TMACS: Type-based Distributed Middleware for Mobile Ad-hoc Networks

Jinsong Lin Eusden Shing Wing-Kai Chan Rajive Bagrodia

{*jinsong, eusden, kai, rajive* }*@cs.ucla.edu*

Mobile Systems Lab, University of California, Los Angeles

Abstract

This paper presents the design and implementation of TMACS – a distributed middleware framework for Mobile Ad-hoc Network (MANETs). TMACS leverages type-based group communication paradigm in which type is used as a first-class abstraction for identifying groups and provides a novel group-based RPC-mechanism called TRPC as a higher-level communication abstraction suitable for MANET computing environments. A fully decentralized discovery service has been provided to lookup the meta-information of the distributed objects and services. At the network layer, TMACS implements TypeCast routing protocol to efficiently support TRPC and service discovery via effective type dissemination and aggregation mechanisms. A complete system implementation of TMACS has been deployed on linux-based mobile devices and has been used to program a variety of applications. We present results from a selected set of applications and services that include an ad-hoc distributed caching service and an ad-hoc marketplace application. The physical implementations were used to evaluate the performance of TMACS and demonstrate its resiliency in the presence of mobility-induced topology changes.

Category and Subject Descriptor
D.2.11 [Software Engineering]: Software Architectures -- *patterns*

Generate Terms
Design

Keywords
MANETs, Middleware, TRPC, Service Discovery, TypeCast

1. Introduction

Mobile Ad-hoc Networks (MANET) are an emerging platform to deploy diverse distributed applications, such as emergency response, battlefield communications and ad-hoc conferences. There has been tremendous progress in MANET-related network research in recent years; hardware support for MANET is also readily available. However, there have not been comparable advances in the availability of programming or application development platforms for MANET environments. Such environments present a number of unique challenges as discussed next:

First, unlike wired Internet or access point based wireless networks, a MANET is a pure peer-to-peer network in which all nodes may move frequently. There is no network "core" to provide relatively stable routing topology, network availability and bandwidth. Such a network architecture precludes DNS-like infrastructure which relies on dedicated, always-on and centrally managed hosts. Without such infrastructures, the World Wide Web would arguably not have become the ubiquitous phenomena that it is today. Such infrastructure services are necessary to promote the use of MANET by application developers. The challenge here is to provide such infrastructure services in a fully distributed fashion where they are resilient to node failure, mobility and network partition without incurring high system and network overhead.

Second, the fact that users voluntarily join a MANET requires them to have a strong incentive to participate, which usually stems from common interests and goals. For example, in an emergency response scenario, police officers, firefighters, and paramedics do not necessarily know each other's identity or network address; nevertheless, they must coordinate with one another in order to fulfill the rescue mission. Such coordination is most likely group oriented, where groups are classified based on the participants' organizational duties and roles. Providing system support for group-based communication in MANET is as important as providing such support for point-to-point communication. In summary, we believe that a middleware framework for MANET must address the following research challenges:

- What are the appropriate high level abstractions for representing group communications that reflect the real world group relationships among MANET users?
- How to provide an expressive yet efficient distributed programming model that can be used to facilitate the development of group-oriented as well as point-to-point based ad-hoc applications in MANET? The programming abstraction should make mobility and network details transparent from applications developers; at the same time, must not cause high performance penalty and system overhead in the face of high mobility and network dynamism.
- How to support a lookup service for mobile nodes to locate available group and individual service information in the network without requiring a centralized naming or directory servers? The discovery service should provide

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512

rich taxonomy to describe the intended targets while minimizing the consumption of network bandwidth.

In this paper, we present the design and implementation of TMACS (Type-based Middleware for Ad-hoc Communication Systems) – an object-oriented distributed platform intended to address the above challenges. TMACS introduces three novel features: an efficient implementation of network routing protocol based on hierarchical group addressing scheme, a group-oriented distributed computing model called TRPC to simplify the ad-hoc application development, and a decentralized, mobility-resilient Discovery Service. The combination of these functionalities forms a light-weight minimalistic MANET middleware upon which more sophisticated system services, programming abstractions and applications can be built.

In TMACS, MANET applications are modeled as distributed objects; the type of an object is used as the key abstraction for classifying and grouping objects and as the first level construct for sending messages. The power of type-based group abstraction lies in its ability to form hierarchical group relationship via the principles of subtyping and type composition[16] to model complex relationships among ad hoc users. Applications can dynamically change the scope of a specific communication by targeting individual messages to different levels in the type hierarchy. New groups can be easily introduced by plugging new types into the existing hierarchy.

With type as the fundamental group abstraction, TMACS introduces a novel distributed computing model called type-based RPC (TRPC) which simplifies the development of group-oriented ad-hoc applications. TRPC extends the traditional point-to-point RPC[6][29] model by allowing an application to simultaneously invoke a remote method call to a group of distributed objects. The group is identified by the combination of type and *Scope* – an abstraction of non-type properties. The set of Java API and the associated language runtime environments makes developing group-oriented ad-hoc applications as easy as writing distributed applications for wired networks. An important deviation from traditional RPC is that all the method calls of TRPC are non-blocking – the return values is retrieved asynchronously via the abstraction of *Future*[30]. This avoids the potential performance penalty of a synchronous method invocation.

To efficiently support TRPC, TMACS utilizes TypeCast routing protocol[16] - a network protocol based on hierarchical type information. The protocol compresses the type hierarchy into Bloom filters[5], which are disseminated and aggregated in the network to facilitate routing. The memory and bandwidth cost for managing routing table is not increased with the number of types in MANET. Our earlier work on TypeCast routing[16] focused on the design and the simulation study of the protocol. This paper significantly expands on the previous work by providing a complete implementation of the TypeCast routing protocol, integrating the protocol into TRPC's language runtime environment, and using the implementation to study system performance on a physical testbed and to validate the previously reported simulation results.

TMACS also provides a discovery service to look up meta information associated with distributed objects in the network. TMACS' discovery service does not require centralized naming servers; rather the service is fully decentralized and is capable of coping with node mobility and network partitions.

To demonstrate the utility & expressiveness of TMACS, we describe the deployment of a novel service: an ad-hoc distributed caching service, and a representative application: an ad-hoc marketplace. The ad-hoc caching service allows any node in a MANET to serve as a cache for other nodes. A cache server can replicate arbitrary types of distributed objects with the cached copies providing the same types of services as the original sources. Such an ad-hoc caching service provides an effective mechanism to increase service availability by relieving hotspot, overcoming network partitions, and node failures.

The marketplace application is considered to be a generic communication pattern for ad-hoc networks[10]. Our implementation of ad-hoc marketplace allows buyers and sellers to trade items via auction style communications in a MANET. The auction is executed completely in peer-to-peer style without a centralized broker. Our implementation experience has shown that TMACS can significantly reduce the complexity of developing group-oriented ad-hoc applications by eliminating the effort for managing group membership and designing low level handshake protocols and messaging mechanisms.

The organization of the rest of the paper is as follows: Section 2 presents the architecture of TMACS. Section 3 discusses the TypeCast routing implementation. The TRPC programming model and TMAC's discovery service is discussed in section 4 and 5. Section 6 demonstrates the ad-hoc caching service and auction application built on top of TMACS. Section 7 presents representative experiment results from a prototype implementation. In section 8, we compare our work with previous research. Section 9 is the conclusion.

2. System Architecture

The architecture of TMACS is entirely based on the peer-to-peer model where all mobile nodes have identical copies of the software stack and they coordinate with each other to provide a distributed infrastructure service. The software stack needed to support the TMACS functionality on each node is shown in Figure 1.

At the bottom level is the TypeCast routing component, which implements a language independent routing protocol supporting efficient type dissemination and type-based packet forwarding. The TypeCast routing component consists of a user space routing daemon and a Linux kernel module.

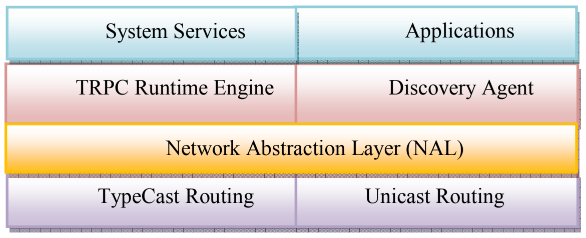


Figure 1: TMACS Stack per Mobile Node

Above the typecast routing component is the Network Abstraction Layer (NAL) which is used to hide details of routing protocols from the core middleware functions. NAL dynamically selects the routing protocols used for forwarding outgoing packets by analyzing the intended targets. It is extensible for plugging new protocol implementation.

The core of TMACS consists of the TRPC Runtime Engine and the Discovery Agent. TRPC is an extension of the traditional RPC model that supports *concurrent* execution of remote methods on one or more distributed objects and the asynchronous retrieval of the results from the corresponding invocation(s). The TRPC Runtime Engine manages the lifecycle of the distributed objects activated on the local node, and is responsible for dispatching remote method invocations and correlating responses. An important function of the Runtime Engine is to aggregate the type information on the local node and to notify the TypeCast routing daemon when any change is detected. The type information will be promptly propagated through the network.

The Discovery Agent coordinates with its counterparts on other nodes to look up the meta information of TRPC service objects in a MANET via a decentralized light-weight discovery protocol. The combination of the TRPC Runtime Engine and the Discovery Agent serves as a distributed framework for the ad-hoc communication systems.

We summarize the characteristics of the TMACS' architecture that make it suitable for MANETs:

- First the TMACS architecture does not include any centralized components. Every node plays an equal role – when a node fails, its functions can immediately be replaced by any other node without disrupting the functioning of the whole system.
- The service discovery is supported via a network of Discovery Agents, it does not require any centralized naming systems.
- The network topology is constructed in a self-organized fashion via the TypeCast routing protocol. The link breakage due to mobility and node failure is automatically detected and repaired to ensure network connectivity. Note that it is possible to implement TMACS using other ad hoc unicast and/or multicast routing protocols; however, TypeCast provides better efficiency that is adaptive to the number of active objects distributed along type hierarchies.
- Last, the combination of the preceding layers provides a light-weight framework that can be used to construct additional services, on an as needed basis, so as to provide a small memory & resource footprint that can be ported to diverse mobile platforms.

In the following sections, we will present in-depth discussion of the design and implementation of each component.

3. TypeCast Routing Implementation

3.1. Protocol Overview

To support type-based communication in MANET, we have previously proposed the TypeCast routing protocol[16] to efficiently route data packets to a group of members identified by their types. For completeness, we provide a summary here.

The basic idea of TypeCast is to leverage any MANET multicast protocol to construct the basic routing topology, while adding a light-weight packet filtering layer to filter the messages based on the type of the recipient objects. To achieve space and bandwidth efficiency, TypeCast compresses type information as a Bloom filter[5] without destroying the structural information of the type hierarchy. The Bloom filters will be disseminated and aggregated within the network to populate the routing table on each node. The TypeCast routing table contains the following entries:

- ***LBF*** (*Local Bloom Filter*): the Bloom filter of all the aggregated types on the local node
- ***FBFs*** (*Forwarding Bloom Filters*): the Bloom filter of all the aggregated types that can be reached via a specific neighbor.

All data packets is tagged by a target Bloom filter (*TBT*), which is the Bloom filter of the target type T. The forwarding decision is based on whether there is a *subfilter* relationship between *TBT* and the *FBFs* in the routing table.

As discussed in[16], the TypeCast routing protocol has the following characteristics:

1. It supports the principles of subtyping and type-composition: A packet targeted to type T will be delivered to all objects of this type, and any of its subtypes. A packet targeted to the composition of two types A and B will be delivered to all objects that have both types A and B.
2. As a Bloom filter is used to aggregate the type information, the memory and bandwidth cost of managing the type routing tables does not increase as a function of the number of types in a MANET. Nevertheless, the false positive ratio will become larger when the number of types increases in the network. Technique such as compression and Explicit False Notification [16] can be used to mitigate the problem.
3. By utilizing the MANET multicast to manage routing topology, TypeCast inherits the resilience against mobility and link errors from the multicast protocol. This also considerably simplified the routing implementation

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512

since existing multicast protocol implementation can be used as the basis for adding TypeCast extension.

3.2. Routing Implementation

The TypeCast routing protocol is implemented on top of MAODV[24], which itself is designed as an extension of AODV[22]. The design of the TypeCast routing protocol does not impose special requirements on the underlying MANET multicast protocol; MAODV was chosen because a robust implementation was readily available. The MAODV implementation we used is from University of Maryland[17], which is built on top of AODV-UU[3]. The implementation platform is Linux 2.6.

The TypeCast routing implementation consists of a user space routing daemon and a kernel module. The user space daemon manages the routing tables for unicast, multicast and TypeCast. The TypeCast routing tables are populated based on two information sources: 1) the TYPE_ANNOUNCE packets periodically received from the routing daemons of the neighboring nodes, which contain the aggregated type information that can be reached via neighbors; 2) the *LBF* update command from the TRPC Runtime Engine on the local host, which contains the aggregated type information of the local node.

The kernel module uses the Netfilter[18] hooks to intercept the incoming and outgoing multicast data packets at various points along the network stack. The Netfilter hook will either pass or drop the packets based on whether the *subfilter* relationship exists between the *TBF* in the packet and the *FBF*s in the routing table. Bloom filter used in the implementation is 16 bytes long and is prepended to every TypeCast data packet.

4. Type-based RPC (TRPC)

4.1. Overview

RPC[6][29] based distributed object models (e.g. RMI[28], CORBA[18], Web Service[33]) have been successfully applied in building distributed systems. Such a model simplifies the development of distributed applications by providing a programming model that is similar to developing non-distributed applications.

Group-based RPC schemes (Replicated RPC[8], MultiRPC[25]) have been proposed in the past to increase system reliability, availability and concurrency in a LAN environment. These schemes usually support the same exactly-once synchronous call semantics as traditional RPC does. Such semantics is appropriate when the network bandwidth is abundant and link connectivity is relatively reliable. But it will be costly to maintain in MANETs where wireless link is highly unreliable, the overall end-to-end throughput is limited and all the target objects can be constantly moving. Furthermore, the group abstraction provided by these schemes are generally ID based, it is difficult to map them to the hierarchical organization relationship associated with users in MANETs.

Our design goal is to preserve the simplicity of RPC model and extend it for group communication in MANETs. Our proposed RPC scheme is based on type-based communication paradigm and is called TRPC with the following characteristics:

First, TRPC uses type as the basic abstraction to identity groups, and the invocation scheme obeys the subtyping and type-composition principles: when the target type of a method invocation is type T, all the objects in a MANET that are of type T or any of its subtypes should execute the method; when the target type is the composition of two types A and B, all the objects in a MANET that are both type A and B or their subtypes should execute the method.

Secondly, though type is an expressive group abstraction, real MANET applications may require additional properties to constrain the set of recipients. For instance, in a disaster relief scenario, a message may be targeted to the Firefighters within a certain range from the sender. To support such use cases, TRPC introduces the concept of *Scope* to constrain the set of target objects with non-type properties. There are two categories of Scopes: System Scope and Application Scope. System Scope models properties of the environment in which the objects are executing. Examples of System Scopes include the distance of the target nodes from the caller, the IP address of the recipients, or the geographic location of the target nodes. The significance of System Scope is that it can be directly used to assist the routing of the messages to improve efficiency. Application Scope is defined based on the intrinsic properties of the objects, such as object names. Since these attributes are application specific, they are usually not to be processed by network routing layer and are handled by TRPC Runtime Engine. The combination of type and *Scope* provides a rich and flexible group abstraction for ad-hoc applications. It also allows a point-to-point RPC to be treated as a special case of Group-based RPC scoped to a single node.

Thirdly, to reduce the performance overhead in the presence of mobility, variable channel quality and high network dynamism, TRPC adopts the at-most-once call semantics - there is no guarantee that all the objects of the target type have executed the method invocation and no retransmission will be attempted by the TRPC runtime engine. Furthermore, the method call of TRPC is non-blocking; a caller can retrieve results from a TRPC asynchronously via the abstraction of *Future*. Using asynchronous mechanism increases the concurrency at the caller side in face of potential long delay between request and response caused by multi-hop wireless paths, network partition or long service execution time. The notion of Future has been proposed before for asynchronous computation or RPC invocation[2][30], we extend it here to receive potentially multiple responses from a group of target objects associated with a single method call.

4.2. API

The main APIs for consuming and providing TRPC services are TypeCastClient and TypeCastServer (see below).

```
class TypeCastClient {
     TypeCastClient ();
     TypeCastClient(EndPoint endPoint);
      <T>  T getTypeCastProxy(Class<T> targetType);
     Object getTypeCastProxy(Class[] compositeTypes);
     void setScope(Scope scope);
}
class TypeCastServer {
    TypeCastServer ();
    TypeCastServer(EndPoint endPoint);
     <T> void export(Class<T> type, T serviceObject);
     void export(Class[] compositeTypes, Object service,
                   Properties props);
     void unexport(Object serviceObject);
}
```

TypeCastClient is used to create a client-side proxy to make a remote method call to all the objects of the target type, which may either be a single or a composite type. And *Scope* can be set to the TypeCastClient to further constraint the object set.

TypeCastServer is used to export a TRPC service object to be consumed by remote peers. The application should specify the type(s) of the object and a set of properties (e.g. name) associated with the object that can potentially be used for scope-based matching.

Scope

Figure 2 shows the type hierarchy of *Scope* currently supported in TMACS.

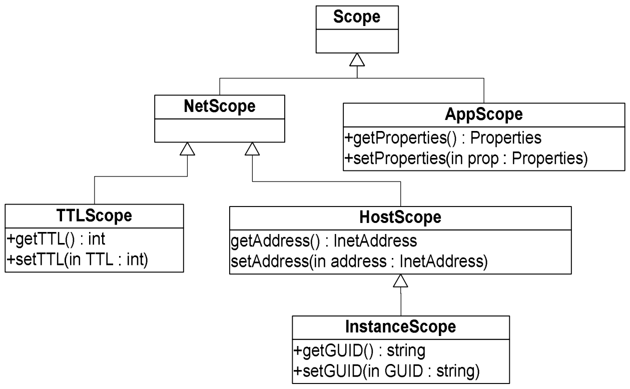

Scope
NetScope
AppScope
+getProperties() : Properties
+setProperties(in prop : Properties)
TTLScope
+getTTL() : int
+setTTL(in TTL : int)
HostScope
getAddress() : InetAddress
setAddress(in address : InetAddress)
InstanceScope
+getGUID() : string
+setGUID(in GUID : string)

Figure 2: Scope Hierarchy

There are two categories of *Scopes*: NetScope and AppScope. Netscope specifies the constraints based on properties related to the nodes hosting the objects and will be directly passed to NAL for routing optimization. AppScope specifies constraints based on the properties of the object itself and will be processed by TRPC Runtime Engine at the receiving nodes for the purpose of object matching. NetScope has two subclasses: TTLScope and HostScope. The former limits the distance of the recipients from the consumers, while the latter restricts the TRPC invocation based on IP address. If HostScope is specified, NAL will use IP unicast to tunnel the method invocation to the specified node instead of using TypeCast routing protocol.

HostScope can be further constrained with InstanceScope. InstanceScope specifies GUID of the target object. GUID is a unique identifier assigned to a distributed object by TRPC Runtime Engine hosting it. InstanceScope ensures that only a specific object on a target host will execute the method call.

Future

TRPC method call is non-blocking. If the return type of a method is void, the calling side of TRPC does not expect the response from any callee. The method invocation is simply a one-way notification to all the target objects. For applications requiring return value from the callees, the method declaration must have *Future* as the return type:

```
interface Future<T>  {
     int getCount();
     T get(int index);
     void setFutureListener(FutureListener<T> listener);
     void freeze();
     void setFreezeCondition(long timeout,  int maxCount);
}
interface FutureListener<T> {
    void  newValueArrived(Future<T> future, T value);
    void freezed(Future<T> future);
}
```

The caller can either poll result from the *Future* object at any time, or it can register a *FutureListener* to receive event notification once any new result arrives. A *Future* object can be "*freezed*" so that the caller side TRPC Runtime Engine stops waiting for responses from the target objects and any resource associated with the method invocation can be garbage collected. The caller can explicitly freeze a *Future* object or set a freezing condition based on the combination of time out value and/or minimal number of objects in the *Future*.

```
interface LocationService {
      Future<Location> getLocation();
}
class  GetNeigbhorLocation implements FutureListener<Location> {
   private ArrayList<Location> neighbors = new ArrayList();
   void locateNighbors() {
       TypeCastClient tc = new TypeCastClient();
        tc.setScope(new TTLScope(1));
        LocationService ls = tc.getTypeCastProxy(LocationService.class)
        Future<Location>  future  = ls.getLocation();
        future.setFutureListener(this);
        future.setFreezingCondition(10, -1);  // freeze after 10 seconds.
   }
   void  newValueArrived(Future<Location> future, Location location) {
          neighbors.add(location);
   }
   void  freezed(Future<Location> future) {
          …  // report neighbor locations
   }
}
```

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512

Above is a sample code fragment to use TRPC to locate all the neighboring nodes, assuming that each mobile node has a GPS receiver and can report its own location via *LocationService*

Unicast TRPC

By default, all TRPC services will be exported via a reserved multicast end point and can be accessed via either TypeCast or unicast tunneling. Applications can also choose to export the service objects via an IP unicast address to set up private communication channel between two objects. The unicast end point is usually dynamically determined by negotiating via public TRPC service.

4.3. Implementation

Figure 3 shows the internal architecture of the TRPC Runtime Engine on each mobile node:

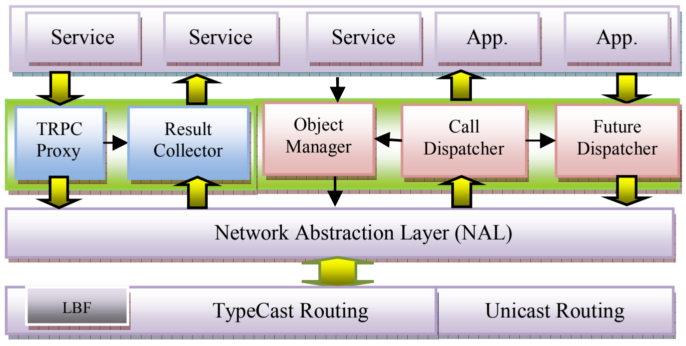


Figure 3: TRPC Runtime Engine

The TRPC Runtime Engine consists of five major components: the TRPC Proxy and Result Collector control the caller side whereas the Object Manager, Call Dispatcher, and Future Dispatcher manage the callee side functionalities. An Object Manager manages the lifecycle of all the active distributed objects and their types on a node. Whenever the aggregated types of the local active objects have been changed, Object Manager will notify the TypeCast routing layer by setting *LBF* to NAL; the updated *LBF* will be advertised to the neighbors by the TypeCast routing protocol.

All outgoing method invocations are handled by the TRPC Proxy. This is a Java dynamic proxy object that marshalls a method invocation into byte stream and encodes the target type(s) into a *Target Bloom filter (TBF)*. The *TBF*, the *NetScope* and the payload will be passed to the NAL which will forward to the target group using the appropriate routing protocol.

INVOCATION-ID	TYPE(S)	SCOPE	METHOD	PARAMETERS

Figure 4: Payload of TRPC Method Call

The marshalled content of a TRPC invocation is shown in Figure 4. The invocation ID uniquely identifies a method call on the originating node and is used to correlate the results if a method call expects return value.

All incoming TRPC method invocation are processed by Call Dispatcher. Call Dispatcher unmarshalls the payload into a method invocation, queries the Object Manager for the matching objects based on target types and scope, and subsequently dispatches the method invocation to the matching ones.

If the TRPC method call has a return value, the object will be handed over to Future Dispatcher, which will forward the result to the caller by making a special one-way TRPC invocation to the Result Collector scoped to the calling node. Result Collector at the caller will correlate the result object to the original call session, and notifying the calling application about the arrival of the new result via *FutureListener*.

5. Service Discovery

Service discovery in MANET plays the role of DNS in Internet. Its purpose is to look up the meta information of the active TRPC services so that applications can examine the returned information in order to engage communication to specific services.

In TMACS, service discovery is implemented via the coordination of the Discovery Agents on mobile nodes. The Discovery Agent is both a TRPC consumer and provider. There are two types of services provided by Discovery Agent: ServiceDiscoveryService and ServiceAdvertisementListener (see below). ServiceDiscoveryService is used to issue discovery request, and ServiceAdvertisementListener is for receiving response.

```
interface ServiceDiscoveryService extends MetaService{
      void discover(Class<T> type);
       Future<ServiceAdvertisement> discoverInPrivate(Class<T> type);
}
 interface ServiceAdvertimsentListener {
        void notify(ServiceAdvertisement serviceInfo);
  }
```

To discover whether there is any object of type T on the network, a Discovery Agent invokes the *discover* method of *ServiceDiscoverySerivce*, using type T as the parameter. When passing the discovery method invocation payload to NAL, TRPC treats it differently from other methods: instead of setting *TBF* to the Bloom filter of *ServiceDiscoveryService*, it sets it to the Bloom filter of T.

Since the *TBF* is set as T, the discovery request will be routed exactly as if the target type is T. Since the packets will be routed to only the nodes having objects of type T or its subtypes, this avoid wasting network bandwidth without always flooding the whole MANET with discovery solicitation.

When the Discovery Agent on a node receives the discovery request for a type T, it finds all the matching objects on the local host. If the set is not empty, the node will issue ServiceAdvertisementListener.notify(ServiceAdvertisement) to send the meta information of the matching objects to all interested nodes via TypeCast. The meta information includes the object type(s), the IP address of the host, the GUID of the object, and any other properties associated with the object. The using of TypeCast-TypeCast coordination pattern enables a single response to satisfy potentially multiple requestors. The response will be cached by all the Discovery Agents. The cached value can be used to satisfy further discovery requests on the same type and suppress redundant discovery messages.

ServiceDiscoveryService also defines a second method discoveryInPrivate() to use TypeCast-Unicast coordination pattern. When this method is invoked, the discovery response will be only sent back to the original requestors via Unicast in the *Future* result.

Since discovery is a TRPC method call, clients can use *Scope* to constraint the set of recipients. For instance, application can use TTLScope to lookup services only on the immediate neighbors, or AppScope to look up a service with a specific name.

A Discovery Agent can also proactively advertise a service by calling the *notify* method of *ServiceAdvertisementListener* without waiting for the discovery request. Such a mechanism can result in wasted bandwidth consumption and should use judiciously by properly scoping the advertisement.

Using TRPC has greatly simplified the implementation of discovery service. Moreover, the abstractions (e.g. type and *Scope*) used for expressing the invocation targets are reused for expressing the discovery targets, resulting in a simple and uniform programming model. TMACS' discovery service provides much needed flexibility to allow applications to select different coordination patterns (TypeCast-TypeCast vs. TypeCast-unicast) and responsive models (proactive vs. reactive) based on the requirements and the access pattern of individual applications. By utilizing TypeCast and intelligently setting *TBF*, our discovery service is much more "directional" for finding the required objects and achieves better efficiency than existing MANET discovery services relying on broadcast or multicast (e.g. [7] [14]). Using a single network protocol for both MANET routing and discovery also distinguishes TMACS from the solutions relying on separate discovery infrastructure [13][15][26].

6. Case Studies

6.1. Ad-hoc Caching Service

Caching is an important middleware service to improve service availability in the dynamic network environment with high mobility. With caching, the service consumers can access a service from a cached server closer to it than the original service provider. Here we build a light-weight distributed caching systems with TMACS that is tailored to the peer-to-peer nature of MANET environments. The caching architecture allows any node in a MANET to voluntarily serve as caching server for other mobile peers, and there is no restriction on what type of services can be cached.

The ad-hoc caching is managed by CacheManager object on each node. The CacheManager objects coordinate among each other to fulfill the caching requests via two simple TRPC services: CacheService and ContentSource:

```
interface CacheService {
   void requestCache(EndPoint contentEndPoint,  ObjectDesp object);
}
interface ContentSource {
   Future<byte[]>  getContent(String GUID);
}
```

The CacheService is the interface for mobile pees to send caching requests; the ContentSource is the interface for the caching node to retrieve the object state from the requestor. The coordination among the CacheManager objects utilizes the TypeCast-Unicast pattern: when a node has an object to be cached, it sends a request to all the activated caching nodes by calling CacheService.requestCache(). The call consists of two parameters: the first is a unicast end point from which the object state can be retrieved. The second contains the meta information about the object, including its type T, the implementation class name, the cache expiration time, the GUID of the object and the size of the object state.

When the CacheManager on a caching-enabled node receives the caching request, it checks a) whether there is sufficient memory and storage on the local host, and b) whether the local policy allows caching T. When all conditions are met, the node will invoke ContentSource.requestContent() to retrieve the object state via the private end point specified in the request. The returned value will be used to instantiate a replicated copy. For each cached replica, the CacheManager will create and export a CacheProxy object associated with it (Figure 5). The CachedProxy is a Java dynamic proxy object implementing two interfaces: one is type T of the cached object, another is the CacheControl interface shown below:

```
interface CacheControl  {
  void   invalidateCache(EndPoint contenSourceAddr,  String GUID );
   void   refreshCache(EndPoint contentSourceAddr,  String GUID);
}
```

CacheControl provides methods for the content source to request all the caching nodes to either invalid or reload the cached copies when the original data content is deleted or changed before cache expires. This allows the content source to maintain consistency among cached copies if the nature of the applications requires so. When making cache control calls, the content source will set the target type to be the *composition* of CacheControl and T, thus ensure that the method will only be executed by the nodes that have the right types of cached copies and avoid flooding the requests needlessly to other caching nodes. Since the CacheProxy is of type T, it can accept any TRPC call targeting to T. Such call will be delegated to the cached replica (Figure 5). For

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512

each call on T, CacheProxy will also update the access frequency and timestamp. This information will be used as input to the cache replacement algorithm when deciding which cache replica should be removed from the cache pool when the pool is full.

Even though there are complex coordination patterns among CacheManages, using TRPC has greatly simplify the development work by eliminating the efforts to write low level protocol handling and message processing. The use of type-composition for cache control invocation further demonstrates the expressiveness of TMACS' programming model. More complex caching strategies (e.g. [34]) can be exploited by utilizing on TMACS provided system service and programming abstractions.

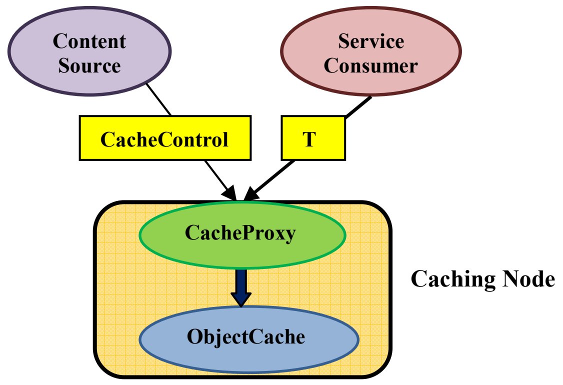
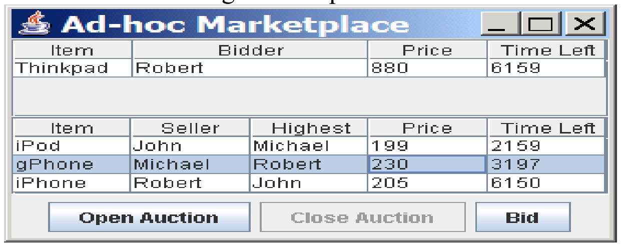


Figure 5: Cache Architecture

6.2. Ad-hoc Auction/Marketplace

EBay-like auction services have proven to be a highly successful business model for conducting commerce on the Internet. A MANET provides another ideal platform for such type of services. Unlike the Internet, the ad-hoc nature of a MANET cannot assume the availability of a trusted broker during the transaction; such a disadvantage can be offset by the physical proximity of bidders and sellers, the local and serendipitous nature of MANETs. For instance, people with extra concert tickets can auction the tickets at the concert hall right before the show starts by forming a MANET with those who are waiting there to purchase the tickets.



Item	Bidder	Price	Time Left
Thinkpad	Robert	880	6159

Item	Seller	Highest	Price	Time Left
iPod	John	Michael	199	2159
gPhone	Michael	Robert	230	3197
iPhone	Robert	John	205	6150

Figure 6: Screenshot of Ad-hoc Auction

Our ad-hoc auction implementation (see Figure 6 for UI) follows the peer-to-peer model, letting sellers and bidders directly engage in the auction process. A seller opens an auction by providing the item to sell, the starting price and the closing time. The auction status will be advertised to potential bidders via TypeCast periodically or whenever the bidding price is updated. Each buyer submits its bid via Unicast to the seller, and the price will be automatically updated based on the highest bid. When the auction ends because the closing time is reached or the seller has decided to accept a bid, a confirmation message will be sent to the highest bidder via Unicast and the closing event will be TypeCast to all the bidders.

The TRPC interfaces used for coordination among sellers and bidders are shown below:

```
interface AuctionEventListener  {
   void auctionStarted(Item item, SellerInfo seller, AuctionStatus state);
   void auctionClosed(Item  item, SellerInfo seller, AuctionStatus state);
   void auctionUpdate(Item item, SellerInfo seller, AuctionStatus state);
}
interface Seller {
     void  bid (Item item, BidderInfo bidder, int price);
     Future<AuctionStatus > query(Item  item);
}
interface Bidder {
        void confirm(Item item, SellerInfo seller);
        Future<Acknowledgement> isAlive();
}
```

The *AuctionEventListener* is used by seller to notify bidders about auction status via TypeCast. The notification includes the description of the items for sale, the seller information such as the private TRPC end point for submitting bid, and auction status (e.g. the current bidding price and the closing time). The *Seller* is the TRPC interface for bidder to query the auction status of an item or submit a bid. The *Bidder* is the interface for the seller to send the confirmation to the final auction winner. It also provides a method for the seller to query the reachability of the current highest bidder. In the event of network partition or node leaving the network, the seller can invalid the current highest bid and restart the auction process. The *Seller* and *Bidder* interface are exported via private unicast end points. These private end point information are exposed to the auction participants in the *SellerInfo* and *BidderInfo* objects exchanged during the auction process.

TMACS greatly simplifies the implementation of the ad-hoc application: it eliminates the efforts to design specific auction protocols as adding a protocol is the equivalent of adding a new TRPC method. The implementation effort was spent primarily to develop the user interface. The total line of Java code for the auction application is just over 1000, and it took a student about 3 weeks to finish the implementation.

Our current implementation uses *AuctionEventListener* to convey information on all ongoing auctions. A future extension is to have a hierarchical *AuctionEventListener* so that buyers can activate a specific type of listener for only the kinds of items that he/she is interested in. For instance, we can have *TicketAuctionEventListener* for auctioning tickets only.

7. Experimentation

Large-scale simulation has been conducted in our earlier work[16] to study the performance and scalability of Type-Cast routing protocol. This paper focuses on evaluating the implementation of the integrated software stack of TMACS and application on physical test bed.

7.1. Test Bed Setup

We use eight laptops connected in an IEEE802.11 ad-hoc network in the topology illustrated in Figure 7. Among them, five are Dell Latitude D600 and three are IBM Think-Pad T42. All are configured with the Linux Fedora Core 4 with 2.6.12 kernel. Due to the lack of controlled physical space for wireless transmissions, iptables[18] are used to enforce the network topology to guarantee that the packets go through multiple hops. Packets from non-neighbor nodes are automatically dropped at the MAC layer based on MAC address filtering.

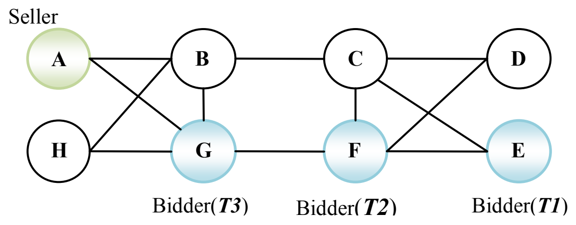


Figure 7: Topology of Testbed

We used the ad-hoc auction application for all our performance measurements.We wrote a *Bid Bot* software as the bidder that automatically submit a bid whenever the *AuctionEventListener.auctionUpdate()* is invoked. To measure the impact of the type hierarchy, we created four subtypes T1, T2, T3 and T4 under AuctionEventListener, and they form a linear hierarchy as AuctionEventListener →T1 → T2 → T3 →T4. We used three bid bots, with types and hosting nodes as shown in Figure 7. The seller runs on node A; it periodically calls the *actionUpdate*() method of the target type and records the responses in its *Seller.bid()* method invoked by the bidders. The target type varies from T1 to T4. According to subtyping principle, when the target type is T1, all three bidders should submit bids. For target type T2, two bidders on node G and F should respond. For T3, only bidder on G should bid.

The IP payload of the *auctionUpdate()* method invocation after marshalling is 661 bytes, and the payload of the *bid()* method invocation is 416 bytes. *auctionUpdate()* uses TypeCast routing, while *bid()* is called via unicast.

7.2. Performance of Ad-hoc Auction

The metrics we use are as follows:

- **One-way goodput**: the percentage of the *auctonUpdate* method executed per target bidder.
- **Two-way goodput**: the percentage of the actual *bids* received by the seller out of the expected total *bids*.
- **Call latency**: the time difference between the receiving of *bid* and the start of the *auctionUpdate* measured by the seller. Since the path discovery phase of AODV/MAODV can take up to a few seconds and has a distortion effect on the average latency, when calculating latency, the value within the first 60 seconds will not be used.
- **TypeCast traffic ratio**: the number of TypeCast data packets transmitted per *auctionUpdate* method call. A smaller value means better efficiency. The metric measures whether Bloom filter based TypeCast routing mechanism can effectively filter data packets based on the target type.

In each experiment, the seller invokes *auctionUpdate* method of a fixed target type at regular interval for 300 seconds. Each experiment is repeated at least 12 times for each target type. Figure 8 shows the results when the invocation rate of *auctionUpdate* method is 1 per second.

From the graph, we can see that the one-way and two-way goodput are both above 90% for different target types, and the two-way goodput is almost identical to the one-way goodput. This implies that both the TypeCast *auctionUpdate* method and the unicast *bid* method invocation have very high success ratio. For object types that are lower on the type hierarchy, the goodput (both one-way and two-way) increases slightly. This results from the combination effect of the smaller number of target objects (which should result in less data traffic) and the shorter average distance between the seller and the target bidders. The overall call latency is small, and its distribution along the type hierarchy reflects the average distance between the seller and the target bidders identified by the type.

When the target type goes down the type hierarchy from T1 to T3, the number of target objects shrinks from 3 to 1. The TypeCast traffic ratio drops proportionally, which confirms the effectiveness of the TypeCast filtering mechanism. When the target type is T4, since there is no matching target object in the network, the TypeCast traffic ratio is close to zero, implying that no network bandwidth is wasted on

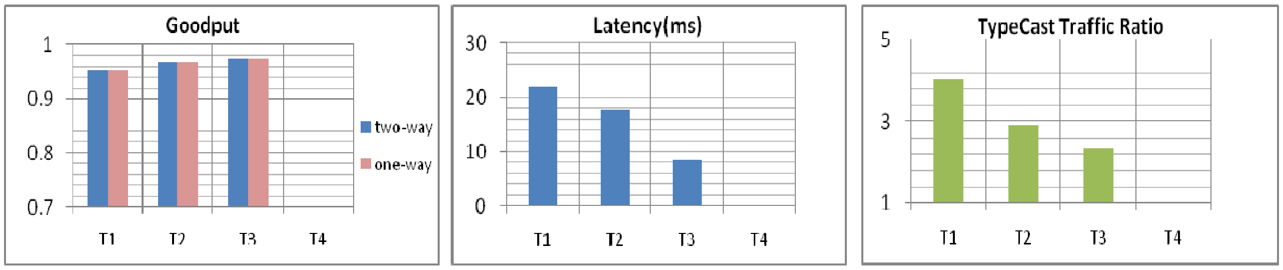


Figure 8: Performance of Ad-Hoc Auction under Different Target Types

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512

transmitting such data packets. If the experiment runs with multicast instead of TypeCast, as all nodes belong to the same multicast group, the network traffic will not be filtered by an object's type, thus wasting significant bandwidth, a critical resource in a MANET.

We repeated the experiments with different sending rates. In general, the goodput will decrease when the sending rate becomes larger, but the correlation between TRPC latency, TypeCast traffic ratio and the target type remains the same.

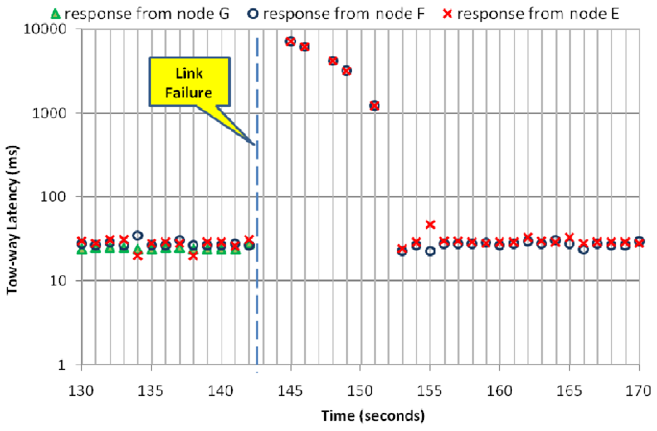


Figure 9: System Response during Link Failure

Link Failure

A MANET experiences link failures due to mobility or variable channel conditions. To evaluate the robustness of the system against such failures, in the next experiment, all links from node G are made to fail. As the original multicast path is A→G→F→E, this will disrupt the packet forwarding route until the tree is reformed (presumably using the path based on the new topology A→B→C→F/E).

Figure 9 shows the latency of *bid* responses observed by the seller for each *auctionUpdate* issued at time t. Before disconnection of G, which happens at t=142s, the seller receives all three bids per auction request, and the latency is mostly between 20~40 ms. After the disconnection, it takes the routing layer about 3s to detect the link failure, and all the *auctionUpdate* invocations issued during this phase are lost. (The duration of the failure detection period is determined by HELLO message transmission frequency and number of permitted retransmits by MAODV).

Once the disconnection is detected, MADOV takes about 7 seconds for the route discovery and reestablish process. The current MAODV implementation does not take advantage of the *expanding ring search* algorithm provided in AODV, which will considerably reduce the discovery latency (inclusion of this optimization is currently in progress). The packets issued during the discovery phase were buffered until the routing tree was reestablished. The long invocation latency observed by seller within this phase is predominately contributed by the queuing delay. There is also relatively higher ratio of packet loss (>30%), which we believe is caused by the burst transmission when flushing the buffered packets as the tree is reformed.

The new routing tree was successfully established about 10 seconds after link failure occurred; the seller successfully received the bids from the two remaining connected bidders.

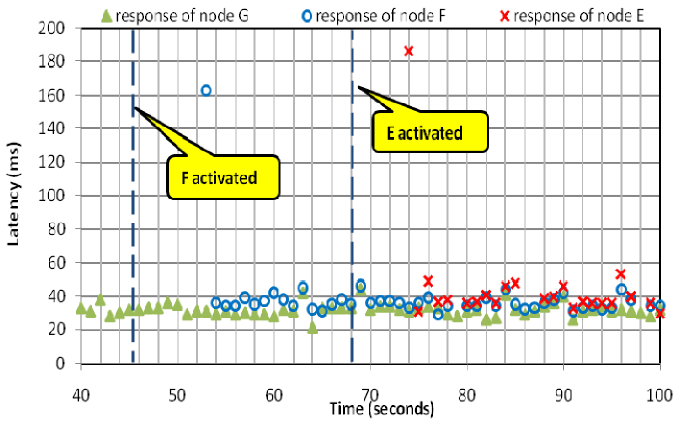


Figure 10: System Response for Dynamic Object Activation

Dynamic Object Activation

In this experiment, we study system responsiveness when a mobile node with a TRPC service object joins a network. The same network setup and object distribution is used as before. However, instead of simultaneously activating bidders, we now activate them incrementally in the order of G, F and E. The seller issues the *autionUpdate* at 1 per second continuously. Figure 10 shows the bid latency observed by the seller for each *auctionUpdate* invoked at time t. Note the delay of approx 8s, between when the new bidder starts to when it receives the first *auctionUpdate*. This latency, called *activation latency*, is almost entirely caused by the route discovery process for the server node to join the multicast tree, which was shown in the previous experment to be at least ~7 s based on the current MAODV implementation. Once the bidder joins the multicast tree, the seller is able to receive the bid from the newly activated bidder, without introducing a significant drop of goodput during or after the bidder's activation.

It should be noted that the route discovery overhead is only incurred when the first object is activated on a node. For subsequent object activations on the same node, the same multicast tree will be used; only the new type information will be updated among the neighbors via TYPE-ANNOUNCE packets. Our experiments (results not shown here due to space limitation) have shown that the average activation latency for subsequent objects is within 1s.

The graph also shows that the first method call of the newly activated object has a much higher invocation latency than the following ones. This is primarily due to the cost of route discovery process for finding the unicast path back for the *bid* method. Since AODV implementation uses *expand-*

ing ring search algorithm, the discovery latency for unicast is much shorter (< 200ms).

To summarize, the system can quickly react to new mobile nodes/objects joining the network and properly direct the TRPC calls to the newly joined objects. The activation latency is ~7 s for the first object on a host and < 1 s for any subsequent objects.

7.3. Performance of Discovery Service

In this experiment, we measure the performance of TMACS' discovery service. The same network and application set up is used. Instead of invoking *auctionUpdate* method, the Disocvery Agent on node A invokes discovery request for a target type periodically, and the discovery responses via TypeCast are collected. We measure discovery success ratio, discovery latency, and discovery traffic ratio (defined as number of TypeCast data packets transmitted per discovery request).

Figure 11 shows the result of discovery experimentation when the invocation rate of discovery requests is 1 per second. Since the *TBF* (Target Bloom Filter) of discovery requests is set to the target type to be discovered, the requests are routed only to the nodes that have the target types. This can be confirmed from the discovery traffic ratio, which decreases proportionately down the type hierarchy, just as it should for invocation of a TRPC method to the same type. When the network does not have any objects belonging to the target type (e.g. T4), the discovery traffic is reduced to almost zero, as before. If multicast is used here for sending discovery request, the data traffic will be constant even when there is no intended object in the network.

8. Related Work

Jini[27] is a distributed computing platform providing discovery service and a tuple space[11] based programming model. Jini's discovery service relies on dedicated lookup servers, thus introduces single point of failure. The tuple space based programming model is a powerful abstraction for group coordination, but its requirements on global storage space and strong consistency and persistence semantics can result in high system and network overhead in MANET. To address this limitation, Lime[18] has relaxed the semantics of standard tuple space by letting each node or agent host its own tuple space, and multiple hosts/agents proximate to each other can transiently share tuple spaces with common names. Though Lime provides explicit abstraction for data sharing among a group of nodes/agents, its name-based group scheme is not as rich and flexible as what TMACS supports. Lime like coordination model can be built on top of TMACS as a higher level system service.

Publish/Subscribe[9] is a data-centric middleware for distributed and mobile systems. In this scheme, senders publish events, which will be routed by a network of event brokers to all receivers subscribed to them. Traditional RPC model is less favorable than publish/subscribe in mobile systems due to its point-to-point and synchronous invocation semantics. TRPC is designed to explicitly address the limitation of RPC while keeping its benefit. Publish/Subscribe is very useful for in-network processing, filtering and aggregation based on data content; while TRPC is more suitable with service-oriented architecture, in which distributed applications are exposed as services, and the data content is transparent to the intermediate routers. For many applications and products already developed with RPC style model, TRPC provides an easy path to port them from point-to-point, wired environment to many-to-many wireless environment.

Intentional Naming[1] is a naming system targeted for mobile systems. It builds an overlay network to forward messages based on high level application names consisting of arbitrary attribute-value pairs. The routing table stores a name tree which is disseminated within the overlay network. Though the naming scheme of Intentional Naming is expressive, the complexity and overhead of building an overlay network on MANET and storing application specific attributes directly in the routing table can potentially limit its broad adoption.

SpatialViews[20] provides a high level programming notation for MANETs. The core abstraction is *spatial view* and *spatial view iterator*. *Spatial view* defines a virtual network consisting of services confined to a location-time region. The *spatial view iterator* is used to discover the nodes bound to a *spatial view* and migrate computation to them. The requirement of program migration may limit the adoption of SpatialViews on resource or security-sensitive platforms. In TMACS, the mobile nodes are identified via type and *Scope*, and the collaboration is achieved via remote invocation instead of mobile code. It is more light-weight with respect to the requirements on the run-time environment. SpatialViews-like sophistic programming notation can leverage TMACS as the building block for its run-time implementation.

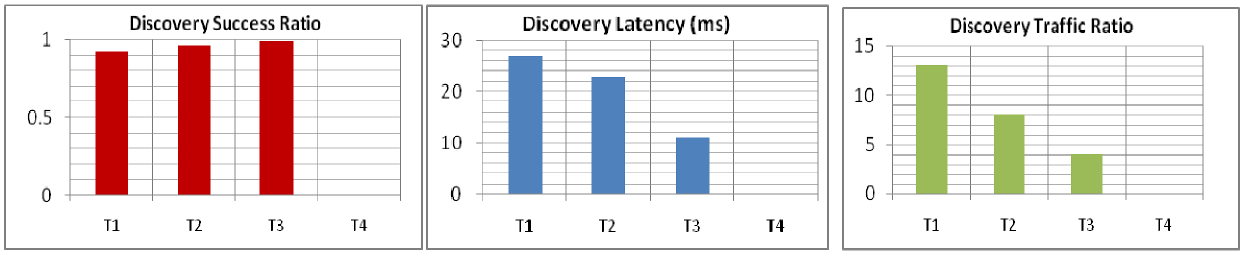


Figure 11: Performance of Discovery Service

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512

M2MI[12] is a Java framework providing interface-based group communication primitives for ad-hoc collaboration in wireless networks. Comparing to TMACS, it did not address subtyping, type composition, asynchronous invocation and service discovery. M2MI uses broadcast to transmit packets, which is impractical in multi-hop wireless network due to high contention caused by flooding.

Hood[32] and Abstract Region[31] are programming abstractions for sensor networks to simplify data sharing, filtering and processing among neighboring nodes These data centric notations are different from TRPC's service-oriented approach. It will be interesting to exploit TRPC like group communication paradigm in sensor networks.

9. Conclusion

In this paper, we present the design and implementation of TMACS, a middleware to support distributed applications in MANETs. TMACS provides TRPC as the distributed programming model for invoking a method on a group of distributed objects. The group is identified by the common type(s) shared by the objects and can be further constrained with *Scope*. The results of an invocation are retrieved asynchronously via *Future*. TMACS also provides a fully distributed, resilient discovery service.

A physical implementation of TMACS was deployed on a testbed. We implemented a distributed caching service and an ad-hoc auction application to demonstrate the expressiveness of TMACS and implemented TypeCast routing protocol to efficiently support TRPC and discovery service. Our experiments demonstrate the feasibility and effectiveness of TMACS as a potential distributed platform for building ad-hoc applications in MANET. Our future work is to conduct experimental study on a larger-scale test bed with more realistic mobility patterns.

10. Acknowledgement

This work is funded by the National Science Foundation under NR grant ANI-0335302 *WHYNET: Scalable Testbed for Next Generation Mobile Wireless Networks Technologies* and US Army Research Grant W911NF-05-1-0246 *DAWN: Dynamic Ad-hoc Wireless Networking*.

11. References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley, "The design and implementation of an intentional naming system", in SOSP'99, pp186-201.

[2] A. L. Ananda , B. H. Tay , E. K. Koh, A survey of asynchronous remote procedure calls, ACM Operating Systems Review, v.26 n.2, p.92-109, April 1992

[3] AODV-UU, http://core.it.uu.se/core/index.php/AODV-UU

[4] A. Bakre , B. R. Badrinath, "M-RPC: a remote procedure call service for mobile clients", Proceedings of the 1st annual international conference on Mobile computing and networking, p.97-110, November 13-15, 1995

[5] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors", Communications of ACM, 13(7), July 1970, pp 422-426.

[6] A. Birrell , B. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems (TOCS), v.2 n.1, p.39-59, Feb. 1984

[7] C.Campo, M.Munoz, J.C.Perea, A.Marin, C.Garcia-Rubio,"PDP and GSDL: a new service discovery middleware to support spontaneous interactions in pervasive systems", in PerCom'05.

[8] E. C. Cooper, "Replicated distributed programs", Proceedings of the tenth ACM symposium on Operating Systems Principles, p.63-78, Dec. 1985.

[9] P. Eugster, P. Filber, R. Guerraoui, A. Kermarrec, "The Many Faces of Publish/subscribe", ACM Computing Surveys", Vol. 35, Issue 2, 2003, pp.114-131

[10] H. Frey, D, Görgen, J. K. Lehnert, P. Sturm, "Auctions in Mobile Multihop Ad-hoc Networks Following the Marketplace Communication Pattern", in ICEIS'04.

[11] D. Gelernter, "Generative Communication in Linda", Trans. On Programming Languages and Systems, vol 7, no. 1, Jan. 1985, pp80-112.

[12] A. Kaminsky and Hans-Peter Bischof, "Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems," in OOPSLA'02.

[13] W. Gao, "Towards Scalable and Robust Service Discovery in Ubiquitous Computing Environments via Multi-hop Clustering," in mobiquitous'07

[14] S. Helai, N. Desai, V. Verma, C.Lee, "Konark-A Service Discovery and Delivery Protocol for Ad-hoc Networks", in WCNC'03, pp2107-2113.

[15] U. Kozat and L. Tassulas, "Network Layer Support for Service Discovery in Mobile Ad Hoc Networks," in INFOCOM '03.

[16] J. Lin, T. Phan and R. Bagrodia,"TypeCast -- Type-based Routing in Wireless Adhoc Networks", in Mobiquitous'06.

[17] MAODV-UMD, http://www.hynet.umd.edu/research/maodv/MAODV-UMD.html

[18] A. L. Murphy, G. P. Picco, and G. Roman. "Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents". ACM Transactions on Software Engineering and Methodology, vol. 15, no. 3, pp. 279-328, July 2006

[19] Netfilter, http://www.netfilter.org

[20] Y. Ni, U. Kremer, A. Stere, and L. Iflode, "Programming Ad-hoc Networks of Mobile and Resource-Constainted Devices", in PLDI'05.

[21] Object Management Group, "Common Object Request Broker Architecture", http://www.omg.org/

[22] C. E. Perkins and E. M. Royer. "Ad hoc On-Demand Distance Vector Routing." Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, February 1999, pp. 90-100

[23] M. Petrovic, V. Muthusamy, H. Jacobsen, "Content-Based Routing in Mobile Ad Hoc Networks," mobiquitous'05, pp. 45-55,

[24] E.M.Royer and C.E.Perkins, "Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol", Mobicom'99, pp.207-218

[25] M. Satyanarayanan, E.H. Siegel, "Parallel Communication in a Large Distributed Environment," IEEE Transactions on Computers, vol. 39, no. 3, pp. 328-348, Mar., 1990

[26] F. Salihan, V. Issarny, "Scalable Service Discovery for MANET", in PerCom'05

[27] Sun Microsystems Inc. "Jini Specification", http://www.sun.com/software/jini

[28] Sun Microsystems Inc., "Java RMI Specification", http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html

[29] B. H. Tay , A. L. Ananda, A survey of remote procedure calls, ACM SIGOPS Operating Systems Review, v.24 n.3, p.68-79, July 1990

[30] E.F. Walker, R. Floyd and P. Neves, Asynchronous remote operations in distributed systems, Proc. 10th International Conference on Distributed Computing Systems (ICDCS-10) , Paris, France (1990) pp. 253-259

[31] M. Welsh , G. Mainland, "Programming sensor networks using abstract regions," in NSDI'04

[32] K. Whitehouse , C. Sharp , E. Brewer , D. Culler, "Hood: a neighborhood abstraction for sensor networks," in Mobisys'04, p99-110

[33] The World Wide Web Consortium (W3C), "Simple Object Access Protocol", http://www.w3.org/TR/soap/

[34] L. Yin and G. Cao, "Supporting Cooperative Caching in Ad Hoc Networks," IEEE Transactions on Mobile Computing, January, 2006, p77-89

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3512
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3512