Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications

Xingchen Chu, Krishna Nadiminti, Chao Jin, Srikumar Venugopal, Rajkumar Buyya

***Gri**d Computing and **D**istributed **S**ystems (GRIDS) Laboratory*
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
Email: {xchu, kna, chaojin, srikumar, raj}@csse.unimelb.edu.au

Abstract

In this paper, we present the design of Aneka, a .NET based service-oriented platform for desktop grid computing that provides: (i) a configurable service container hosting pluggable services for discovering, scheduling and balancing various types of workloads and (ii) a flexible and extensible framework/API supporting various programming models including threading, batch processing, MPI and dataflow. Users and developers can easily use different programming models and the services provided by the container to run their applications over desktop Grids managed by Aneka. We present the implementation of both the essential and advanced services within the platform. We evaluate the system with applications using the grid task and dataflow models on top of the infrastructure and conclude with some future directions of the current system.

1. Introduction

The term "desktop grid computing" refers to systems that harness the unused CPU cycles of desktop Personal Computers (PCs) connected over a corporate network or the Internet to accelarate application performance. Within an enterprise, desktop grids allow an organisation to improve the utilization of its IT resources, by allowing it to harness the power of unused PCs for computational tasks without affecting productivity of the PC users. While the notion of desktop grid computing is well-understood, there are a lot of challenges in realising such a system. Some of the key issues include: resource management, failure management, reliability, application composition, scheduling and security [3].

Our previous efforts in desktop grid computing resulted in Alchemi [6], a Microsoft .NET-based framework which provides an object-oriented threading API and file-based grid job model to create grid applications over various desktop PCs. However, Alchemi was limited to a master-slave architecture, and lacked the flexibility for efficiently implementing other parallel programming models such as message-passing and dataflow. We have improved upon Alchemi to create a service-oriented, desktop grid system called Aneka, also developed on top of the .NET platform. This paper describes its design and implementation.

Aneka was conceived with the aim of providing a set of services that make grid construction and development of applications as easy as possible without sacrificing flexibility, scalability, reliability and extensibility. The key features supported by Aneka are:

- A configurable container enabling pluggable services, persistence solutions, security implementations, and communication protocols;
- decentralized architecture peering individual nodes;
- multiple programming models including object-oriented grid threading programming model (fine-grained abstraction), file-based grid task model (coarse-grained abstraction) for grid-enabling legacy applications, and dataflow model for coarse-grained data intensive applications;
- multiple authentication/authorisation mechanisms such as role-based security, X.509 certificates/GSI proxy and Windows domain-based authentication;
- multiple persistence options including RDBMS, ODBMS and XML or flat files;
- Web services interface supporting the task model for interoperability with custom grid middleware (e.g. for creating a global, cross-platform grid

environment via a resource broker) and non-.NET programming languages.

The rest of the paper is organized as follows: First, we discuss related work; we follow this with a presentation of the architecture of our Aneka platform along with detailed description of its services. Then, we present the detailed implementations of the core services and discuss the threading, task and dataflow application models supported in our system along with the performance evaluation of running sample applications with different models in our container environment. Finally, we conclude the paper with future directions.

2. Related Work

The idea of using under-utilized networked PCs for performing computational tasks is well-established and there are several projects in this area. Some of the more well-known ones are the @Home projects (SETI@Home[2], Folding@Home[13]), Entropia[3], XtremeWeb[5], Alchemi[6] and SZTAKI Desktop Grid [7]. The approach followed by SETI@Home and other related projects is to dispatch workloads consisting of data to be analysed, from a central server to millions of clients running on desktops around the world, and was specific to the processing of astronomy application data. These and similar projects are considered as the "first generation" of desktop grids[9]. The infrastructure underlying SET@Home was generalized to create the Berkeley Open Infrastructure for Internet Computing (BOINC)[8]. BOINC allows desktop clients to select the project to which they wanted to donate idle computing power to and is used by many scientific distributed computing projects (e.g. climateprediction.net[15], SZTAKI Desktop Grid [7]).

Entropia[3] and United Devices[4] are similar systems in the sense that they create a Windows desktop grid environment using an architecture in which a central job manager is responsible for decomposing the jobs and distributing them to the desktop clients. XtremWeb[5] also provides a centralized architecture which consists of three entities, the coordinator, the worker and the clients to create a XtremWeb network. Clients submit tasks to the coordinator, along with binaries and optional parameter files and retrieve the results for the end user. The workers in the network are the software components that actually execute and compute the tasks. As mentioned previously, Alchemi also follows a master-slave architecture consisting of managers and executors wherein the former can either connect to the executors or other managers to create a hierarchical network

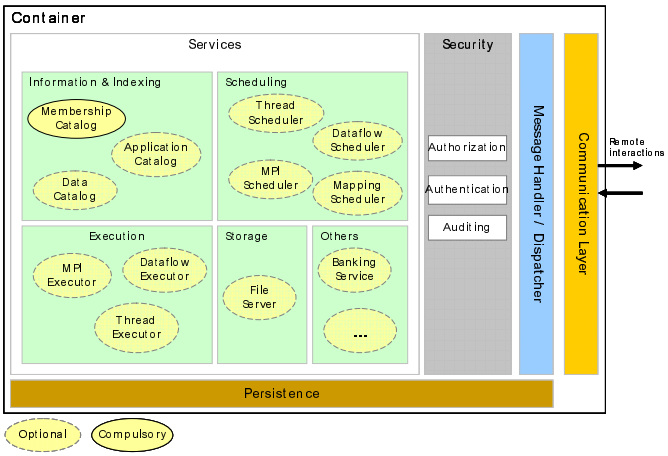

Figure 1. Aneka Single Node Architecture

structure. The executor can run in either a dedicated or a non-dedicated mode.

According to Capello[9], these grid systems can be categorized as the second generation of desktop grid. They are built with a rigid architecture with little or no modularity and extensibility; and components such as the job scheduler, data management and communication protocols are dedicated to the system. The execution nodes need to directly communicate with a central master node in both the centralized and the hierarchical architecture. The major problems with this approach are latency, performance bottlenecks, single point of vulnerability of the system, and high cost of the centralised server. In addition, it lacks the capabilities required for advanced applications that involve complex dependencies between parallel execution units, and the flexibility required for implementing various types of widely-employed parallel models such as message-passing and dataflow.

The limitations of the above systems motivate the introduction of a new architecture for desktop grid computing in which the capabilities required for different applications are separated from the message-passing infrastructure so that the platform is able to support different configurations as required. In the recent past, the Grid community has standardized on the Web Services Resource Framework (WSRF)[16] in which the different functionalities offered by a grid resource are made available through loosely-coupled, stateful service instances hosted in a Web-enabled container that provides the basic infrastructure. However, WSRF encompasses a lot of standards designed for wide-area grid infrastructure.

This paper therefore, presents the following contributions:

1. The design and implementation of a lightweight, service-oriented, desktop computing platform that

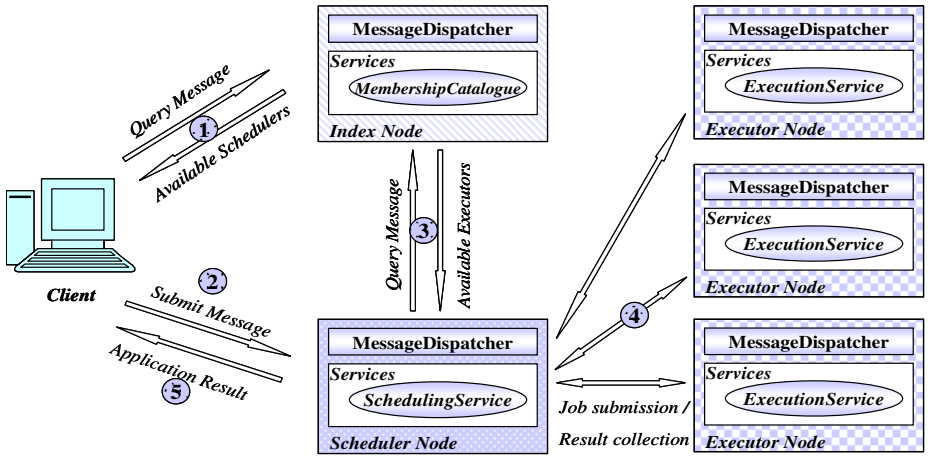

Figure 2. Application running in Aneka environment

consists of a configurable container hosting pluggable services

2. The use of this platform to realise multiple parallel and distributed programming models. This is illustrated through two case studies implementing task farming and dataflow computing models.

Aneka's design makes it very flexible and extensible so that multiple application models, security solutions, communication protocols and persistence can be supported without affecting an existing Aneka ecosystem. Therefore, Aneka is an example of the "third generation" of desktop grids[9].

3. Architecture Overview

Aneka provides a highly modular architecture, as shown in Figure 1. An Aneka node consists of an instance of a configurable container that hosts several compulsory services and any number of optional services. The compulsory services provide functions such as security, persistence mechanisms, and communication protocols, and are together called as the base infrastructure. The optional services include specific executors for different types of programming models and/or associated schedulers. The following sections will give more details about each of these components within Aneka.

3.1. Container

The Aneka container is designed as a runtime host and coordinator for other components. The container uses the Inverse of Control (IoC)[14] concept to inject dependencies at runtime. Details of compulsory and optional services, security, persistence, and associated communication protocols are specified in an XML configuration file which is read by the container when it is initialized. The main responsibility of the container is to initialize the services and present itself as a single point for communication to the rest of the system. However, to improve the reliability and flexibility of the system, neither the container nor the hosted services are dependent on each other. This is so that a malfunctioning service will not affect the others and/or the container. Also, this enables the administrator of an Aneka system to easily configure and manage existing services or introduce new ones into a container.

3.2. Base Infrastructure

The base infrastructure for the runtime framework provides message dispatching, security, communication, logging, network membership, and persistence functions that are then used by the hosted services. However, it is possible to substitute different implementations of these functions as per requirements of the services. For example, users can choose either a light- weight security mechanism such as role-based or a certificate-based security such as X.509 certificate by modifying the configuration file, and the runtime system will automatically inject them on-demand by the services. In a similar manner, the system can support different persistence mechanisms such as memory, file or database backends. A message dispatcher acting as a *front controller* enables node to node service communication. Every request from the client or other nodes to the container is treated as a message, and is identified and dispatched through the message dispatcher component. The communication mechanism

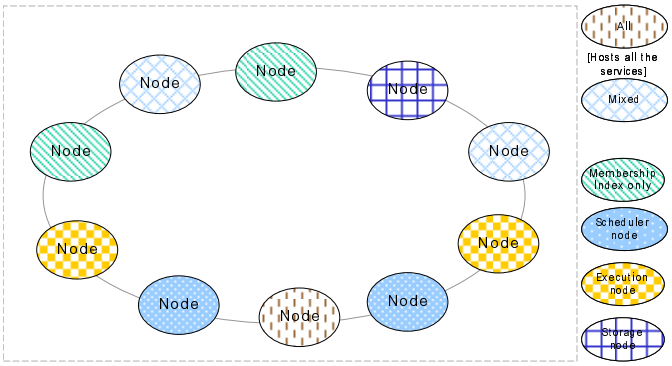

Figure 3. A Sample Aneka configuration

used by the message dispatcher can also be configured to use socket, .NET remoting or web services.

3.3. Services

The services provide the core functionality of the distributed computing environment, while the infrastructural concerns are handled by the runtime framework. This model is similar to a web-server or application-server where the user hosts custom services / modules that run in a managed container. For enabling a distributed computing environment on top of the container, various services such as resource information indexes, execution services, scheduling and resource allocation, and storage services would be necessary. The only mandatory service that a container needs to host is the Membership Catalogue, which maintains network connectivity between the nodes.The services themselves are independent of each other in a container and only interact with other services on the network, or the local node through known interfaces.

Figure 4 shows the interaction between different components within the Aneka environment. A client program first searches, through the index nodes in the Aneka network, for available Aneka nodes where the appropriate scheduling service is deployed. Once a scheduler is discovered, the client program will submit its work along with its credentials. The scheduling service will authenticate the client's request, and discover appropriate executors for executing the client's program, using the index nodes. Once suitable executors are found, the jobs will be dispatched to those nodes and executed. A service on the scheduler node will monitor the executions, collect the results and send them to the client once the executions are completed. The messages exchanged between client, schedulers and executors contains information about the security token, source and destination URLs, the name of the service that actually handles the message, and any required application data. The services will never communicate with each other and exchange the messages between themselves directly, all the messages are dispatched and handled through the MessageDispatcher deployed in each container.

3.4. Node Arrangement

The network architecture is dependent on the interactions among the services, as each Container has the ability to directly communicate with any other Container reachable on the network. Each Aneka node in the network takes on a role depending on the services deployed within its container. For example, a node can be a pure indexing server if only the indexing services (Membership Catalog) are installed in the container; nodes with scheduler services (ThreadScheduler, DataflowScheduler) can be pure scheduler nodes that clients submit their jobs to; nodes with execution services (ThreadExecutor, DataflowExecutor) can be solely concerned with completing the required computation. A node can also host multiple services, and be both a scheduler and executor at the same time. As can be seen in Fig 3, where different types of Aneka nodes are configured to create a network in which each node works as a peer, a request from the end user can potentially spread to every node with the appropriate functions. In this case, as there is no central manager to manage other executors, the request will be filtered by each node which will decide whether to handle or to ignore the request.

4. Implementation of Multiple Application Models on Aneka

Aneka runtime is implemented by leveraging Microsoft .NET platform and using the IoC implementation in the Spring .NET framework [12]. We chose Microsoft .NET on account of its ubiquity on Windows desktops and the potential of running Aneka

on Unix-class operating systems through the .NET-compliant Mono platform[1]. The multiple application models are implemented as extended services on top of the runtime framework. The following sections explain the implementation of two known distributed programming models on top of Aneka, and also how the users configure and deploy an Aneka node.

4.1. Task Farming Model

A task is a single unit of work processed in a node. It is independent from other tasks that may be executed on the same or any other node at the same time. It is also atomic, in the sense that it either executes successfully or fails to produce any meaningful result.

The task model involves the following components: the client, the scheduler and the executor. The task object is serialised and submitted by the client to the scheduler. The task scheduler is implemented as a service hosted in the Aneka container, and continuously listens for messages for requests such as task submission, query, and abort. Once a task submission is received, it is queued in its database. The scheduler thread picks up queued tasks and maps them to available resources based on various parameters including priorities, user QoS requirements, load and so on. These parameters and scheduling policy is pluggable and can be replaced with custom policies. The task scheduler keeps track of the queued and running tasks and information about the perceived performance of the task executor nodes it is able to find in the network, by communicating with the membership service.

The task executor is also implemented as a service hosted in a container, and its main job is to listen for task assignments from the scheduler. When the executor receives a task, it unpacks the task object and its dependencies, creates a separate security context for the task to run, and launched the task. This allows the task to run in a sandboxed application domain separate from the main domain in which the container runs. The executor supports multi-core and multi-CPU scenarios by accepting as many tasks to run in parallel as there are free CPUs / cores. Once a task is complete, it notifies and sends the results back to the scheduler. The executor can accept tasks from any scheduler in the network.

In order to enable the interoperability with custom grid middleware and the creation of a global, cross-platform grid environment, a web services interface that provides the job management and monitoring functionalities has been implemented on top of the task model.

4.2. Dataflow Programming Model

Dataflow programming model abstracts the process of computation as a *dataflow graph* consisting of *vertices* and directed *edges*. The *vertex* embodies two entities: the data created during the computation or the initial input data if it is the first vertex, and the execution module to generate the corresponding vertex data. The directed *edge* connects vertices, which indicates the dependency relationship between vertices.

The dataflow programming model consists of two key components, the scheduler and the worker. The scheduler is responsible for monitoring the status of each worker, dispatching ready tasks to suitable workers and tracking the progress of each job according to the data dependency graph. It is implemented as a set of three key services:

- *Registry service*: maintains the location information for available vertex data. In particular, it maintains a list of indices for each available vertex data.
- *Dataflow Graph service*: maintains the data dependency graph for each job, keeps track of the availability of vertices and explores ready tasks. When it finds ready tasks, it will notify the scheduler component.
- *Scheduling service*: dispatches ready tasks to suitable workers for executing. For each task, the master notifies workers of inputs & initiates the associated execution module to generate the output data.

The worker works in a peer to peer fashion. To cooperate with the scheduler (which acts as the master), each worker has two functions: executing upon requests from master and storing the vertex data. Therefore, the worker is implemented as two services:

- *Executor service*: receives execution requests from the master, fetches input from the storage component, stores output to the storage component and notifies *master* about the availability of the output data for a vertex.
- *Storage service*: is responsible for managing and holding data generated by executors and providing it upon requests. To handle failures, the storage component can keep data persistently locally or replicate some vertices on remote side to improve the reliability and availability.

To improve the scalability of the system, workers transfer vertex data in a P2P manner between themselves. Whenever the executor service receives an executing request from the master node, it sends a fetch request to the local storage service. If there is one local copy for the requested data, the storage service will

fetch the data from remote worker according to the location specified in the executing request. When all the input data is available on the worker node, the executor service creates an instance for the execution module based on the serialized object from the master, initialises it with the input vertices and starts the execution. After the computation finishes, the executor service saves the result vertex into local storage and notify the registry service. The storage service keeps hot vertex data in memory while holding cold data on the disk. The vertex data will be dumped to disk asynchronously to reduce memory space if necessary. The worker schedules the executing and network traffic of multiple tasks as a pipeline to optimize the performance

4.3. Configuration and Deployment

The Aneka container provides a unified environment for configuration and deployment of services. All services are able to use the configuration APIs which store per-user, per-host settings in a simple XML file for each service. This way the settings and preferences for each service are separated from each other, and also allow for customised settings for each user. The deployment of services is a simple operation involving modification of the application configuration file, and adding in entries for the new service to be included in the container's service dictionary.

5. Experimental Evaluation

We have conducted two sets of experiments: the first examined the performance of a single container, and the second evaluated case studies of applications using Aneka's task farming and dataflow programming models to execute over a distributed system.

5.1. Performance Results of Single Container

The Aneka container is the interface to the rest of the distributed system. That is, it sends and receives all messages on behalf of the services hosted within it. In the following experiments, we will evaluate whether this aspect of design has an impact on the performance and scalability of the system. In particular, we will measure the impact of number of services, number of connected clients, and the size and volume of messages on the performance of the container. All the experiments were performed using a single Aneka container running on a PC with an Intel Pentium4 3 GHz CPU, 1 GB of RAM and with Windows XP as the operating system.

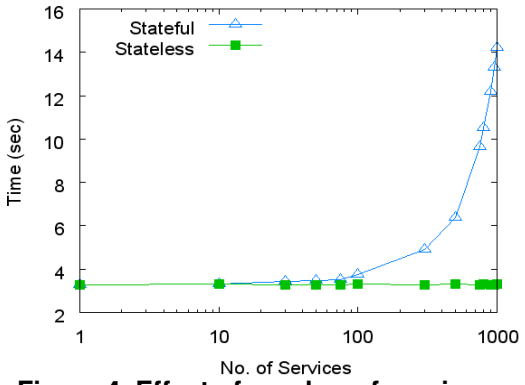

Figure 4. Effect of number of services on startup time

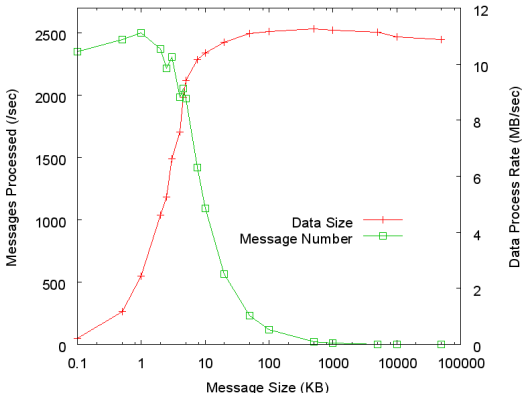

Figure 5. Effect of message size on throughput

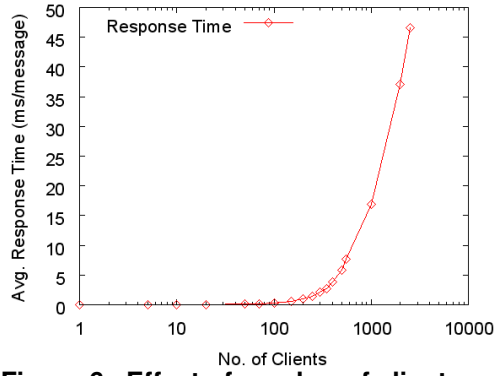

Figure 6. Effect of number of clients on response time

In the first experiment, we measured the variation in startup time of a container with respect to the number of services that are hosted inside it. We evaluated this with two types of services, viz., stateful and stateless. A stateless service is similar to a Web server where the service does not track the state of the client. A stateful service on the other hand tracks requests and connects to the database to store the state of the request. A stateful service also runs in a separate thread. We performed the experiment by starting 1 to 1000

services of each type and measuring the time required for initialising the container.

Figure 4 shows the results of our evaluations. Stateless services do not request any resources and therefore, the time measured here is that required for starting up the container alone. This, as can be seen from the graph, is constant for any number of stateless services. However, the time increases exponentially if the services are stateful. This can, of course, be attributed to the more resource-intensive nature of these services. The curve is uniformly exponential in this case as the same service was started multiple times. However, this may not be true always as different stateful services could affect the startup times in different ways by requiring different amounts of resources. It can also be seen that, in this case, the effects of stateful services become significant only when their number exceeds 300.

As discussed in previous sections, the Aneka container is designed as a lightweight hosting mechanism that provides the bare minimum functionality to the hosted services to create a desktop grid. Figure 3 shows an expected deployment where an Aneka node will offer specific functionality enabled by a small number of specialized services that are likely to be stateful. The above results show that the container does not impact start-up performance in such cases.

The second experiment measures effect of the size and number of messages on the throughput of the Aneka container. The container was initialized with an echo service with a constant time for processing a single message. We then send 10000 messages to the container with sizes varying from 0.1 to 100000 KB and measure the aggregate response time. The results, as shown in Figure 5, are predictable with the message handling rate (number of messages per sec) decreasing uniformly as the size of the message increases. However, the amount of data processes becomes almost constant after a message size of 100 KB. This is because of the configuration of the underlying 100 Mbps network to the container and is not due to the container itself. It can be inferred from the results that Aneka is suitable for embarrassingly parallel applications such as those following master-worker model of computation where the communication occurs only at the end of task execution, and for message-passing applications where the message size is less than 100 KB. However, it may not be suitable for Data Grid applications that require constant access to large amounts of data.

The last experiment determines the response time of the container with respect to number of clients connecting to it. We performed this experiment by keeping the total number of received messages constant at 10,000, but increased the number of threads sending the messages, thereby emulating simultaneous connections from multiple clients. It can be seen from the results shown in Figure 6 that the average response time per message increases steeply when the numbers of clients exceed 400. Even so, the response time per message is within 20 ms for up to 1000 concurrent clients. Currently, every message is synchronised which means it is a blocking call on the container, and therefore performance for large number of clients is affected.

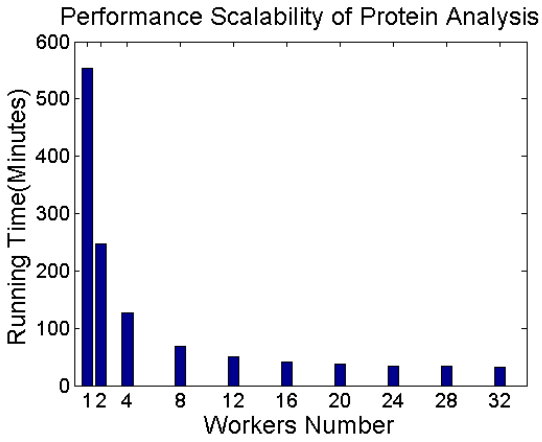

Figure 7. Execution time vs. No. of nodes for Protein Sequence Analysis

5.2. Case Study

We illustrate the versatility of Aneka through case studies involving two distributed applications that were implemented using two different programming models on top of the same infrastructure. The first application predicted the secondary structure of a protein given its sequence, using Support Vector Machines-based classification algorithms[17]. This was implemented using the independent task programming model. The other application performed matrix multiplication and was implemented using the dataflow programming model presented in the previous section. These applications were evaluated on a testbed consisting of 32 PCs located in a student laboratory; each of which were similar to the PC on which the container was tested. These PCs were connected through a 100 Mbps network.

The structure prediction application was executed as a master-worker application across the testbed. Each executor (or worker) node runs an instance of BLAST[18] for each protein sequence, the results of which are then input to a set of classifiers that attempts to predict the secondary structure. The result of this

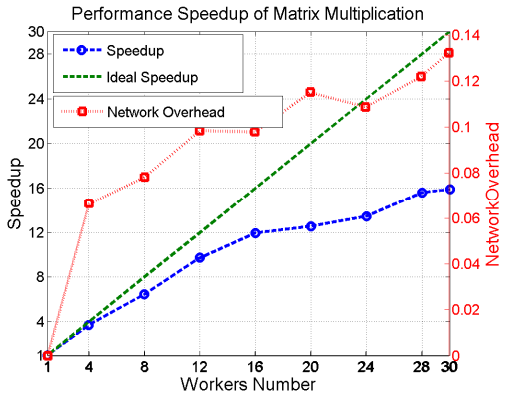

Figure 8. Speedup Gain for Matrix Multiplication

process is returned to the master process. Each instance of the application accessed a 2.8 GB-sized database which, in this case, was replicated across all the nodes. The evaluation was carried out using 64 protein sequences at a time, with varying number of worker nodes. The results of the experiment are shown in Figure 7. The execution time decreases logarithmically until the number of nodes reaches 16 after which there is no more gain in performance to be derived from increased parallelization.

The block-based square matrix multiplication experiment was evaluated with two 8000 x 8000 matrices over a varying number of nodes up to a maximum of 30 nodes. The matrix was partitioned into 256 square blocks where each block was around 977 KB. On the whole, the experiment used 488 MB of data as input and generated a result of size 244 MB. The results of the experiment are shown in Figure 8. There are 2 main factors that determine the execution time of the matrix multiplication: the distribution of blocks between the executors (or workers) and the overhead introduced by the transmission of intermediate results between the executors. The network overhead is measured here as the ratio of the time taken for communication to the time taken for computation. As can be seen from Figure 8, for larger number of executors, while the speedup improves, the network overhead is also substantially increased. The speedup line starts diverging from the ideal when the network overhead increases to more than 10 % of the execution time.

6. Conclusion and Future Work

This paper presented the design, implementation and evaluation of Aneka, a new service-oriented enterprise grid computing framework. Aneka improves over existing desktop grid implementations with fixed capabilities, by using a container in which services can be added to augment the capabilities of the node. We have demonstrated the flexibility of Aneka through case studies using two different programming models executed on top of the same desktop grid. Besides these two models, the threading programming model derived from Alchemi, and a limited subset of MPI, are also supported in Aneka.

From the results, it can be seen that while the container is lightweight in itself, there is scope for improving message handling at the container level and the response time for a large number of clients. The system needs to provide facilities for asynchronous message passing. In the future, we plan to evaluate the design using a larger testbed that spans wide-area networks. We also plan to add peer-to-peer indexing service so that applications can create custom overlays on top of the Aneka infrastructure.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work is partially supported by grants from the Australian Research Council (ARC) and the Australian Department of Education, Science and Training (DEST).

Reference

[1] Mono, http://www.mono-project.com/Main_Page (accessed December 2006).

[2] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D.Werthimer, SETI@home: An Experiment in Public-Resource Computing, *Communications of the ACM*, Vol. 45 No. 11, ACM Press, USA, November 2002.

[3] A. Chien, B. Calder, S. Elbert, K. Bhatia, Entropia: Architecture and Performance of an Enterprise Desktop Grid System, *Journal of Parallel and Distributed Computing*, Volume 63, Issue 5, Academic Press, USA, May 2003.

[4] Intel Corporation, United Devices' Grid MP on Intel Architecture, http://www.ud.com/rescenter/files/wp_intel_ud.pdf (accessed November 2006)

[5] C. Germain, V. Neri, G. Fedak, F. Cappello, XtremWeb: building an experimental platform for Global Computing, *Proc. of the 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, Bangalore, India, Dec. 2000.

[6] A. Luther, R. Buyya, R. Ranjan, S. Venugopal, Alchemi: A .NET-Based Enterprise Grid Computing System, *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, June 27-30, 2005, Las Vegas, USA.

[7] P. Kacsuk, N. Podhorszki, T. Kiss, Scalable desktop Grid system. *Proc. of 7th International meeting on high performance computing for computational science (VECPAR 2006)*, Rio de Janeiro, 2006.

[8] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage, *Proc. of 5th IEEE/ACM International Workshop on Grid Computing*, November 8, 2004, Pittsburgh,USA.

[9] F. Cappello, 3rd Generation Desktop Grids, *Proc. of 1st XtremWeb Users Group Workshop (XW'07)*. Hammamet, Tunisia, 2007.

[10] M. Litzkow, M. Livny, M. Mutka, Condor - A Hunter of Idle Workstations, *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS 88)*, San Jose, CA, IEEE, CS Press, USA, 1988.

[11] R. Ranjan, R. Buyya, A. Harwood, A Model for Cooperative Federation of Distributed Clusters, Poster Paper, *Proc. of the 14th IEEE International Symposium on High Performance Distributed Computing* (HPDC-14), July 2005, Research Triangle Park, N. C., USA, IEEE CS Press, Los Angeles, USA.

[12] Spring.NET, http://www.springframework.net, (accessed November, 2006).

[13] S. M. Larson, C. D. Snow, M. R. Shirts, V. S. Pande, Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology, *Computational Genomics*, Richard Grant (ed.), Horizon Press, 2002.

[14] M. Fowler, Inversion of Control Containers and the Dependency Injection pattern, http://www.martinfowler.com/articles/injection.html, (accessed October, 2006).

[15] climateprediction.net, http://www.climateprediction.net, (accessed November, 2006).

[16] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, S. Tuecke, Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF, *Proceedings of the IEEE*, volume 93, pages 604 – 612, March 2005.

[17] J. Gubbi, M. Palaniswami, D. Lai, M. Parker, A Study on the Effect of Using Physico-Chemical Features in Protein Secondary Structure Prediction, *Applied Artificial Intelligence*, pp. 609-617, World Scientific Press, 2006.

[18] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D. J. Lipman, Basic Local Alignment Search Tool, *Journal of Molecular Biology*, 1990 Oct 5; 215(3):403-10.