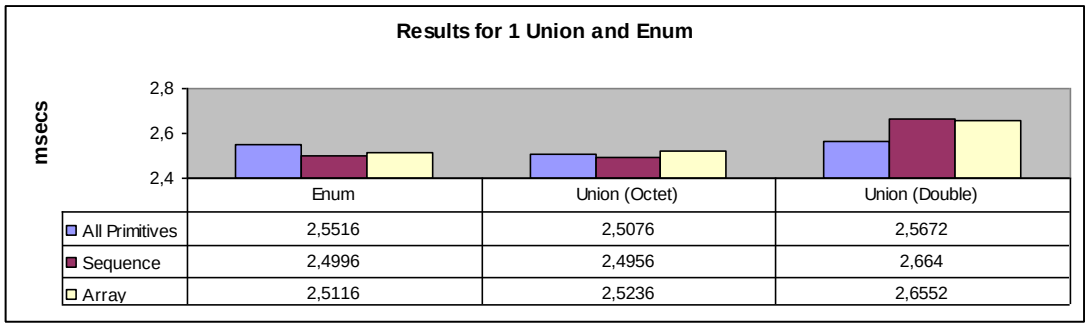
5.4.4.8 about R_T_OS interface and interface container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_OS results we see that local calls are faster than remote calls.

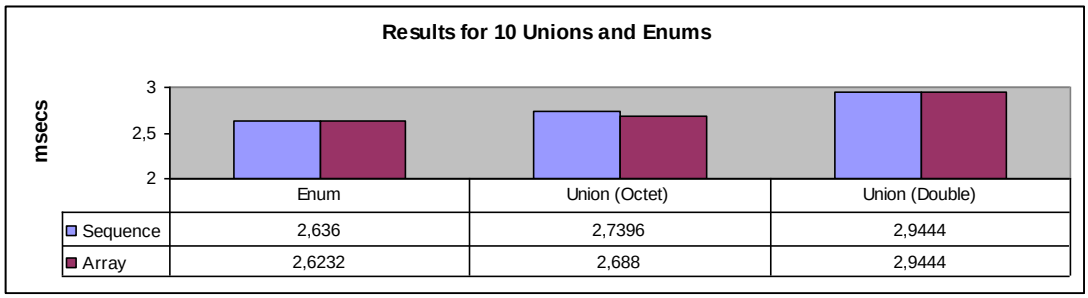
5.4.4.9 R_T_OS union and enum results

Figures 5.123 through 5.127 shows the results obtained.



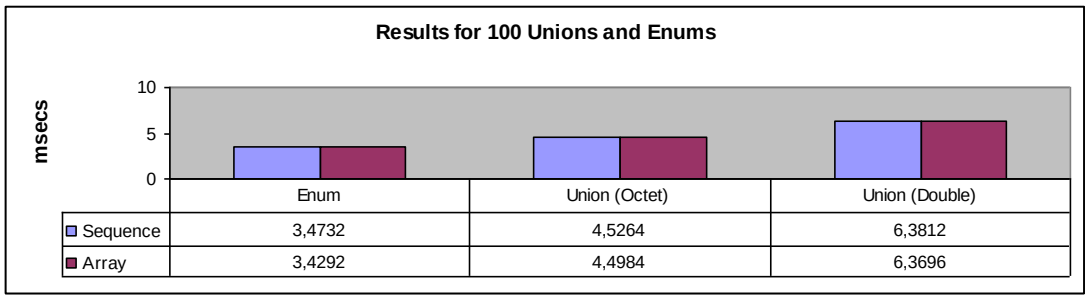
	Enum	Union (Octet)	Union (Double)
All Primitives	2,5516	2,5076	2,5672
Sequence	2,4996	2,4956	2,664
Array	2,5116	2,5236	2,6552

Figure 5.123 : R_T_OS Results for Union, Enum and Containers with 1 Union and Enum



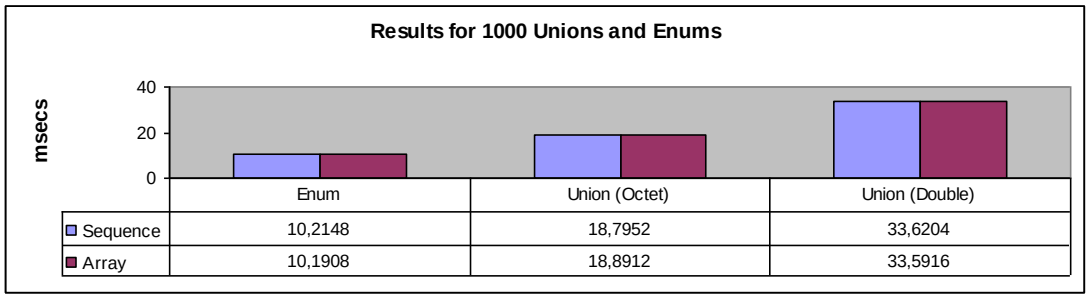
	Enum	Union (Octet)	Union (Double)
Sequence	2,636	2,7396	2,9444
Array	2,6232	2,688	2,9444

Figure 5.124 : R_T_OS Results for Containers with 10 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	3,4732	4,5264	6,3812
Array	3,4292	4,4984	6,3696

Figure 5.125 : R_T_OS Results for Containers with 100 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	10,2148	18,7952	33,6204
Array	10,1908	18,8912	33,5916

Figure 5.126 : R_T_OS Results for Containers with 1000 Unions and Enums

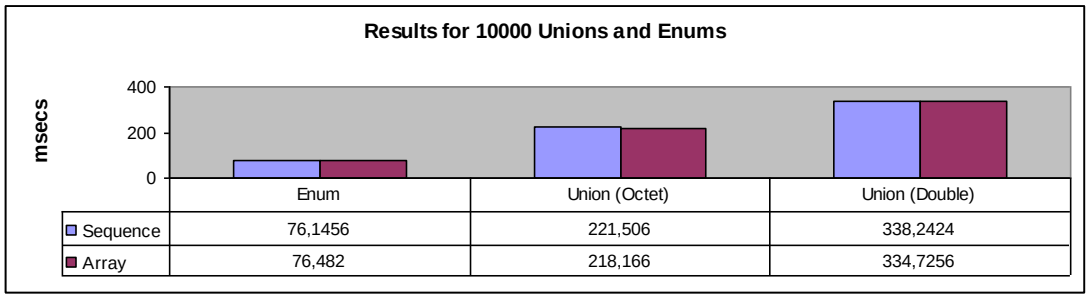


Figure 5.127 : R_T_OS Results for Containers with 10000 Unions and Enums

5.4.4.10 about R_T_OS union and enum results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_OS results we see that local calls are faster than remote calls.

5.4.5 twoway – send get results

We will briefly refer to these results as R_T_SG (Remote_Twoway_SendGet) results.

5.4.5.1 R_T_SG primitive and primitive container results

Figure 5.128 through 5.132 shows the results obtained for primitive types and containers with primitive types.

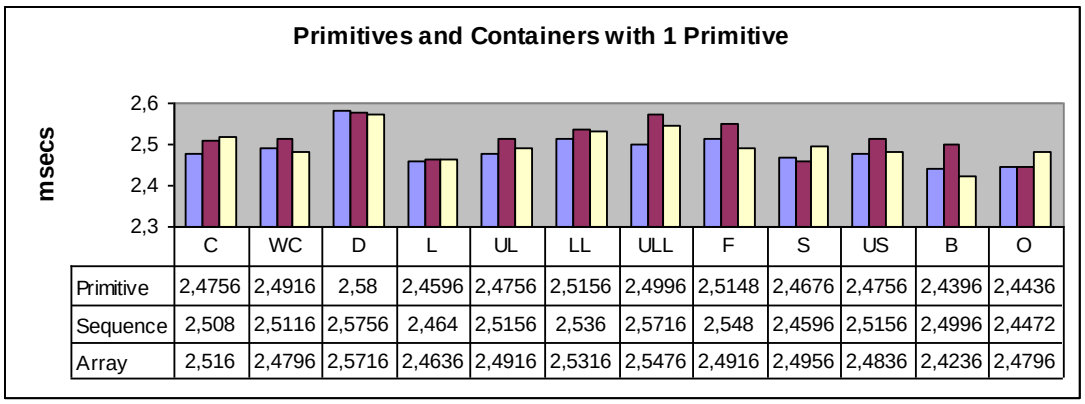
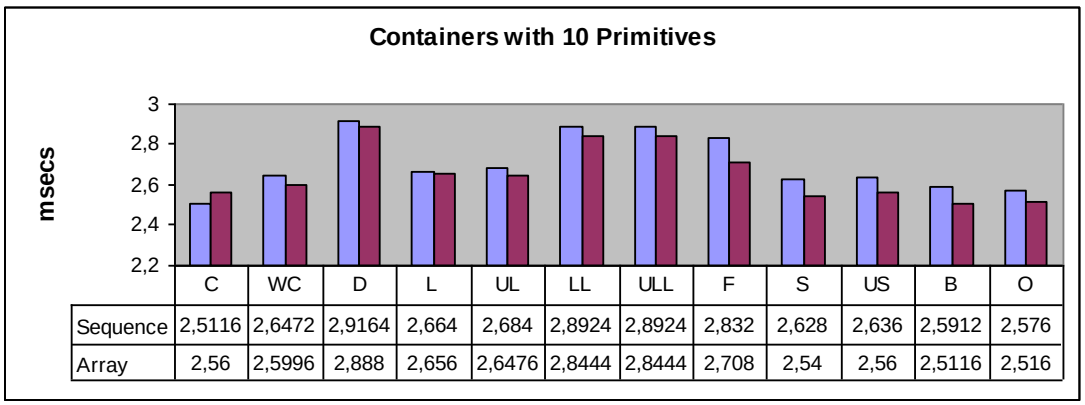
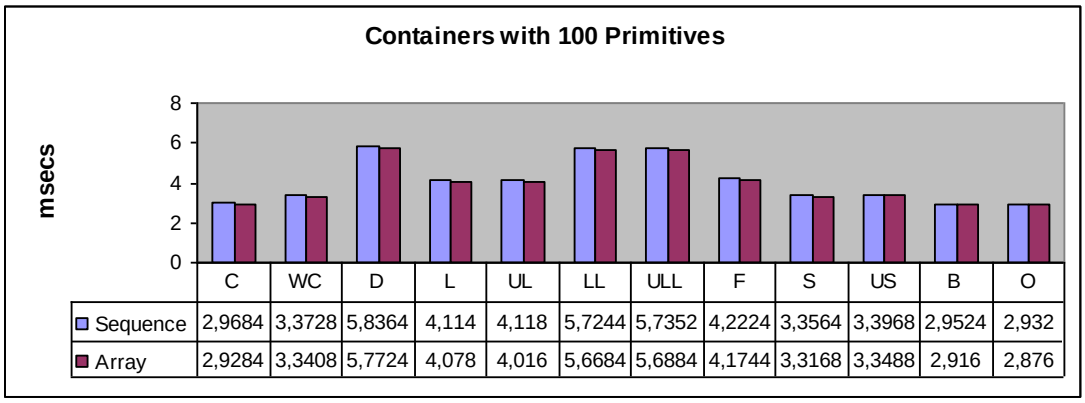


Figure 5.128 : R_T_SG Results for Primitives and Containers with 1 Primitive



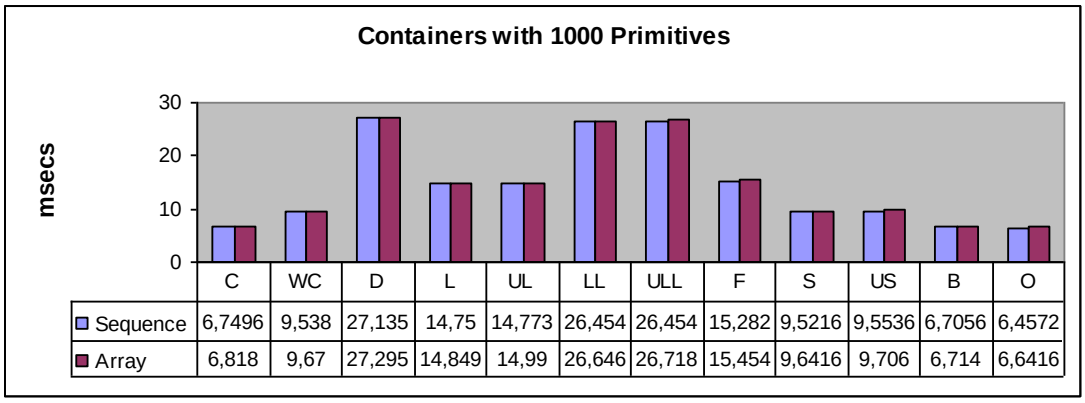
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,5116	2,6472	2,9164	2,664	2,684	2,8924	2,8924	2,832	2,628	2,636	2,5912	2,576
Array	2,56	2,5996	2,888	2,656	2,6476	2,8444	2,8444	2,708	2,54	2,56	2,5116	2,516

Figure 5.129 : R_T_SG Results for Containers with 10 Primitives



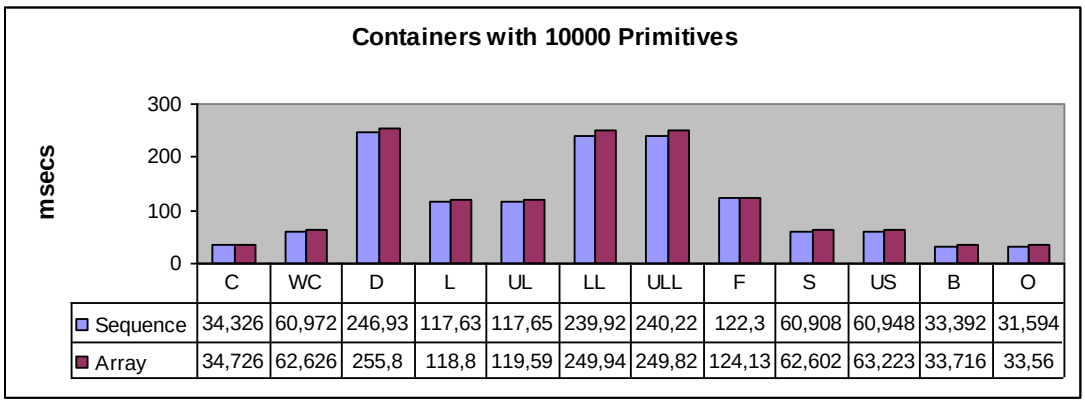
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,9684	3,3728	5,8364	4,114	4,118	5,7244	5,7352	4,2224	3,3564	3,3968	2,9524	2,932
Array	2,9284	3,3408	5,7724	4,078	4,016	5,6684	5,6884	4,1744	3,3168	3,3488	2,916	2,876

Figure 5.130 : R_T_SG Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	6,7496	9,538	27,135	14,75	14,773	26,454	26,454	15,282	9,5216	9,5536	6,7056	6,4572
Array	6,818	9,67	27,295	14,849	14,99	26,646	26,718	15,454	9,6416	9,706	6,714	6,6416

Figure 5.131 : R_T_SG Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	34,326	60,972	246,93	117,63	117,65	239,92	240,22	122,3	60,908	60,948	33,392	31,594
Array	34,726	62,626	255,8	118,8	119,59	249,94	249,82	124,13	62,602	63,223	33,716	33,56

Figure 5.132 : R_T_SG Results for Containers with 10000 Primitives

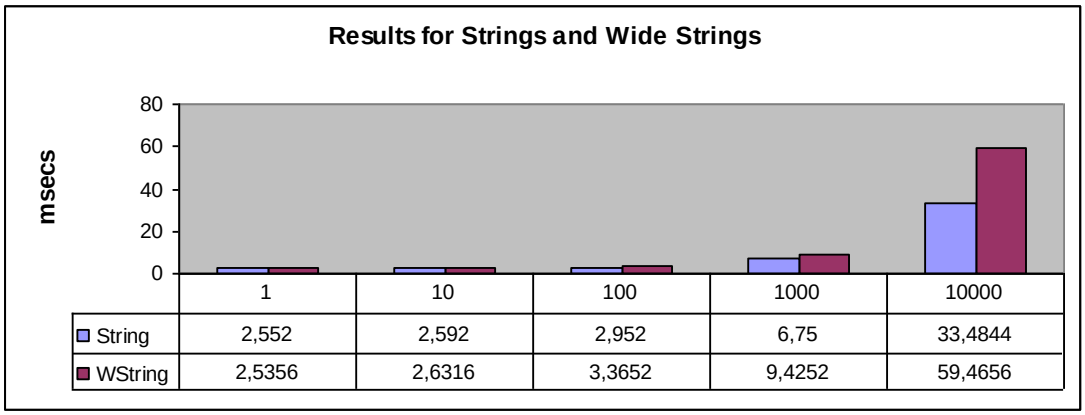
5.4.5.2 about R_T_SG primitive and primitive container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_SG results we see that local calls are faster than remote calls.

5.4.5.3 R_T_SG string and wide string results

Figure 5.133 shows the results obtained.



	1	10	100	1000	10000
String	2,552	2,592	2,952	6,75	33,4844
WString	2,5356	2,6316	3,3652	9,4252	59,4656

Figure 5.133 : R_T_SG Results for Strings and Wide Strings

5.4.5.4 about R_T_SG string and wide string results

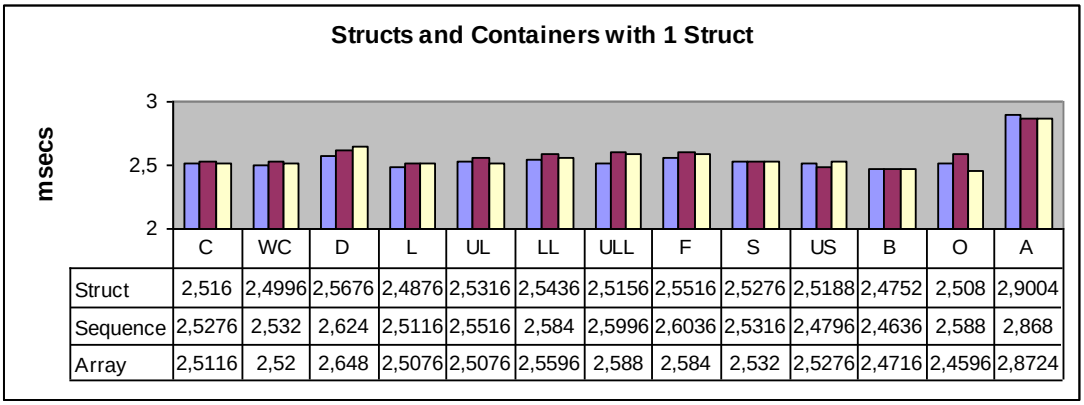
Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

- If we compare with L_T_SG results we see that local calls are faster than remote calls.

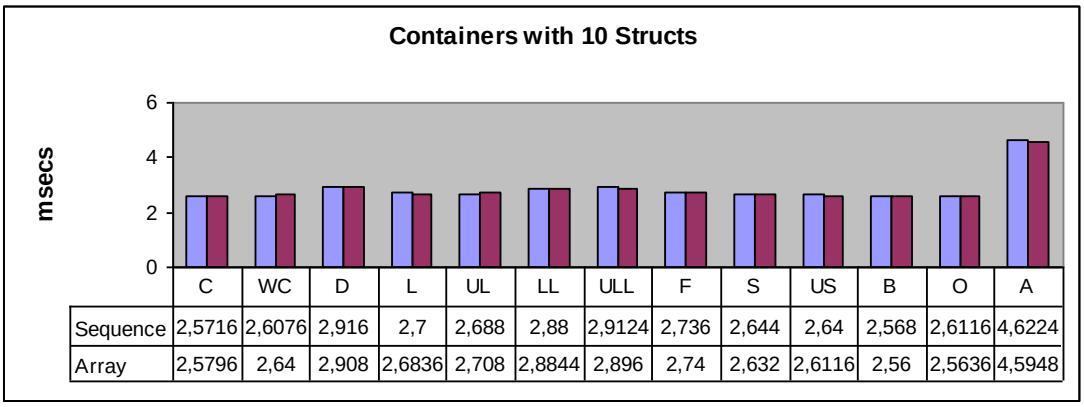
5.4.5.5 R_T_SG struct and struct container results

Figures 5.134 through 5.138 shows the results obtained.



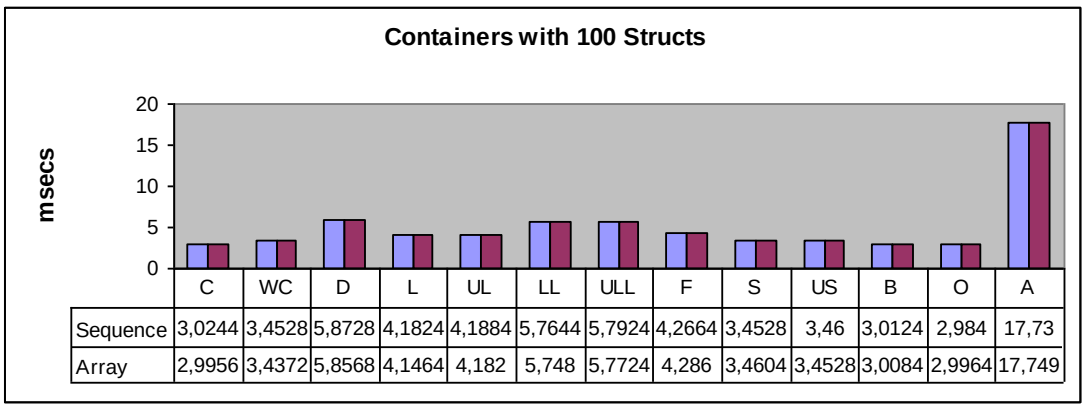
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Struct	2,516	2,4996	2,5676	2,4876	2,5316	2,5436	2,5156	2,5516	2,5276	2,5188	2,4752	2,508	2,9004
Sequence	2,5276	2,532	2,624	2,5116	2,5516	2,584	2,5996	2,6036	2,5316	2,4796	2,4636	2,588	2,868
Array	2,5116	2,52	2,648	2,5076	2,5076	2,5596	2,588	2,584	2,532	2,5276	2,4716	2,4596	2,8724

Figure 5.134 : R_T_SG Results for Structs and Containers with 1 Struct



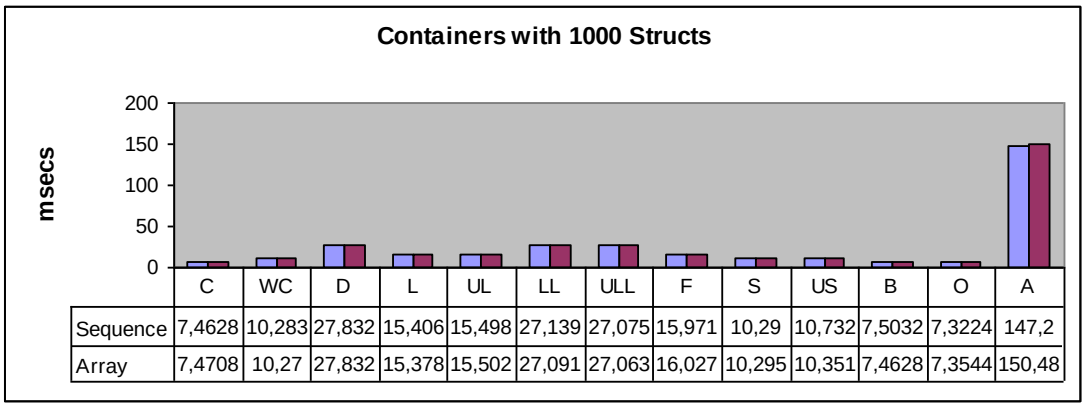
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,5716	2,6076	2,916	2,7	2,688	2,88	2,9124	2,736	2,644	2,64	2,568	2,6116	4,6224
Array	2,5796	2,64	2,908	2,6836	2,708	2,8844	2,896	2,74	2,632	2,6116	2,56	2,5636	4,5948

Figure 5.135 : R_T_SG Results for Containers with 10 Structs.



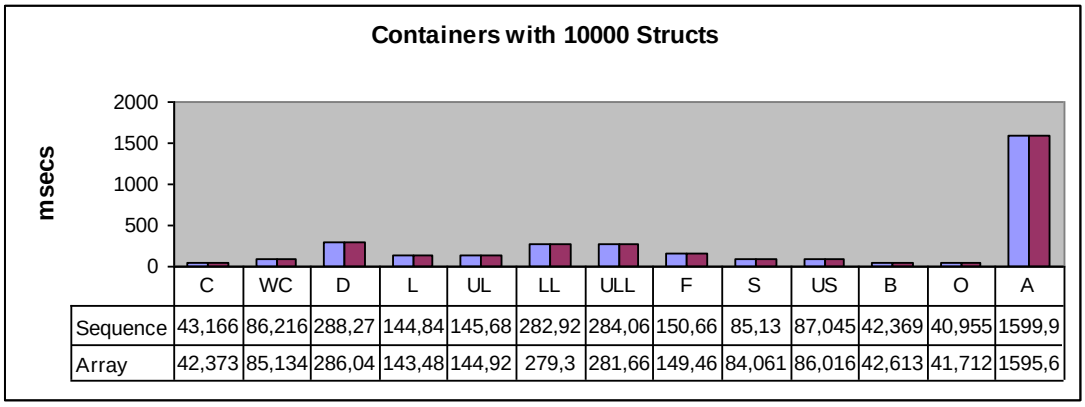
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	3,0244	3,4528	5,8728	4,1824	4,1884	5,7644	5,7924	4,2664	3,4528	3,46	3,0124	2,984	17,73
Array	2,9956	3,4372	5,8568	4,1464	4,182	5,748	5,7724	4,286	3,4604	3,4528	3,0084	2,9964	17,749

Figure 5.136 : R_T_SG Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	7,4628	10,283	27,832	15,406	15,498	27,139	27,075	15,971	10,29	10,732	7,5032	7,3224	147,2
Array	7,4708	10,27	27,832	15,378	15,502	27,091	27,063	16,027	10,295	10,351	7,4628	7,3544	150,48

Figure 5.137 : R_T_SG Results for Containers with 1000 Structs.



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	43,166	86,216	288,27	144,84	145,68	282,92	284,06	150,66	85,13	87,045	42,369	40,955	1599,9
Array	42,373	85,134	286,04	143,48	144,92	279,3	281,66	149,46	84,061	86,016	42,613	41,712	1595,6

Figure 5.138 : R_T_SG Results for Containers with 10000 Structs

5.4.5.6 about R_T_SG struct and struct container results

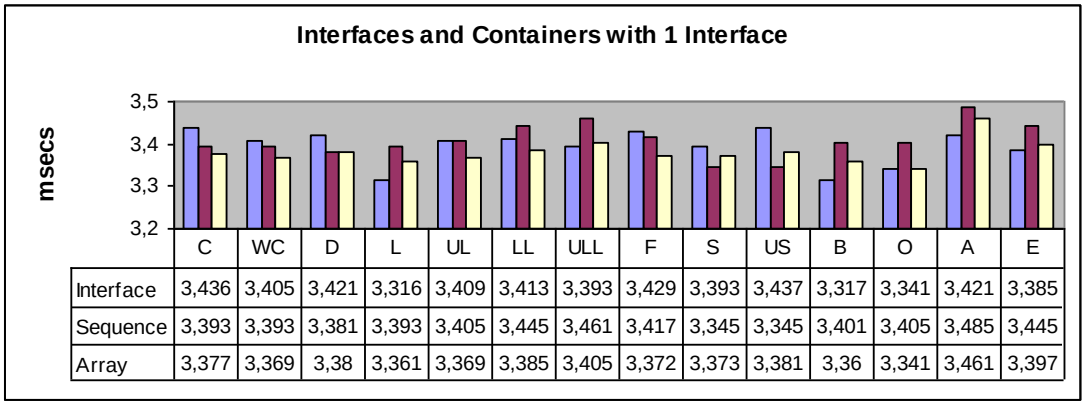
Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

- If we compare with L_T_SG results we see that local calls are faster than remote calls.

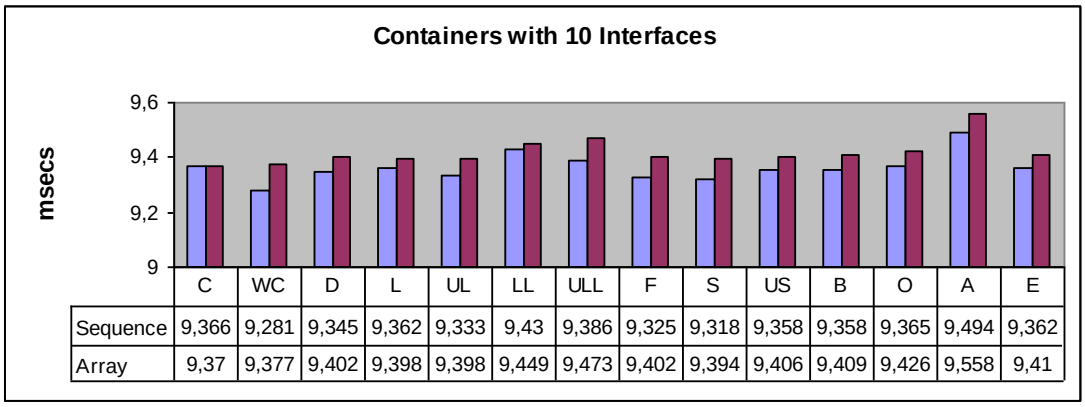
5.4.5.7 R_T_SG interface and interface container results

Figures 5.139 through 5.143 shows the results obtained.



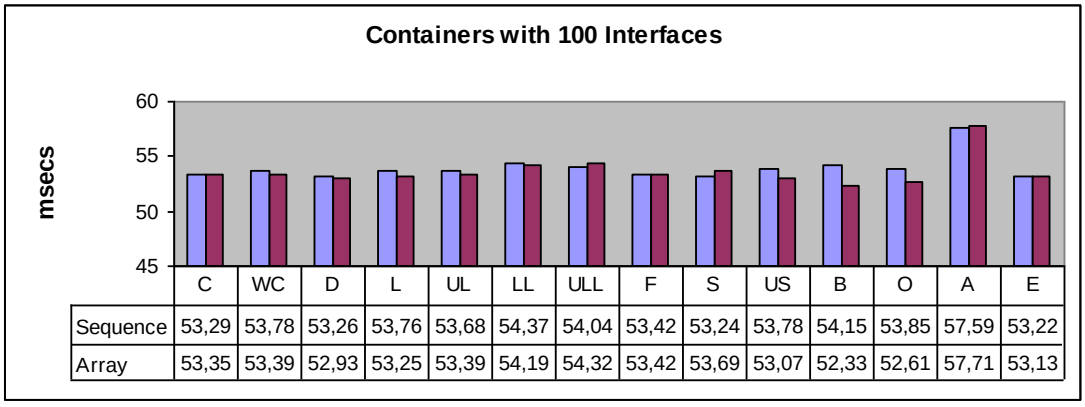
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	3,436	3,405	3,421	3,316	3,409	3,413	3,393	3,429	3,393	3,437	3,317	3,341	3,421	3,385
Sequence	3,393	3,393	3,381	3,393	3,405	3,445	3,461	3,417	3,345	3,345	3,401	3,405	3,485	3,445
Array	3,377	3,369	3,38	3,361	3,369	3,385	3,405	3,372	3,373	3,381	3,36	3,341	3,461	3,397

Figure 5.139 : R_T_SG Results for Interface and Containers with 1 Interface



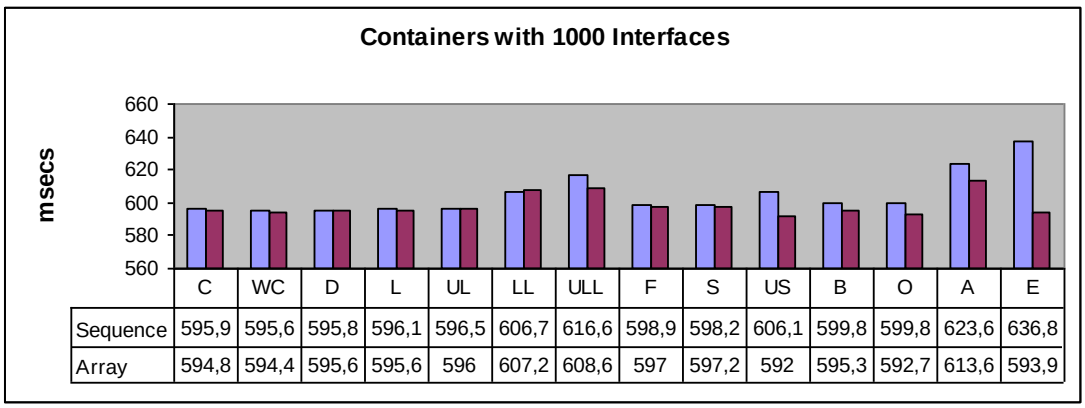
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	9,366	9,281	9,345	9,362	9,333	9,43	9,386	9,325	9,318	9,358	9,358	9,365	9,494	9,362
Array	9,37	9,377	9,402	9,398	9,398	9,449	9,473	9,402	9,394	9,406	9,409	9,426	9,558	9,41

Figure 5.140 : R_T_SG Results for Containers with 10 Interfaces



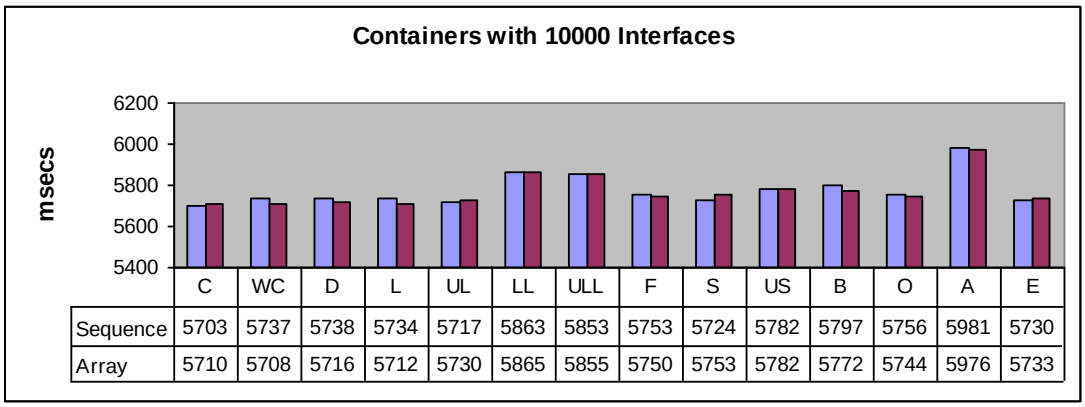
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	53,29	53,78	53,26	53,76	53,68	54,37	54,04	53,42	53,24	53,78	54,15	53,85	57,59	53,22
Array	53,35	53,39	52,93	53,25	53,39	54,19	54,32	53,42	53,69	53,07	52,33	52,61	57,71	53,13

Figure 5.141 : R_T_SG Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	595,9	595,6	595,8	596,1	596,5	606,7	616,6	598,9	598,2	606,1	599,8	599,8	623,6	636,8
Array	594,8	594,4	595,6	595,6	596	607,2	608,6	597	597,2	592	595,3	592,7	613,6	593,9

Figure 5.142 : R_T_SG Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	5703	5737	5738	5734	5717	5863	5853	5753	5724	5782	5797	5756	5981	5730
Array	5710	5708	5716	5712	5730	5865	5855	5750	5753	5782	5772	5744	5976	5733

Figure 5.143 : R_T_SG Results for Containers with 10000 Interfaces

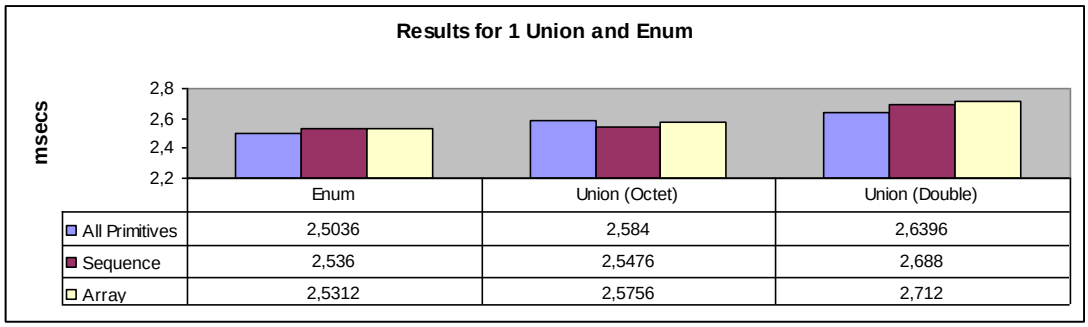
5.4.5.8 about R_T_SG interface and interface container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_SG results we see that local calls are faster than remote calls.

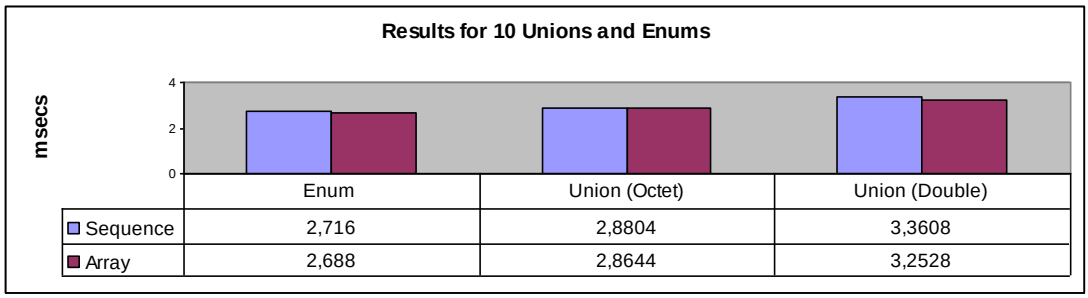
5.4.5.9 R_T_SG union and enum results

Figures 5.144 through 5.148 shows the results obtained.



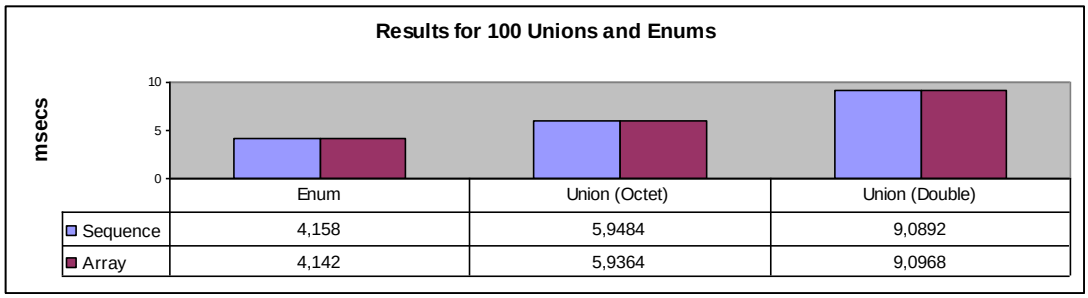
	Enum	Union (Octet)	Union (Double)
All Primitives	2,5036	2,584	2,6396
Sequence	2,536	2,5476	2,688
Array	2,5312	2,5756	2,712

Figure 5.144 : R_T_SG Results for Union, Enum and Containers with 1 Union and Enum



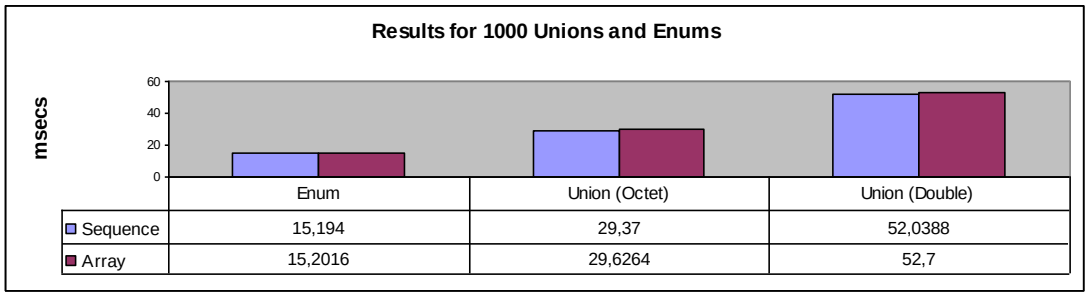
	Enum	Union (Octet)	Union (Double)
Sequence	2,716	2,8804	3,3608
Array	2,688	2,8644	3,2528

Figure 5.145 : R_T_SG Results for Containers with 10 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	4,158	5,9484	9,0892
Array	4,142	5,9364	9,0968

Figure 5.146 : R_T_SG Results for Containers with 100 Unions and Enums



	Enum	Union (Octet)	Union (Double)
Sequence	15,194	29,37	52,0388
Array	15,2016	29,6264	52,7

Figure 5.147 : R_T_SG Results for Containers with 1000 Unions and Enums

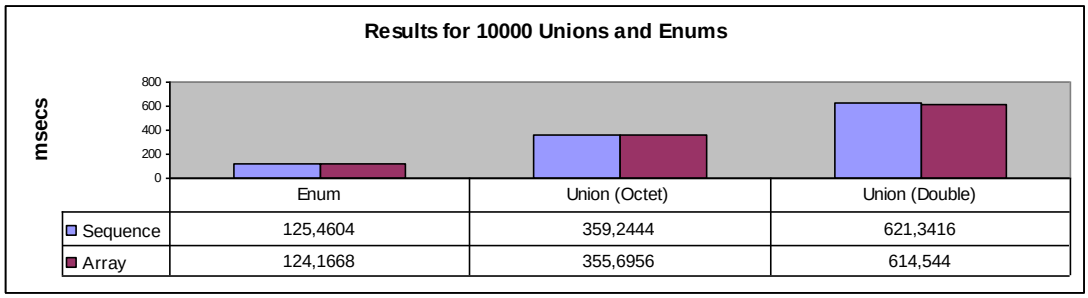


Figure 5.148 : R_T_SG Results for Containers with 10000 Unions and Enums

5.4.5.10 about R_T_SG union and enum results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_SG results we see that local calls are faster than remote calls.

5.4.6 twoway – only get results

We will briefly refer to these results as R_T_OG (Remote_Twoway_OnlyGet) results.

5.4.6.1 R_T_OG primitive and primitive container results

Figure 5.149 through 5.153 shows the results obtained for primitive types and containers with primitive types.

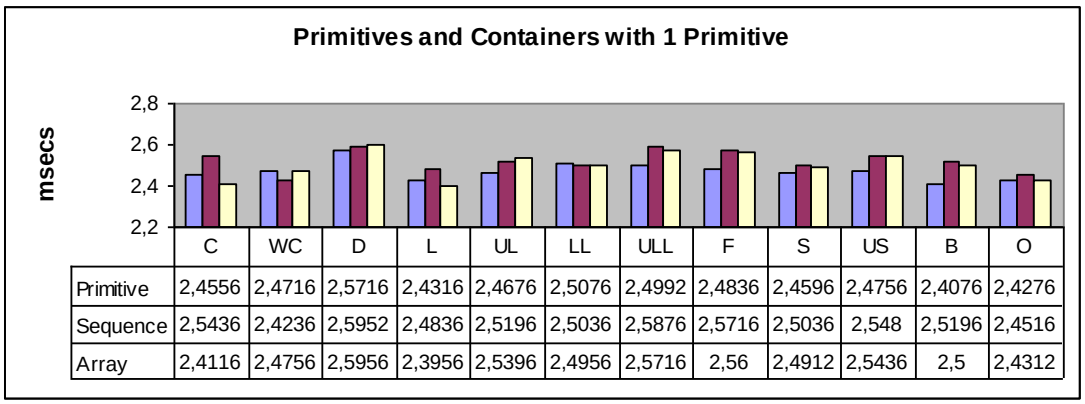
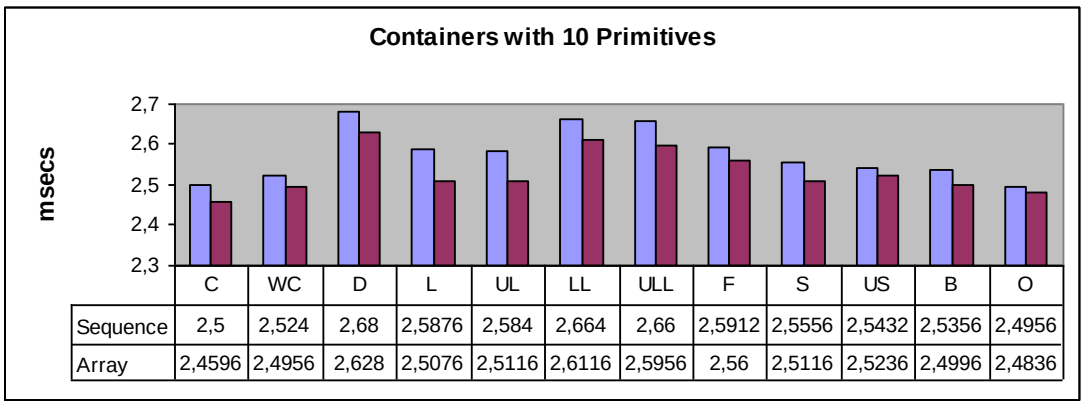
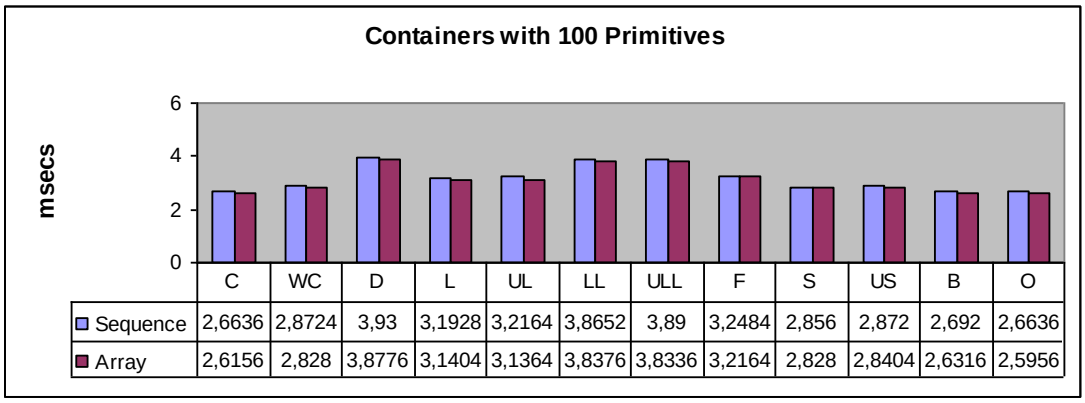


Figure 5.149 : R_T_OG Results for Primitives and Containers with 1 Primitive



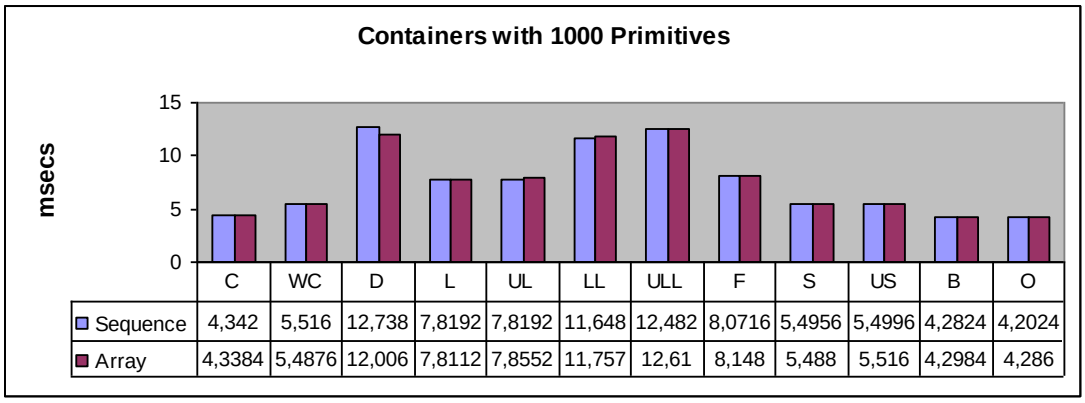
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,5	2,524	2,68	2,5876	2,584	2,664	2,66	2,5912	2,5556	2,5432	2,5356	2,4956
Array	2,4596	2,4956	2,628	2,5076	2,5116	2,6116	2,5956	2,56	2,5116	2,5236	2,4996	2,4836

Figure 5.150 : R_T_OG Results for Containers with 10 Primitives



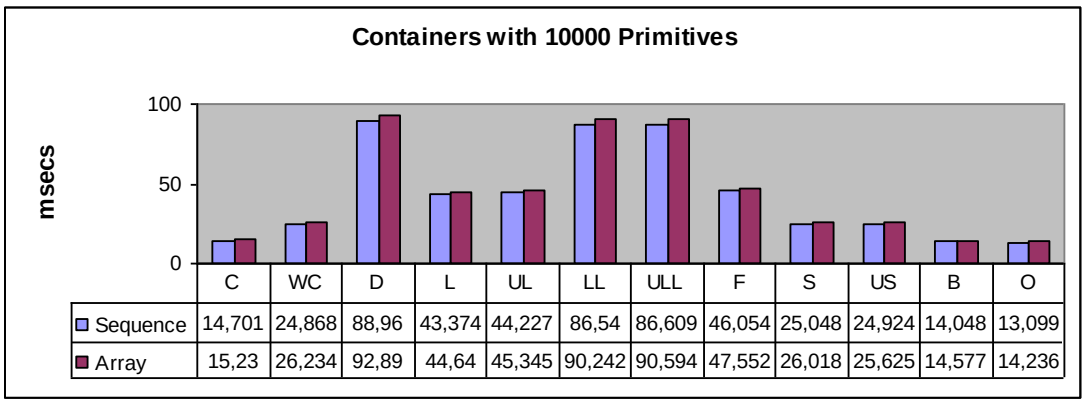
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	2,6636	2,8724	3,93	3,1928	3,2164	3,8652	3,89	3,2484	2,856	2,872	2,692	2,6636
Array	2,6156	2,828	3,8776	3,1404	3,1364	3,8376	3,8336	3,2164	2,828	2,8404	2,6316	2,5956

Figure 5.151 : R_T_OG Results for Containers with 100 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	4,342	5,516	12,738	7,8192	7,8192	11,648	12,482	8,0716	5,4956	5,4996	4,2824	4,2024
Array	4,3384	5,4876	12,006	7,8112	7,8552	11,757	12,61	8,148	5,488	5,516	4,2984	4,286

Figure 5.152 : R_T_OG Results for Containers with 1000 Primitives



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O
Sequence	14,701	24,868	88,96	43,374	44,227	86,54	86,609	46,054	25,048	24,924	14,048	13,099
Array	15,23	26,234	92,89	44,64	45,345	90,242	90,594	47,552	26,018	25,625	14,577	14,236

Figure 5.153 : R_T_OG Results for Containers with 10000 Primitives

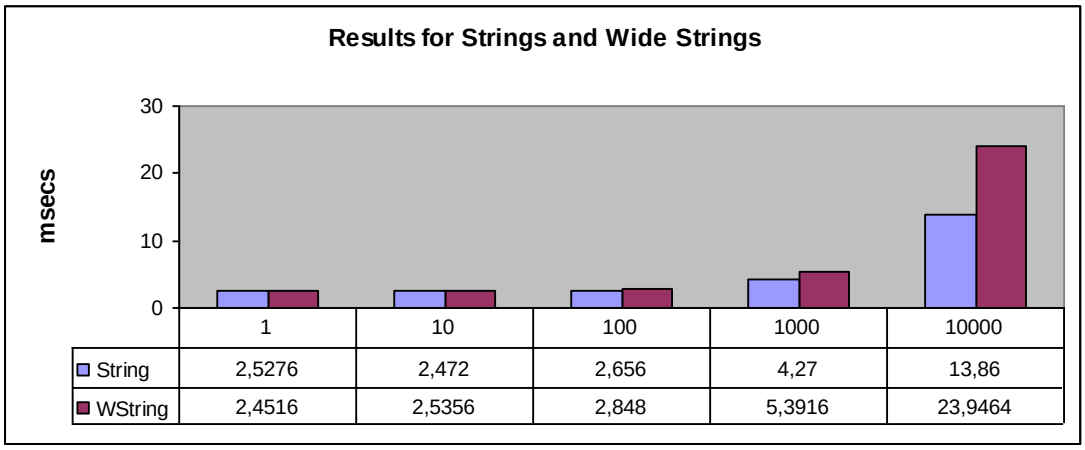
5.4.6.2 about R_T_OG primitive and primitive container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_SG results we see that local calls are faster than remote calls.

5.4.6.3 R_T_OG string and wide string results

Figure 5.154 shows the results obtained.



	1	10	100	1000	10000
String	2,5276	2,472	2,656	4,27	13,86
WString	2,4516	2,5356	2,848	5,3916	23,9464

Figure 5.154 : R_T_OG Results for Strings and Wide Strings

5.4.6.4 about R_T_OG string and wide string results

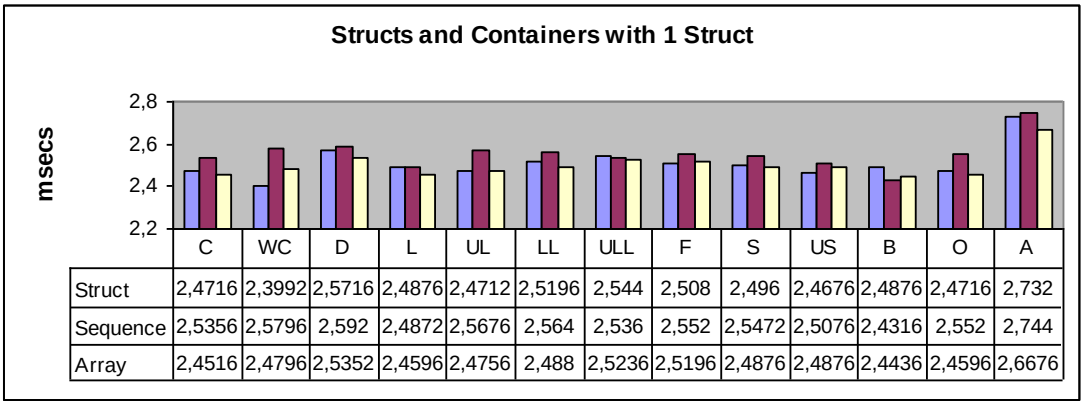
Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

- If we compare with L_T_SG results we see that local calls are faster than remote calls.

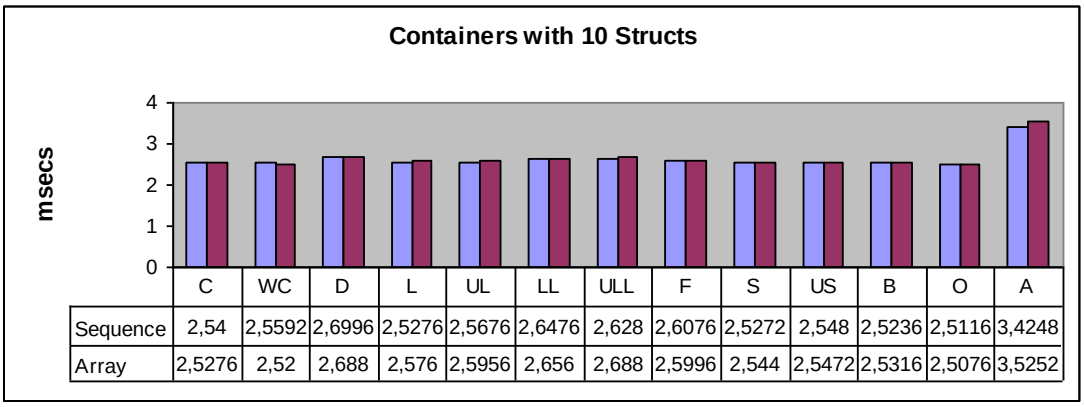
5.4.6.5 R_T_OG struct and struct container results

Figures 5.155 through 5.159 shows the results obtained.



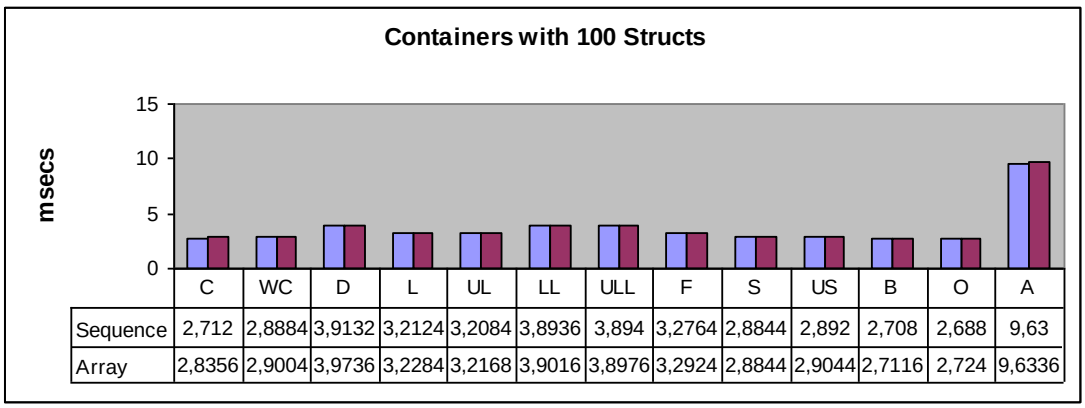
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Struct	2,4716	2,3992	2,5716	2,4876	2,4712	2,5196	2,544	2,508	2,496	2,4676	2,4876	2,4716	2,732
Sequence	2,5356	2,5796	2,592	2,4872	2,5676	2,564	2,536	2,552	2,5472	2,5076	2,4316	2,552	2,744
Array	2,4516	2,4796	2,5352	2,4596	2,4756	2,488	2,5236	2,5196	2,4876	2,4876	2,4436	2,4596	2,6676

Figure 5.155 : R_T_OG Results for Structs and Containers with 1 Struct



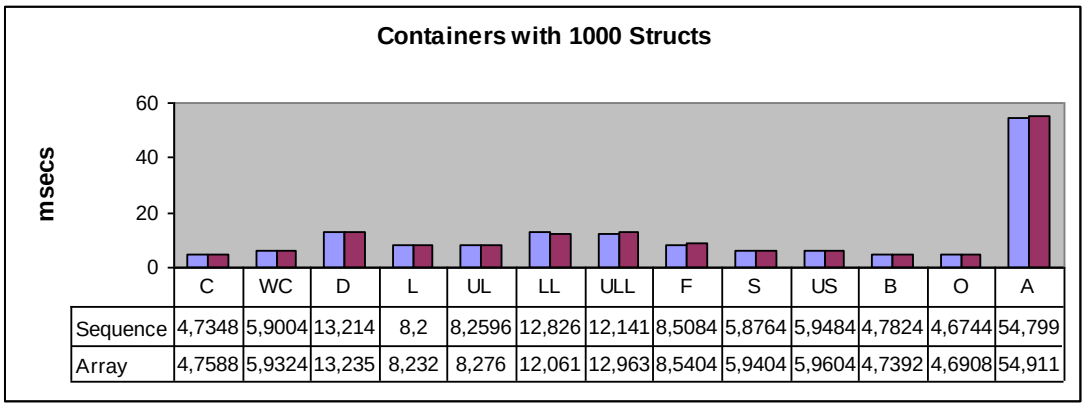
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,54	2,5592	2,6996	2,5276	2,5676	2,6476	2,628	2,6076	2,5272	2,548	2,5236	2,5116	3,4248
Array	2,5276	2,52	2,688	2,576	2,5956	2,656	2,688	2,5996	2,544	2,5472	2,5316	2,5076	3,5252

Figure 5.156 : R_T_OG Results for Containers with 10 Structs.



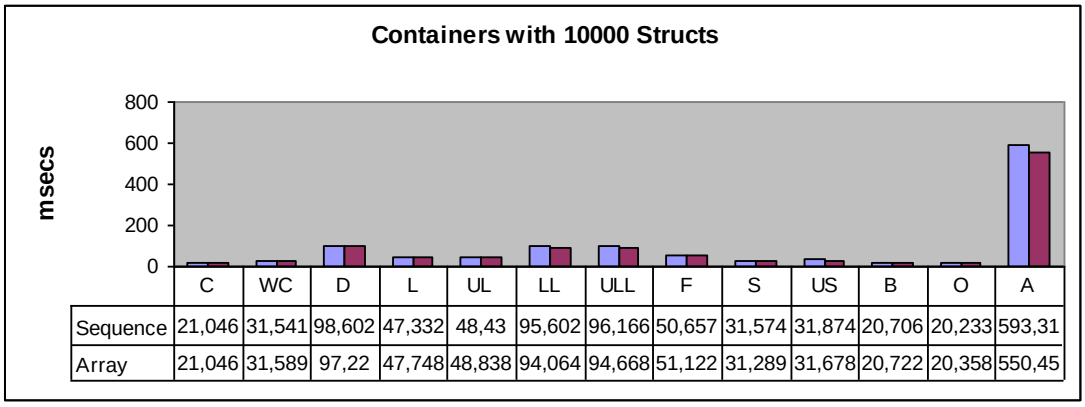
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	2,712	2,8884	3,9132	3,2124	3,2084	3,8936	3,894	3,2764	2,8844	2,892	2,708	2,688	9,63
Array	2,8356	2,9004	3,9736	3,2284	3,2168	3,9016	3,8976	3,2924	2,8844	2,9044	2,7116	2,724	9,6336

Figure 5.157 : R_T_OG Results for Containers with 100 Structs



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	4,7348	5,9004	13,214	8,2	8,2596	12,826	12,141	8,5084	5,8764	5,9484	4,7824	4,6744	54,799
Array	4,7588	5,9324	13,235	8,232	8,276	12,061	12,963	8,5404	5,9404	5,9604	4,7392	4,6908	54,911

Figure 5.158 : R_T_OG Results for Containers with 1000 Structs.



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A
Sequence	21,046	31,541	98,602	47,332	48,43	95,602	96,166	50,657	31,574	31,874	20,706	20,233	593,31
Array	21,046	31,589	97,22	47,748	48,838	94,064	94,668	51,122	31,289	31,678	20,722	20,358	550,45

Figure 5.159 : R_T_OG Results for Containers with 10000 Structs

5.4.6.6 about R_T_OG struct and struct container results

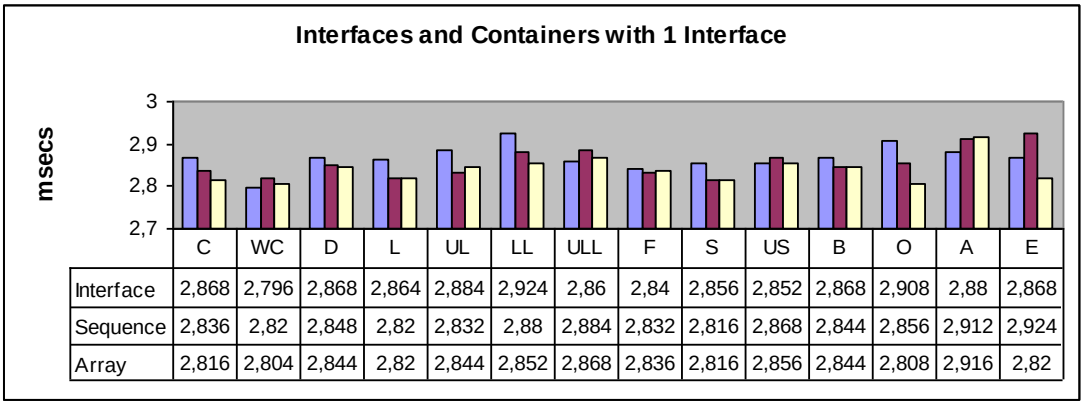
Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.

- If we compare with L_T_SG results we see that local calls are faster than remote calls.

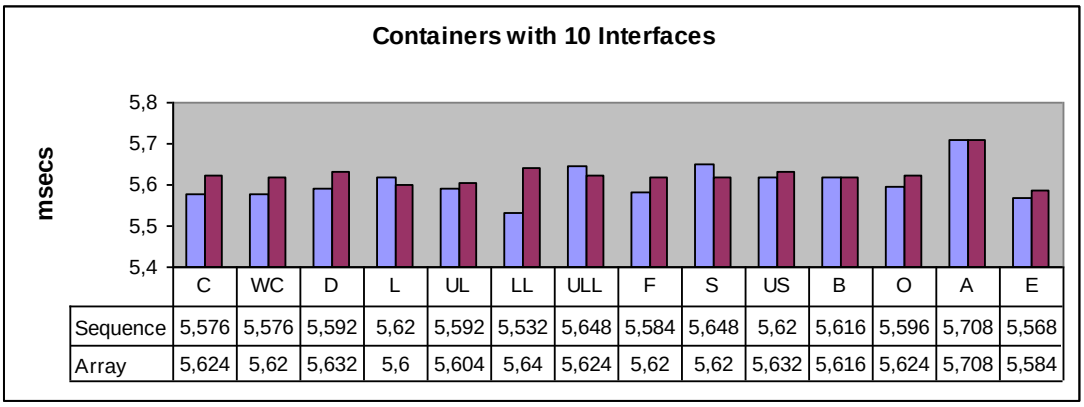
5.4.6.7 R_T_OG interface and interface container results

Figures 5.160 through 5.164 shows the results obtained.



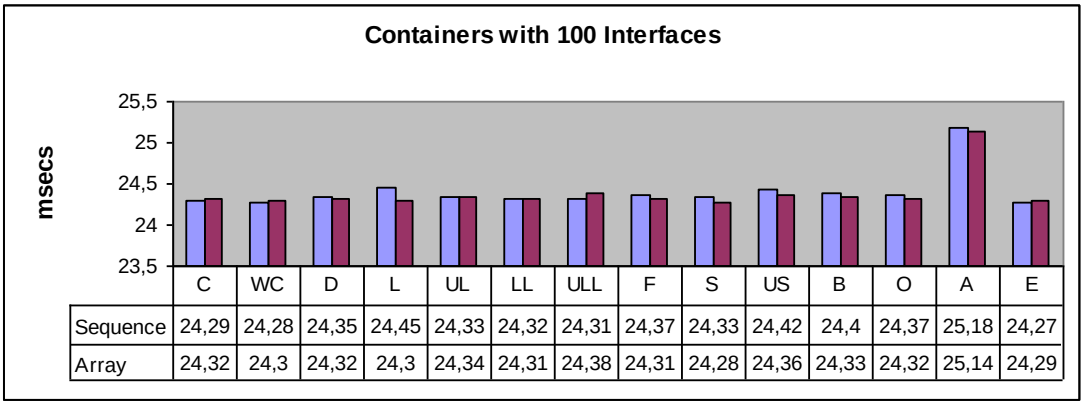
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Interface	2,868	2,796	2,868	2,864	2,884	2,924	2,86	2,84	2,856	2,852	2,868	2,908	2,88	2,868
Sequence	2,836	2,82	2,848	2,82	2,832	2,88	2,884	2,832	2,816	2,868	2,844	2,856	2,912	2,924
Array	2,816	2,804	2,844	2,82	2,844	2,852	2,868	2,836	2,816	2,856	2,844	2,808	2,916	2,82

Figure 5.160 : R_T_OG Results for Interface and Containers with 1 Interface



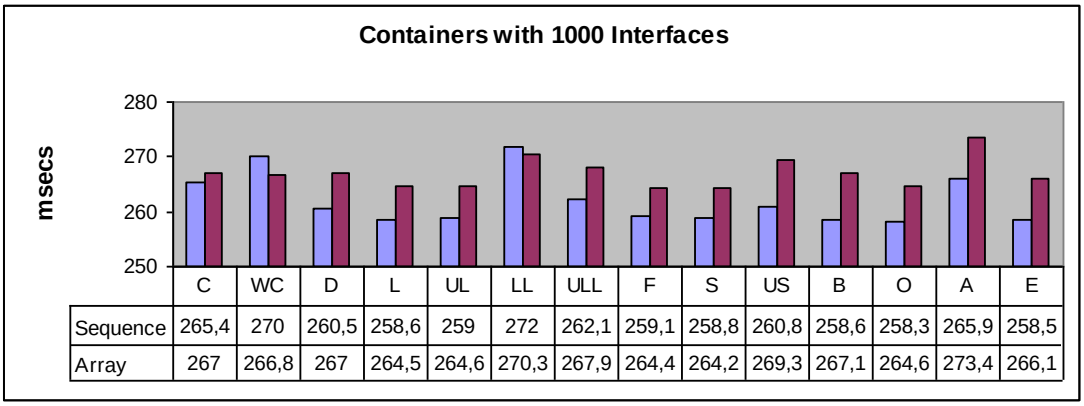
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	5,576	5,576	5,592	5,62	5,592	5,532	5,648	5,584	5,648	5,62	5,616	5,596	5,708	5,568
Array	5,624	5,62	5,632	5,6	5,604	5,64	5,624	5,62	5,62	5,632	5,616	5,624	5,708	5,584

Figure 5.161 : R_T_OG Results for Containers with 10 Interfaces



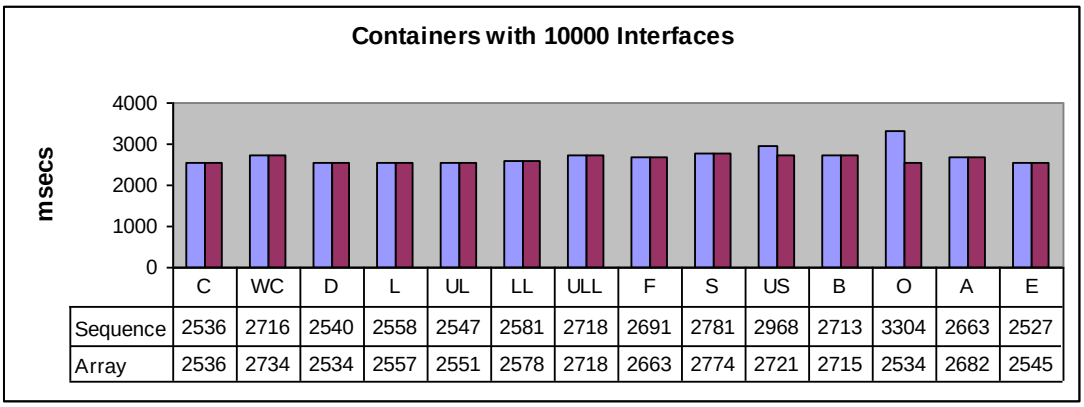
	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	24,29	24,28	24,35	24,45	24,33	24,32	24,31	24,37	24,33	24,42	24,4	24,37	25,18	24,27
Array	24,32	24,3	24,32	24,3	24,34	24,31	24,38	24,31	24,28	24,36	24,33	24,32	25,14	24,29

Figure 5.162 : R_T_OG Results for Containers with 100 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	265,4	270	260,5	258,6	259	272	262,1	259,1	258,8	260,8	258,6	258,3	265,9	258,5
Array	267	266,8	267	264,5	264,6	270,3	267,9	264,4	264,2	269,3	267,1	264,6	273,4	266,1

Figure 5.163 : R_T_OG Results for Containers with 1000 Interfaces



	C	WC	D	L	UL	LL	ULL	F	S	US	B	O	A	E
Sequence	2536	2716	2540	2558	2547	2581	2718	2691	2781	2968	2713	3304	2663	2527
Array	2536	2734	2534	2557	2551	2578	2718	2663	2774	2721	2715	2534	2682	2545

Figure 5.164 : R_T_OG Results for Containers with 10000 Interfaces

5.4.6.8 about R_T_OG interface and interface container results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_SG results we see that local calls are faster than remote calls.

5.4.6.9 R_T_OG union and enum results

Figures 5.165 through 5.169 shows the results obtained.

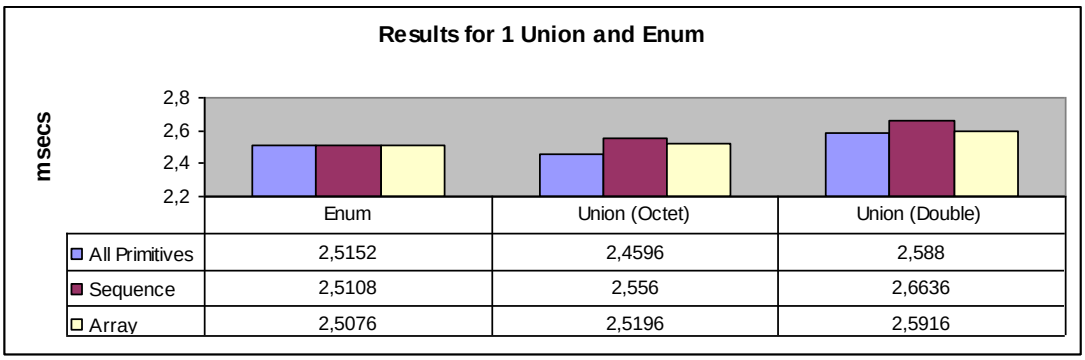


Figure 5.165 : R_T_OG Results for Union, Enum and Containers with 1 Union and Enum

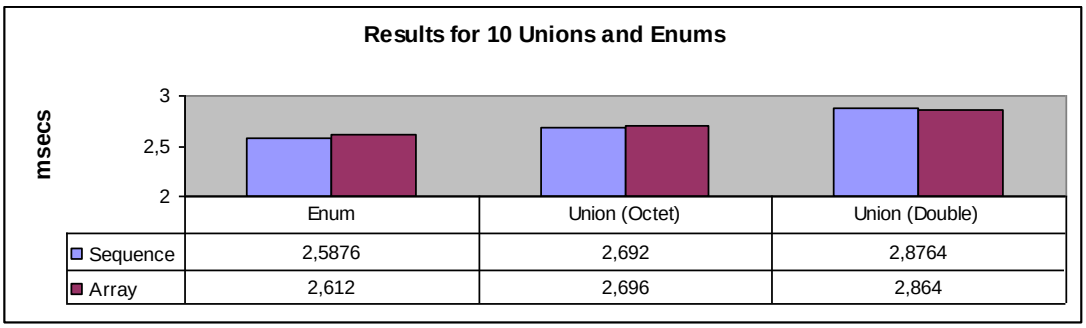


Figure 5.166 : R_T_OG Results for Containers with 10 Unions and Enums

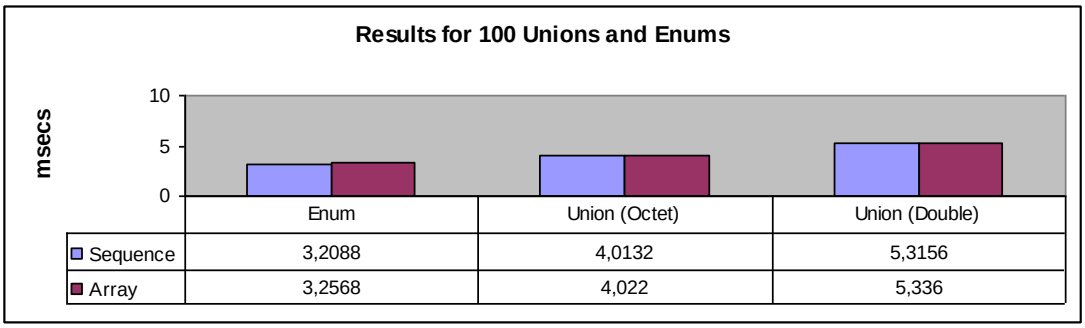


Figure 5.167 : R_T_OG Results for Containers with 100 Unions and Enums

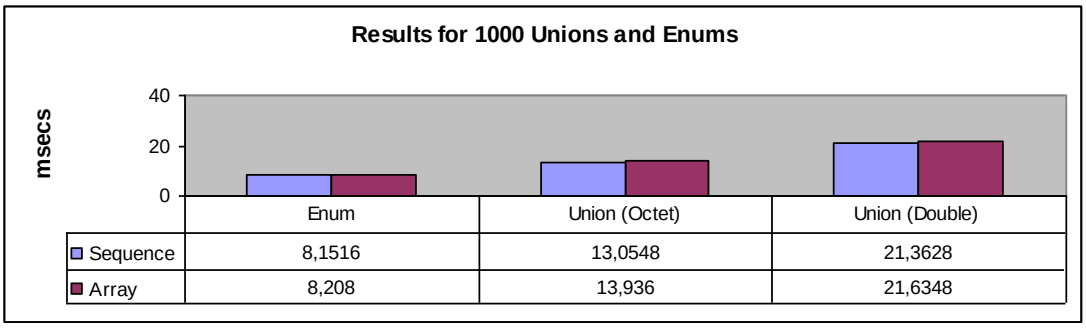


Figure 5.168 : R_T_OG Results for Containers with 1000 Unions and Enums

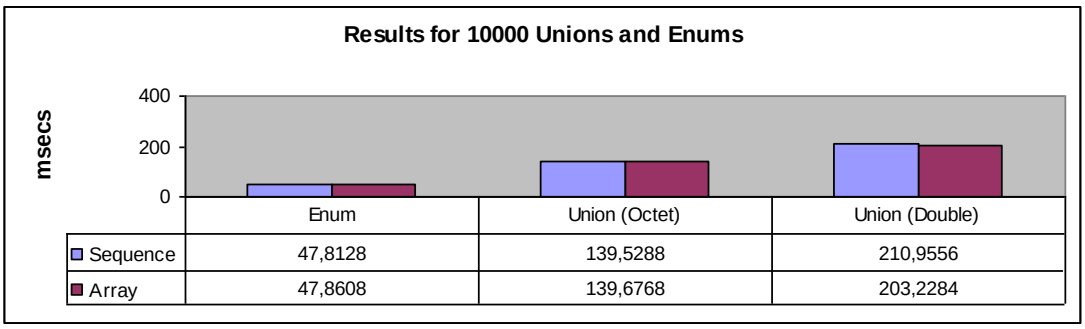


Figure 5.169 : R_T_OG Results for Containers with 10000 Unions and Enums

5.4.6.10 about R_T_OG union and enum results

Some conclusions from the results we have taken are :

- If we consider the aspects other than the local/remote distinction, we have the same results with local ones.
- If we compare with L_T_SG results we see that local calls are faster than remote calls.

6. CONCLUSIONS AND FUTURE WORK

We had a wide benchmark and ran it. We have some conclusions from our study and future work plans. We will briefly mention these at the following lines.

6.1 Conclusions

We tested nearly all static IDL constructs in this study. We have a bulk of raw data, and comparisons can be made on these data. We only give conclusions for some of them. Whoever wants can deduce the conclusions he/she needs from our data.

Our results (generally) show that :

- Local calls are faster than remote calls for big-sized data. For small-sized data, remote calls are faster.
- Oneway invocations are faster than twoway invocations. But oneways are unreliable and some of them could not complete the test. On the other hand, all the twoway calls successfully completed the measurements.
- We see the nearly same results for small sized data for all the flows of data. But for the larger sizes the descending order is from client to server and server to client back, from client to server and from server to client.
- For the primitive types we have the close results for small number of parameters and results are ordered with sizes of types for larger number of parameters.
- For constructed types :
    - We have the same results for primitives with structs. For small sizes. But for the big sizes, structs perform worse.
    - We have the same results for unsigned long with enums.
    - We have extremely slow passing of parameters with interfaces.
    - Passing an octet within a union takes less time than passing a double.
- For the container types :
    - We have the same results with sequences and arrays.
    - We have the same results for strings with chars.

- We have the same results for wstrings with wchars.

6.2 Future Work

As we mentioned, we have only tested the static CORBA, but *common* phrase in the CORBA means static and dynamic [47]. So, our benchmark can be applied to dynamic CORBA.

Also we mentioned that we applied our benchmark to the most commonly available CORBA/Java ORB worldwide. When we were studying on the release 1.3.1, Sun released the new version of its SDK, version 1.4, and it contains a different approach for the object adapter : it uses POA even though it also supports old style. So, our benchmark can be applied to this new release of Java IDL.

Java IDL is not the only CORBA/Java ORB. There are a lot of CORBA/Java ORBs on the market (e.g, JavaORB, JacORB, OpenORB, Visibroker for Java, Voyager ORB, Engine Room CORBA, ... etc) and by applying the benchmark to these ORBs a comparison between the ORBs can be conducted.

We have only thought of marshalling/demarshalling of parameters. But there are another ways of comparisons. So, we can extend our benchmark to cover, for example, dispatching, survivability and reliability as a future study.

REFERENCES

[1] Rosenberger, J.L., 1998, Teach Yourself CORBA in 14 Days, Sams Publishing**.**

[2] Orfali, R., Harkey, D. and Edwards, J., 1996, The Essential Client / Server Survival Guide, John Wiley, New York.

[3] Lewandowski, S. M., 1998, Frameworks for Component-Based Client/Server Computing, ACM Computing Surveys, Vol. 30, No. 1 (March).

[4] Wegner, P., 1997, Frameworks For Active Compound Documents, Brown University Department of Computer Science, Providence, RI.

[5] Wegner, P., 1997, Why interaction is more powerful than algorithms., Commun. ACM (May), 80-91.

[6] Orfali, R., Harkey, D. and Edwards, J., 1996, The Essential Distributed objects Survival Guide, John Wiley, New York.

[7] OMG , 2001, The Common Object Request Broker : Architecture and Specification, revision 2.6, December.

[8] Fay-Wolfe, W., DiPippo, L. C., Cooper, G., Johnston, R., Kortmann, P. and Thuraisingham, B., 2000, Real-Time CORBA , IEEE Transactions on Parallel and Distributed Systems.

[9] Orfali, R., Harkey, D. and Edwards, J., 1997, Instant CORBA, John Wiley, USA.

[10] Lee, D. , 2002, CORBA "Fitting the pieces together", http://www.scit.wlv.ac.uk/~cm1924/cp3025/distrib/reading/corba/corba8/corba11.html

[11] Karlsson, M., 1999, Orbix versus TAO (A comparison between two CORBA implementations), Master Thesis, Uppsala University, Sweden.

[12] Vinoski, S., 1997, CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments, IEEE Communications Magazine, Vol. 35, No. 2, February.

[13] Henning, M., 1998, Binding, Migration, and Scalability in CORBA, Communications of the ACM, Volume 41, No 10, October.

[14] OMG , 2001, IDL to Java Language Mapping Specification, version 1.1, June.

[15] Schmidt, D. C., Levine, D. L. and Mungee, S., 1998, The Design of the TAO Real-Time Object Request Broker, Computer Communications, Elsivier Science, Volume 21, No 4, April.

[16] OMG, 1997, Specification of the Portable Object Adapter (POA), OMG Document orbos/97-05-15 ed., June.

[17] Pyarali, I. and Schmidt, D. C., 1998, An Overview of the CORBA Portable Object Adapter, Special Issue on CORBA in the ACM StandardView magazine, Volume 6, Number 1, March.

[18] OMG, 2002, Catalog of CORBAservices Specifications, http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm

[19] OMG, 2002, Life Cycle Service Specification, version 1.2, formal/02-09-01, September.

[20] OMG, 2002, Persistent State Service Specification, version 2.0, formal/02-09-06, September.

[21] OMG, 2002, Naming Service Specification, version 1.2, formal/02-09-02, September.

[22] OMG, 2001, Event Service Specification, version 1.1, March.

[23] OMG, 2000, Concurrency Service Specification, version 1.0, April.

[24] OMG, 2002, Transaction Service Specification, version 1.3, formal/02-08-07, September.

[25] OMG, 2000, Relationship Service Specification, version 1.0, April.

[26] OMG, 2000, Externalization Service Specification, version 1.0, April.

[27] OMG, 2000, Query Service Specification, version 1.0, April.

[28] OMG, 2000, Licensing Service Specification, version 1.0, April.

[29] OMG, 2000, Property Service Specification, version 1.0, April.

[30] OMG, 2002, Time Service Specification, version 1.1, May.

[31] OMG, 2002, Security Service Specification, version 1.8, March.

[32] OMG, 2000, Trading Object Service Specification, version 1.0, May.

[33] OMG, 2002, Object Collection Specification, version 1.0.1, formal/02-08-03, August.

[34] Minton, G., 1997, IIOP Specification : A Closer Look, Unix Review, January.

[35] OMG, 2002, CORBA FAQ, http://www.omg.org/gettingstarted/corbafaq.htm

[36] Siegel, J., 1996, CORBA Fundamentals and Programming, John Wiley, USA.

[37] Buble, A., 1999, Comparing CORBA Implementations, Master Thesis, Charles University, Prague, Czech Republic.

[38] Gopinath, A., 1993, Performance Measurement and Analysis of Real-Time CORBA Endsystems, Master Thesis, Cochin University of Science and Technology, India.

[39] Gokhale, A. and Schmidt, D. C., 1996, The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks, GLOBECOM, London, November 18-22.

[40] Gokhale, A. and Schmidt, D. C., 1998, Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance, HICSS, Maui, Hawaii, January 9.

[41] Gokhale, A. and Schmidt, D. C., 1997, Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA, proceedings of GLOBECOM, Phoenix, AZ, November.

[42] Gokhale, A. and Schmidt, D. C., 1997, Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks, Proceedings of the International Conference on Distributed Computing Systems, Baltimore, MD, May 27-30.

[43] Hirano, S., Yasu, Y. and Igarashi, H., 1998, Performance Evaluation of Popular Distributed Object Technologies for Java, ACM Workshop on Java for High-Performance Network Computing, February 28.

[44] Brose, G., 2002, JacORB Performance compared, http://www.jacorb.org/performance/index.html

[45] OMG, 1999, Benchmark PSIG White Paper on Benchmarking, version 1.0, OMG Document bench/99-12-01, December 27.

[46] Gokhale, A. And Schmidt, D. C., 1996, Measuring the performance of Communication Middleware on High-Speed Networks, SIGCOMM,August.

[47] Orfali, R. and Harkey, D., 1998, Client/Server Programming with JAVA and CORBA, John Wiley, USA.

[48] Baker, S., 1997, CORBA Distributed Objects Using ORBİX, ACM Press.

[49] Flanagan, D., Farley, J. and Crawford, W., 1999, Java Enterprise in a Nutshell, O'reilly, USA.

CURRICULUM VITAE

Tacettin Ayar was born in 1976. He received his Bsc degree from Middle East Technical University Computer Engineering Department, in 1999. He is now a research assistant at Istanbul Technical University Computer Engineering Department.