CAESAR: Middleware for Complex Service-Oriented Peer-to-Peer Applications

Lipo Chan
University of Melbourne
Victoria 3010, Australia
lipoc@csse.unimelb.edu.au

Shanika Karunasekera
University of Melbourne
Victoria 3010, Australia
shanika@csse.unimelb.edu.au

Aaron Harwood
University of Melbourne
Victoria 3010, Australia
aharwood@csse.unimelb.edu.au

Egemen Tanin
University of Melbourne
Victoria 3010, Australia
egemen@csse.unimelb.edu.au

ABSTRACT

Recent research advances in Peer-to-Peer (P2P) computing have enabled the P2P paradigm to be used for developing complex applications beyond file sharing and data storage. These applications have drawn significant benefits, specifically scalability and low cost, from the P2P paradigm. However, the current approach for designing P2P applications introduce issues that prevent the development of high quality complex P2P applications. These issues, namely tight coupling to P2P protocols, limited logic sharing between peers and complicated recovery processes, motivate us to introduce a service-oriented architecture for P2P applications. We have developed a middleware called **CAESAR** to support the development of service-oriented P2P applications applying the principles of abstraction, dynamic binding, loose coupling and information hiding. In this paper, we discuss the design principles and the components of CAESAR middleware, as well as our experiences in using CAESAR to develop several service-oriented P2P applications.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design

Keywords

Peer-to-peer, service-oriented, middleware

1. INTRODUCTION

Peer-to-Peer (P2P) networks and applications are currently receiving considerable attention. P2P networks are distributed, decentralized and scalable systems comprising of a very large number of computers working together. The P2P approach eliminates the dependency on one or more central resources, thus offers great potential for developing scalable distributed applications at low cost. Some examples of these widely used applications are telephony services using Voice Over IP, such as Skype, and Massively Multiplayer Online Games.

Current use of P2P technology has not provided for rich experiences due to the limitations of design approach used for P2P applications. One of the main limitations is the tight coupling of the application to the underlying P2P protocol. This makes the task of developing P2P applications a tedious activity requiring detailed knowledge about the underlying P2P protocol (much like knowing the details about TCP protocol in order to use sockets). In addition, such tight coupling also constrains an application from choosing different protocols at runtime to meet various requirements, specifically the quality of service needed by the users. Besides tight coupling, existing P2P applications are usually limited to sharing data storage. These applications have minimal ability to delegate processing to other peers in the network, thus not able to realize the full extent of the computing power provided by the underlying P2P network. Furthermore, complicated development process arising from integrating with complex P2P protocols hinders current P2P applications from giving due consideration to important software quality attributes such as reliability, security, interoperability and performance.

The limitations in the development of P2P applications create a challenge for the wide acceptance of P2P paradigm when developing complex applications. Nevertheless, as we present in this paper, using a service-oriented approach, the P2P paradigm is a fitting solution for the development of practical complex applications. In this paper, we present a middleware called *CAESAR* that provides support for developing complex P2P applications through a service-oriented approach. We adopt a component-based framework in our design, which promotes reuse and extensibility. CAESAR offers the following benefits:

- Simplifying complex P2P application development via *abstraction*;
- Sharing logic within the P2P network through a *service-oriented approach*;
- Enhancing peer functionality with *dynamic binding*;

- Developing robust P2P applications with appropriate *network information hiding*;

CAESAR is developed with Key Based Routing (KBR) P2P protocols in mind. Therefore, throughout this paper, the P2P context refers to the KBR-based approach. The remaining structure of this paper is: Section 2 describes our design principles, Section 3 introduces the CAESAR middleware, Section 4 lays out the implementation details, Section 5 presents applications developed using the CAESAR middleware, Section 6 discusses related works, and Section 7 concludes the paper.

2. DESIGN PRINCIPLES

There are significant software engineering issues confronting P2P application developers that are not properly addressed by current specialized P2P applications (i.e. file sharing), and for which existing distributed systems solutions are inadequate. In this section, we outline the design goals of CAESAR, which provides a basis for building complex P2P applications using a service-oriented approach.

2.1 Simplifying Application Development

The most challenging part of developing P2P applications is the need to understand underlying P2P protocols. Although these protocols are developed with various schemes, the underlying goal remains the same - to provide consistent and efficient lookup of data within a P2P network. These protocols use a variety of data structures and complex algorithms to achieve the underlying goal. In some cases, additional non-functional requirements, such as reliability and anonymity, are also supported by the protocols. Consequently, the complexity of the protocols increases and they become even harder to comprehend and to develop.

Application developers should be given the option of building their P2P applications without the stress of learning the complex P2P protocols. As such, we argue that a solution in simplifying the development of P2P applications is *abstraction.* The details of the protocols or the network should be abstracted into meaningful and easy to understand interfaces. This form of abstraction also resolves the issue of tight coupling, and leads to desirable quality goals, such as interoperability and extensibility.

2.2 Sharing Logic within P2P Networks

Current of use of P2P networks, specifically self-organizing networks, is still fairly limited, which means most applications are only utilizing the network for data storage. This limited capability is acceptable for simple applications, such as file sharing, but fast becoming inadequate when developing complex applications. Complex applications can be computationally intensive, thus requiring more than just sharing of storage, but logic as well.

To achieve this goal, we propose the use of a service-oriented approach for peers to make selective interactions. The Service-Oriented Architecture (SOA) [6] makes a good candidate with a paradigm that relies on the composition of services to provide functionalities for software development. We adopt a modified SOA approach, where peers offering the same types of services are able to form a community amongst themselves within the P2P network. Subsequently, these peers are able to increase their efficiencies by delegating processing to other peers within the same community that have similar capability. We term this community of peers as *service overlay.*

2.3 Enhancing Peer Functionality

In the current development process, application developers have to design their applications with specialized strategies to meet the requirements, and in most cases, it entails selecting a particular P2P protocol to be used. Due to the unavailability of common interfaces, the decision to use a particular protocol can result in the application being tightly coupled to the protocol. However, we know that applications evolve over time, with the need to support more functionalities to resolve different problem areas. Accordingly, an application may wish to use a different protocol that better serves its requirements. In such cases, tight coupling to a particular protocol can result in an enormous amount of rework.

Our proposal of using services in the previous section means the requirements of applications are met through services. Services can have different quality attributes, and typically a service builds these quality attributes using various control strategies working together with different P2P protocols. Our *dynamic binding* solution, which is made possible due to the protocol abstraction, allows services to dynamically bind themselves to different protocols, thus developing services with different quality attributes. Over time, as more protocols and services become available, the functionality supported by the peer will also increase. Dynamic binding is achieved using well-defined interfaces that promote loose coupling.

2.4 Developing Robust P2P Applications

One of the main advantages of the P2P paradigm is the self-organizing behaviour provided by P2P protocols. In the event of one or more peers leaving the network, the protocols ensure that routing is minimally impacted. Nevertheless, application developers still have to explicitly manage the consistency and the integrity of application-level data through appropriate replication and migration mechanisms. These mechanisms are usually non-trivial, thus imposing extra burden on application developers. Although several P2P protocols offer variations of these mechanisms, they are closely tied to the protocols, which is not ideal.

To relieve the application developers from these complicated tasks, we propose a form of network information hiding using an *embedded network management* approach. Network management processes, such as migration and replication, are automatically managed on behalf of the applications.

3. CAESAR MIDDLEWARE OVERVIEW

Fig. 1 shows the components of the CAESAR middleware. The middleware consists of five core and two non-core components. The core components are *Protocol Facade (PF), Service Facade (SF), Network and Object Management (NOM), Application Director (AD) and Internet Simulation & Emulation (ISE)*, and the non-core components are *Service Plugin and Protocol Plugin.* These components interact with each other via a low coupling approach to achieve the design principles as presented in Section 2.

CAESAR supports three abstraction levels of development, namely application, service and protocol. We introduce three access components, i.e. AD, SF and PF. The AD hides all the complex P2P information, and is meant

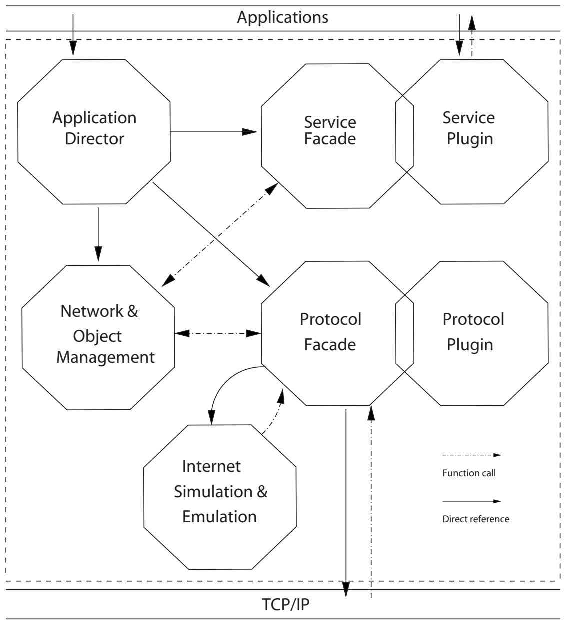

Figure 1: Components of CAESAR

to be used by the application developers to access functions offered by the CAESAR middleware. Similarly, the SF provides access to the middleware for service developers. The SF uses a plugin-based approach to allow service developers to add plugins to the CAESAR middleware. The PF has similar function as the SF, but is meant for protocol developers. The PF encapsulates complex network details (i.e. TCP/UDP), thus reducing the need for protocol developers to get down to such elaborate matters. These three components essentially support the right level of abstraction to facilitate independent application, service or protocol development, thus achieving our first design principle.

The second design principle, logic delegation, is achieved through a service-oriented approach as previously described. The NOM component provides the functionality to manage the service-oriented overlays, which are transparent to applications, services and protocols. The NOM works with the SF and the PF to allow service plugins to dynamically bind themselves to different P2P protocols available in the middleware. The service plugins are then able to make use of the various characteristics of the P2P protocols to set up different service overlays. A service overlay, as pointed out earlier, is basically a subset of the peers in the network that have a particular service plugin, and able to offer a service with the same quality attributes. This is in essence a sub-network within a large P2P community network. A service plugin can be part of multiple service overlays, if it has the ability to offer services with different quality attributes. The NOM manages the association of the service plugins to these service overlays via namespaces. Subsequent interactions amongst peers in the network are only done through service overlays, which allow delegation of tasks that are beyond the typical data storage capability. The NOM is essentially a powerful connector that allows many services with different quality attributes to be produced, which enhance the functionality of a peer.

To assist the service plugins in providing more reliable services, CAESAR supports a separation of data from service plugins using the NOM. The separation allows the service plugins to be stateless, and enables the NOM to silently perform important functions, such as garbage collection, replication or migration, in order to ensure network maintainability and possible recovery in the face of failure. The data structure used in the separation is called *ServiceObject (SO)*. Each SO has its own unique identifier, and contains data information or processing states associated with a service plugin. The NOM manages all the SOs, and each service plugin can have multiple SOs, depending on its own operations.

The final core component of CAESAR is the ISE testbed. The ISE facilitates simulations by allowing applications to execute in a simulation mode, much like a real-time network. This allows a large number of peers to be simulated using substantially less hardware resources than that it would take in a non-simulation environment.

In the remaining part of this section, we describe the details of all components and its API functions, except the ISE component, which is omitted due to space limitations.

3.1 Application Director

The AD is a thin control layer that instantiates all the core components of the framework via the *initialize* functions of the components. The API functions supported by AD are shown as follows:

- $\mathcal{F}$: $getService(serviceName, reqs)$
  returns a reference to the service plugin with *serviceName* and the given requirements, *reqs*.
- $\mathcal{F}$: $stopService(serviceName)$
  stops the operation of the service plugin associated with *serviceName*.
- $\mathcal{F}$: $startUp(simFlag)$
  starts up the peer.
- $\mathcal{F}$: $shutDown()$
  shuts down the peer.

getService has correlation with *get* in the SF, while *stopService* is related to *stop* in the SF. An application can start up a peer by invoking *startUp*. The parameter *simFlag* allows the application developer to decide whether to start the application in a simulation mode or in a deployment mode. *startUp* initializes all the core components, and passes function objects between the components via the individual *initialize* function. All parameters marked with (*) in the *initialize* functions are function objects, with equivalent functions being defined in one of the core components. Finally, when an application developer wishes to stop his/her application, *shutDown* invokes all the $finish$ functions in the core components.

3.2 Service Facade

The SF provides access to the service plugins through a set of simple API functions as outlined in the following list:

- $\mathcal{F}$: $get(serviceName, reqs)$
  returns a reference to the service plugin with *serviceName* and the given requirements, *reqs*.
- $\mathcal{F}$: $stop(serviceName)$
  stops the operation of the service plugin with *serviceName*.
- $\mathcal{F}$: $handleMsg(serviceName, message)$
  handles *message* intended for the service plugin with *serviceName*.
- $\mathcal{F}$: $createObj(serviceName, soId)$
  requests a service plugin with *serviceName* to create a new SO with identifier *soId*.

$\mathcal{F}$: $initialize(*regService, *send, *terminate, *getObj, *delObj)$
initializes the SF with function objects from other core components.

$\mathcal{F}$: $finish()$
stops all operations.

The SF provides access to service plugins via *get*. On invocation, a reference to the specified service plugin is returned, and application developers can subsequently use the functions provided by the service plugin. The parameter *reqs* indicates a selection of the requirements offered by the service plugin with $serviceName$. An application uses the *reqs* to get the different levels of service provided by a service plugin. Interactions via service overlays involve messages, and these messages are passed to the service plugins through $handleMsg$. When an application no longer requires the use of a service plugin, *stop* is called, where tasks such as garbage collection can occur. As previously mentioned, service plugins keep their processing state and data in the NOM via SOs. *createObj* allows the NOM to request each service plugin to create their specific SOs to be stored. *createObj* is further explained in Section 3.6.

3.3 Service Plugin

The Service Plugin is not a core component of CAESAR, but used in conjunction with the SF. Each service plugin has to conform to the API functions described as follows:

$\mathcal{F}$: $start(*send, *getObj, *delObj, *get)$
starts offering services.

$\mathcal{F}$: $stop(*terminate)$
stops offering services.

$\mathcal{F}$: $handleMsg(message)$
handles a message.

$\mathcal{F}$: $createObj(soId)$
creates a new SO with identifier $soId$.

$\mathcal{F}$: $name()$
returns the name of the service plugin.

The SF initializes a service plugin via *start*, and terminates a service plugin via *stop*. The parameters in these functions are passed by the SF. *handleMsg* is called by the SF when a message intended for a service plugin arrives at the peer. *createObj* is called by the SF to allow the individual service plugin to create SOs that contain specific plugin information, such as processing states and data. *createObj* is invoked by the equivalent function in the SF. Finally, *name* allows the SF to access the name of a service plugin. Additionally, each service plugin can support any number of service specific functions that can be offered to the applications. Each service plugin can also access another service plugin via *get* from the SF. By allowing direct interaction between service plugins and applications, CAESAR provides the right amount of control with high degree of flexibility.

3.4 Protocol Facade

In general, P2P protocols provide a peer with three crucial functionalities; to search and join a P2P network, to gracefully leave a P2P network, and to interact with other peers via message routing. The PF manages a collection of P2P protocol plugins, and provides access to these plugins via a set of common interfaces, described as follows:

$\mathcal{F}$: $join(protocolName, namespace)$
requests a protocol plugin with $protocolName$ to join the specified P2P service overlay with the given $namespace$.

$\mathcal{F}$: $leave(namespace)$
leaves the specified P2P service overlay with the given $namespace$.

$\mathcal{F}$: $route(namespace, id, message)$
routes $message$ to a peer that is responsible for identifier id within the service overlay with the given $namespace$.

$\mathcal{F}$: $filter(namespace, soId)$
migration function that determines whether a SO associated with $soId$ is to be migrated to another peer.

$\mathcal{F}$: $transport(address, message, tcp/udp)$
an actual network connection, where $message$ being sent to another peer with the given $address$ using tcp/udp.

$\mathcal{F}$: $receive(namespace, message)$
receives $message$ intended for the peer over the $namespace$.

$\mathcal{F}$: $initialize(*regProtocol, *accept, simFlag)$
initializes the PF with function objects from other core components.

$\mathcal{F}$: $finish()$
stops all operations.

CAESAR allows a peer to join multiple service overlays that could be running on different P2P protocols. For each of the overlays to join, the peer initializes an instance of the associated P2P protocol via *join*, where it connects using a unique *namespace*. After connecting to this overlay, the peer interacts with other peers in the overlay by passing message via *route* and *transport*. *route* uses a unique combination of the namespace and an identifier to allow the associated protocol plugin to determine the destination peer, and *transport* performs the actual network connection. The peer receives a message through *receive* and passes it to the NOM. The interaction via messages continues until the peer decides to leave the overlay by invoking *leave*. This graceful exit maintains the consistency of the overlay by allowing action such as data migration to other peers in the overlay. Data migration is supported via $filter$, which is further described in Section 3.6.

3.5 Protocol Plugin

The Protocol Plugin is also not a direct component of CAESAR, and used in conjunction with the PF. Each protocol developer can offer his/her P2P protocol by building a plugin that conforms to the API functions described in the following list:

$\mathcal{F}$: $join(namespace, host)$
joins the specified service overlay with the given $namespace$ via a known $host$.

$\mathcal{F}$: $leave(namespace)$
leaves the specified service overlay with the given $namespace$.

$\mathcal{F}$: $route(namespace, id, message, *transport)$
routes $message$ to a peer that is responsible for identifier id within the service overlay with $namespace$.

$\mathcal{F}$: $filter(namespace, soId)$
migration function that determines whether a SO associated with $soId$ is to be migrated to another peer.

$\mathcal{F}$: $name()$
returns the name of the protocol plugin.

The PF instantiates an instance of a protocol plugin for every service overlay (note that the service overlay concept is transparent and does not affect the implementation of a protocol plugin). The PF calls *join* to connect the peer to a service overlay, invokes *leave* to leave a service overlay, and uses *route* to send a message over a service overlay. When a message belonging to a *namespace* arrives at the peer,

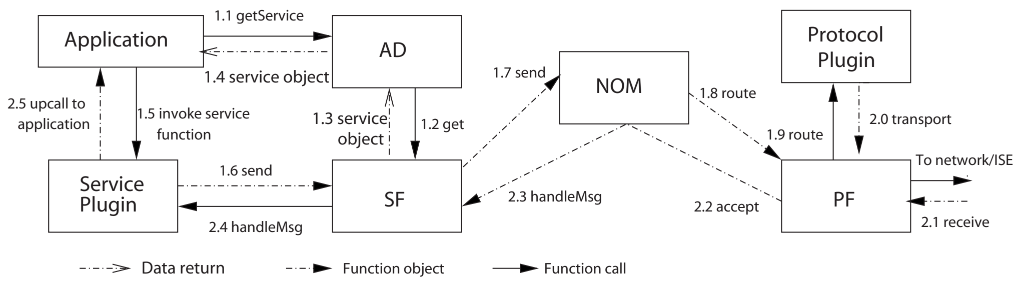

Figure 2: Collaboration between core components of CAESAR

the PF checks for peer responsibility by calling $filter$ of the protocol plugin instance managing the *namespace*. Finally, each protocol plugin is required to return its name via *name*.

3.6 Network and Object Management

The NOM provides various functions to support dynamic binding, data storage, data migration and data replication, as well as garbage collection. The NOM receives directives from the service plugins via the SF, and works with the PF to serve the directives. These directives contain information, such as the name of a protocol, replication and access control strategies, that guide the NOM in invoking the appropriate procedures. The following list presents the API functions of the NOM:

- F: $send(serviceName, reqs, message)$
  sends $message$ to another peer running service plugin with *serviceName*.
- F: $accept(namespace, message)$
  accepts and passes $message$ to a particular service plugin.
- F: $terminate(serviceName)$
  terminates the operation of a service plugin with $serviceName$.
- F: $getObj(serviceName, soId)$
  returns the SO with $soId$ associated with service plugin with *serviceName*.
- F: $delObj(serviceName, soId)$
  deletes the SO with $soId$ associated with service plugin *serviceName*.
- F: $regProtocol(protocolNames)$
  registers the names of the available protocol plugins.
- F: $regService(< serviceNames, reqs >)$
  registers the names of the available service plugins and their associative requirements.
- F: $initialize(*join, *leave, *route, *filter, *createObj, *handleMsg)$
  initializes the NOM with function objects from other core components.
- F: $finish()$
  stops all operations.

At startup, a peer registers all the service and protocol plugins currently available in the framework via *regProtocol* and *regService*. After startup, *send* is called by the SF when a service plugin wishes to send a message to a corresponding peer in the same service overlay. The parameter *serviceName* is a unique name that identifies a particular service plugin. The NOM uses the *serviceName* and the additional requirements *reqs* to form a namespace. When a message arrives from another peer, the PF calls *accept*, and the NOM passes the message to the intended service plugin (the NOM is able to map the message back to a service plugin based on the *namespace*). If a service plugin requires access to a SO, and the SO is not found when calling *getObj*, the NOM probes the corresponding service plugin by calling *createObj* (obtained during initialization from the SF). The service plugin then creates a new instance of a SO with the given *soId*, and returns the SO to the NOM. The SO is stored by the NOM, and a reference is returned to the earlier call of *getObj*. Finally, when a peer wishes to stop the operation of a service plugin, *terminate* is called, where tasks such as garbage collection can occur.

Migration processes are triggered when a peer joins or leaves a network. The NOM of a joining peer sends a special message to its neighbouring or successor peers. Upon receiving this message, the NOMs of the neighbouring or successor peers invoke an internal migrate function (not shown in API), and check for the SOs to be migrated via *filter* (defined in Section 3.4) given to the NOM at initialization. Similarly, a leaving peer invokes its own migrate function to pass all relevant SOs to its neighbouring or successor peers. Note that a SO is only ever migrated to another peer within the same service overlay. Migration is an important process in the P2P networks, as peers come and go fairly frequently, and efficiency is crucial to ensure consistency. In contrast, replication processes are triggered by the requirement of the service offered by a service plugin. Replication assists service plugins in providing reliable and consistent services. Based on the specified requirements (via *get* in Section 3.2), the NOM uses an internal replication function (also not shown in API) to check for the most recently changed SO and applies a naming scheme to create a replica or replicas. The number of replicas is determined by the requirements.

4. MIDDLEWARE IMPLEMENTATION

We have implemented a prototype of CAESAR using the C++ language. This middleware library contains the core components described in Section 3. The current library supports the addition of plugins through manifest files, which contain information such as plugin name, plugin type (service or protocol), description of the plugin and additional quality attributes or requirements to be offered to applications (these can be pre-defined control or replication strategies). The library also provides a few core service and default protocol plugins. These core service plugins, namely Global Management, Data Management, Virtual Machine and Access Management, either cover a large variety of complex applications or are essential to include for most applications. The default protocol plugins, namely Chord [7] and FLOC [4], allow immediate use for application development. More details of the C++ library and the available plugins can be found in [1].

To give better understanding of how the core components interact to serve an application, we illustrate the typical use of the library using a collaboration diagram, shown in Fig. 2. The sequence of calls show when a peer application requests for a reference to a service plugin, and subsequently invokes a function (specific function associated with the service plugin) through the reference, which results in a message being sent to another peer application within the same service overlay. The diagram also depicts the functions invoked when a message associated with a service overlay arrives at a peer, which ultimately reaches the application. Note that CAESAR middleware gives control to the service plugins to handle the upcalls to the applications.

5. APPLICATIONS

CAESAR has been used to develop several complex P2P applications. One of the complex and exciting applications is the *P2P-based Massively Multiplayer Online Games (MMOG)* [3], which is currently being commercialized. This P2P MMOG application is a set of interacting entities in a virtual world with a large population of players. The P2P MMOG prototype uses two services namely Entity Interaction Service (EIS), Spatial Data Service (SDS) and one protocol, Entity Interaction Protocol (EIP). These components have been developed as independent service and protocol plugins to CAESAR. Fig. 3 shows a screenshot of the P2P MMOG.

Figure 3: A screen shot of P2P MMOG

CAESAR has also been used to develop several other complex service-oriented P2P applications: P2P Event Finder that facilitates look up service for events occuring in a city, MPICH-OPeN [5] that provides a platform for running Message Passing Interface (MPI) programs over P2P networks, and Intrusion Detection Service [8] that provides collaborative approach to detect network intrusions.

6. RELATED WORK

JXTA (*www.jxta.org*) is a P2P application development framework that supports an overlay using JXTA protocol. JXTA does not provide an abstraction for a general DHT-based protocol to be used in place of JXTA protocols, thus limiting the flexibility of application in using various protocols. In general, the design goals of JXTA, namely interoperability, platform independence and ubiquity are quite different from the design goals of CAESAR.

CAESAR is both different and complementary to a research effort known as OpenDHT (*www.opendht.org*), which offers a DHT-based routing overlay to applications via a public service deployment (using Bamboo as the underlying DHT protocol). The work of Dabek et. al. [2] is directed towards generalizing and refining the interfaces between applications and protocols. This work is intended to support routing algorithms and is not designed to have general API functions for development of complex P2P applications. In summary, current development infrastructure does not provide an adequate level of abstraction and information hiding that is essential for successful development of complex applications over large scale and dynamic networks; hence the motivation for our work.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented the CAESAR middleware that addresses the challenges of complex P2P application development. Our low coupling component-based framework uses abstraction to simplify application development by decoupling P2P protocols from the applications, and service-oriented approach to allow delegation of processing to other peers in the same service overlay within a P2P network. Using a plugin-based approach, the middleware allows new service and protocol plugins to be easily added, to enhance the functionality of a peer.

In conclusion, our solution is a viable and much needed development support for building complex P2P applications, which we have demonstrated through a few complex application prototypes. Our future agenda includes developing additional plugins, and to investigate a more sophisticated service discovery approach for the framework.

8. ACKNOWLEDGMENTS

Funded in part by the Australian Research Council, grant number DP0451936 and National ICT Australia.

9. REFERENCES

[1] Prototype of CAESAR. http://p2p.cs.mu.oz.au/software/.

[2] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *2nd Int. Workshop IPTPS*, 2003.

[3] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera. Enabling Massively Multi-Player Online Gaming Applications on a P2P Architecture. In *Int. Conference on Information and Automation*, 2005.

[4] A. Harwood, E. Tanin, and M. Truong. Fast Learning of Optimal Connections in a Peer-to-Peer Network. In *IEEE Int. Conference on Networks*, November 2004.

[5] L. Ni and A. Harwood. MPICH-OPeN on the PlanetLab Infrastructure. In *Demonstration, Pacific Rim Applications and Grid Middleware Assembly*, 2006.

[6] M. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. In *Commun. ACM*, Oct. 2003.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM*, 2001.

[8] C. Zhou, S. Karunasekera, and C. Leckie. A Peer-to-Peer Collaborative Intrusion Detection System. In *ICON*, 2005.