Contrail: Enabling Decentralized Social Networks on Smartphones

Patrick Stuedi[1], Iqbal Mohomed[2], Mahesh Balakrishnan[3], Z. Morley Mao[4], Venugopalan Ramasubramanian[3], Doug Terry[3], Ted Wobber[3]

[1] IBM Research, Zurich, Switzerland
stu@zurich.ibm.com
[2] IBM Research, T.J. Watson, USA
iqbal@us.ibm.com
[3] Microsoft Research, Silicon Valley, USA
{maheshba,rama,terry,wobber}@microsoft.com
[4] University of Michigan, USA
zmao@umich.edu

Abstract. Mobile devices are increasingly used for social networking applications, where data is shared between devices belonging to different users. Today, such applications are implemented as centralized services, forcing users to trust corporations with their personal data. While decentralized designs for such applications can provide privacy, they are difficult to achieve on current devices due to constraints on connectivity, energy and bandwidth. Contrail is a communication platform that allows decentralized social networks to overcome these challenges. In Contrail, a user installs content filters on her friends' devices that express her interests; she subsequently receives new data generated by her friends that match the filters. Both data and filters are exchanged between devices via cloud-based relays in encrypted form, giving the cloud no visibility into either. In addition to providing privacy, Contrail enables applications that are very efficient in terms of energy and bandwidth.

1 Introduction

The emergence of powerful smartphones and ubiquitous 3G connectivity has led to a number of new mobile applications. Many of these applications are centered on social networking, where users on mobile devices want to selectively consume content generated by their friends' devices. For example, Alice wants to receive pictures taken by her friends in which she is tagged, view status updates by her friends mentioning the movie "The Social Network", and be notified of her child's location if he strays too far from home.

Today, such applications exist in the form of centralized services such as Facebook, FourSquare or Flickr; new content generated by a device is first uploaded to a central server, which then selectively redistributes it to other devices. A centralized version of the child-tracking application would have the child's phone periodically update a central server with his location; the server would then notify Alice if the location is outside bounds specified by her. Centralized solutions are simple and efficient, allowing a device to upload data just once to the cloud in order to share it with multiple recipients, without requiring any of them to be online at the same time.

However, centralized solutions come at the cost of user privacy. Individuals are forced to trust corporations to not misuse their data or sell it to third parties. They must also trust companies to guard their data against malicious hackers or repressive governments. These concerns are amplified by the very personal nature of data generated on mobile devices. In the example of Alice's location-tracking application, the central server knows both the current location of her child as well as the location of Alice's home. While privacy requirements are subjective and vary from person to person, today's technology offers a stark choice: give up privacy or stay offline.

In contrast, decentralized designs can offer better privacy to end-users. Since our focus is on privacy, we use the term 'decentralized' to refer to any system where a user's data can be viewed unencrypted only on trusted devices, and not at any intermediate point in the network. We expect such systems to execute application logic exclusively on edge devices, using encrypted channels between devices to coordinate across them. Decentralized designs for privacy-aware social networks have been explored in the context of wired end-hosts [1, 3].

Unfortunately, implementing decentralized applications on modern smartphones is challenging. At a basic level, getting messages from one device to another can be surprisingly difficult; smartphones and the wireless 3G/4G networks they run on are designed for simple client-server interactions, not inter-device communication. Assuming smartphones can somehow exchange messages, a more complex challenge for decentralized applications relates to minimizing communication, a crucial goal in the context of battery limitations and bandwidth caps.

In this paper, we present Contrail, a communication platform that enables efficient, decentralized social networks on smartphones. At the heart of Contrail is a simple cloud-based messaging layer that enables basic connectivity between smartphones, allowing them to efficiently and securely exchange encrypted data with other devices. Over this messaging layer, Contrail implements a novel form of publish/subscribe that uses *sender-side* content filters to minimize bandwidth and energy usage while preserving privacy. Additionally, Contrail provides mechanisms that are critical for reducing the energy and bandwidth footprint of applications, such as the ability to flag in-flight data as expired or obsolete.

Contrail's content filters allow devices to selectively receive subsets of data produced by other devices. When Alice wants some data from Bob – for example, all photos taken by Bob in Seattle – she attempts to install a content filter on his smartphone expressing her interest. If Bob agrees to install this filter on his device (he can choose to decline the request), all subsequent photos taken by him in Seattle are routed to Alice's phone. Similarly, Alice could install a filter on her child's phone expressing her interest in his location if he leaves a certain bounding area. Content filters support a wide range of social network applications, including location-based services, photo and video sharing, message walls and social games.

Contrail is implemented on the Windows Azure cloud platform and on Windows Mobile 6.5 devices. Our evaluation shows that this implementation offers latency and throughput between edge devices that is limited only by current 3G network speeds. We have also implemented several social network applications on Contrail, including location-tracking and photo-sharing. This paper makes the following contributions:

- We describe the challenges faced in implementing a decentralized social network on smartphones, and translate these into a set of requirements for a communication platform.
- We describe the design of the Contrail system, which combines the novel idea of sender-side content filters with other techniques to enable efficient social networks on smartphones.
- We present an implementation of Contrail on Windows Azure and Windows Mobile 6.5, and evaluate its performance.

2 Problem Statement

Our primary goal is to enable decentralized social network applications on smartphones. As described, we expect such applications to obtain privacy by placing logic at edge devices and coordinating via encrypted channels. In this section, we elaborate on the challenges such applications face.

We use the child-tracking application as a running example. Consider a simple implementation of this application — once every five minutes, the child's (let's call him Junior) device generates a location update, encrypts it, and sends it to Alice's phone. On Alice's phone, the update is decrypted and then checked against predefined bounds (that correspond to Alice's home, for example). If Junior is out of bounds, an alarm is triggered on Alice's phone. This implementation is decentralized – no central server sees Junior's location or Alice's interests – and consequently offers privacy.

As we mentioned, the first challenge faced in building such an application is basic connectivity: Junior's phone can't easily send messages to Alice's phone. 3G/4G networks do not usually support incoming TCP connections. Even when they do, smartphones are disconnected more often than not; devices can be in low-signal areas, run out of battery, have power-aware radios that sleep intermittently, or simply be turned off. In fact, two devices that wish to communicate with each other may never be online simultaneously. As a result, conventional tunneling solutions used in wired networks do not translate well to this setting.

One option for connectivity is to use existing solutions meant for decoupled communication, such as SMS or e-mail. Junior's phone can send its current location to Alice's phone inside an e-mail. Since SMS and e-mail use centralized servers only as "dumb" message relays, their payloads can be encrypted, offering private communication channels between devices. However, these mechanisms are designed for human-readable content, and can be slow, bulky and inflexible when used as a general message transport.

More fundamentally, transports such as e-mail or SMS offer no support for building efficient social networks on smartphones. To understand this point, we outline a number of key dimensions of efficiency. We also illustrate how the location-tracking application (implemented over e-mail) fails to be efficient on each count.

Download Efficiency: A device should only download data it is interested in. *Alice's phone receives a constant barrage of updates from Junior's phone even when he's at home, draining her battery and using up bandwidth.*

Upload Efficiency: A device should only upload data if some other device is interested in it. *Junior's phone continuously uploads location updates even when he's at home, using up energy and bandwidth.*
Multicast Efficiency: A device should upload data only once for multiple recipients. *Bob wants to know where Junior's phone is, as well. If Junior's phone sends separate messages to Bob and Alice, it now drains even faster and uses up more bandwidth.*
Semantic Efficiency: A device should only download data that is not expired or obsolete. *When Alice turns on her phone after keeping it switched off for a meeting, she receives a flood of location updates from Junior's phone, even though she only cares about his last location.*

Some of these properties (such as upload and download efficiency) can be achieved via extra application logic, while others (such as multicast and semantic efficiency) require explicit hooks from the transport layer. Clearly, the simple decentralized implementation of the location-tracking application that uses e-mail as a transport fails to offer any of these efficiency properties (except multicast efficiency, since a single e-mail can be uploaded once for multiple recipients). In contrast, a purely centralized solution does not provide privacy, but does offer all the efficiency properties (except upload efficiency).

Required is a transport layer that makes it trivial for applications to achieve all four efficiency properties while also providing decoupled connectivity and privacy. In essence, these efficiency properties amount to ensuring that data is only uploaded and downloaded by devices when absolutely necessary. For a transport layer to assist applications in achieving this goal, it has to understand application-level requirements; in other words, the application has to specify to the transport layer which devices require what data.

Why not use existing Pub/Sub implementations? Publish/subscribe interfaces are a natural fit for this problem. In a pub/sub system, the application running on each node *subscribes* to specific data; for example, a server might wish to receive stock quotes of MSFT if it is above $25. Subsequently, data *published* by other nodes — such as updates to the MSFT stock price — is routed selectively to other nodes based on their subscriptions.

Unfortunately, existing pub/sub implementations do not provide the guarantees we need to build decentralized social networks. Pub/sub systems typically *filter* data — i.e., match data to subscriptions — at centralized servers, in which case they do not provide privacy. Alternatively, they filter data at the edge receivers, in which case they cannot provide the upload and download efficiency properties; data must be uploaded by the sender and downloaded by the receiver before it can be determined if the receiver really wants it.

More generally, an important goal of publish/subscribe systems is *anonymous* communication, where senders can transmit data to interested receivers without having to know and enumerate their identities. In contrast, we are interested in secure, private communication between trusted nodes. This leads us to make very different design choices from current pub/sub systems, as will become clear in the following sections.

3 Design of Contrail

Here, we provide a high-level description of Contrail's design. We describe the two main mechanisms in Contrail – sender-side filters and cloud relays – and explain how they provide the properties enumerated in the previous section.

3.1 Sender-Side Filters

The Contrail universe consists of users, the devices belonging to those users, and cloud-based relay servers. In a brand new instance of Contrail, no device sends or receives messages; from this starting point, we progressively describe how communication occurs. Two kinds of messages exist in Contrail — *filter installation requests* and *data messages*. First, we describe when and why these messages are sent between devices; later, we will describe how they are sent.

A Contrail filter is simply an application-defined function that accepts some unit of data as input and returns true or false. Filters are installed by one device (we call this the consumer device) on another device (the producer device). Once a filter is installed on the producer device, it is evaluated by that device on any new data; if it matches, that data is transmitted to the consumer device. Filters are application-defined; for example, they might check if GPS coordinates lie within some area, test photograph tags for equality with some string, or scan status updates for some keyword. For ease of exposition, we assume that there is only one application running on the devices; later, we will describe multiplexing mechanisms.

A device can attempt to install a Contrail filter on some other device by sending a filter installation request. The request only reaches the producer device if it includes the consumer device in a white-list. This is similar to users 'adding' each other on conventional social networks; for example, for Alice's phone to install a filter on Bob's phone, Bob would have to include Alice's phone (or, using a wildcard, any of her phones) on the white-list of his phone (or all of his phones). This allows Alice's device to request filter installations on his device.

The filter installation succeeds only if the producer device accepts the request. On the producer device, incoming filter installation requests are relayed to the application, which decides whether to accept them or not (possibly based on user input). Once a filter is installed, data matching it is allowed to travel back from the producer device to the consumer device.

Contrail's content filters give us privacy, since the filtering of data occurs on trusted edge devices, not central servers. They also give us upload and download efficiency; a device only uploads data matching a filter installed on it by another device. Conversely, it only downloads data matching a filter installed by it on another device.

3.2 Cloud Relays

Now we describe the mechanics of how messages (filter installation requests as well as data messages) travel from one device to another. Contrail consists of a client-side module that executes on each device, and a messaging layer that resides in the cloud. Each client-side module periodically initiates a TCP connection to the cloud-based

messaging layer via 3G (or a WiFi hotspot). In simple terms, a message sent by one device to another is first uploaded to the cloud via one device-to-cloud connection, and subsequently pulled by the recipient device via another such connection. These device-to-cloud interactions are the only network-level connections that occur in the system; for ease of exposition, we assume no out-of-band interactions between devices via channels such as Bluetooth.

Contrail's cloud layer consists of stateless application servers and a persistent storage tier. When devices connect to the cloud, they interact with one of these application servers; we call this the *proxy* for the device. If a device uploads a message meant for an offline recipient, its proxy stores the message in the storage tier. When the recipient device comes online, its proxy checks the storage tier for any messages meant for it and transfers them. On the other hand, if the recipient is online and connected to some other application server, the two proxies interact directly to transfer the message, without the storage tier in the critical path.

As described, the design of Contrail's cloud layer enables decoupled connectivity between devices. To provide multicast efficiency, the cloud layer allows senders to specify multiple recipients on a message. To provide semantic efficiency, it allows senders to set expiry times on messages, and to mark new messages as superseding older in-flight messages. When a message sent to an offline device expires before the device comes online, or is made obsolete by a new message, it is deleted from the cloud's storage tier.

Consequently, Contrail's combination of edge-based content filters and a cloud-based relaying layer allow it to offer all the properties of interest to us. Social network applications built using Contrail are privacy-aware, can work across devices decoupled in space and time, and are naturally efficient in terms of energy and bandwidth.

3.3 Reliability and Security in Contrail

To understand Contrail's reliability and security guarantees, we need to first state our assumptions about the cloud. Our reliability guarantee assumes the cloud does not lie about persistence; data stored in the cloud will not be lost. Our privacy guarantees do not make any assumptions about the cloud. In other words, a malicious cloud can interfere with Contrail's reliability and performance, but cannot view user data. Also, our design can be easily implemented on any existing cloud platform; consequently, if the cloud we use does not offer the desired reliability and performance, we can switch to one that does.

Contrail's cloud layer offers *reliable* communication — all messages are buffered on the sender device until its proxy acknowledges that it has stored the message persistently in the cloud's storage tier. This in-cloud copy of the message is deleted once the receiver device acknowledges receipt to its own proxy. This allows reliable communication between devices that are not simultaneously online. It is also an efficient reliability option when both devices are online, since it allows a fast sender to upload and disconnect once all messages have been persisted, without waiting for the receiver to finish downloading them.

Contrail's cloud layer also offers *secure* communication via a combination of well-known mechanisms. The flow of messages is tightly restricted by the white-lists

described previously; for social network applications, we expect these white-lists to correspond to friend lists, ensuring that messages only travel along the edges of the social graph. White-lists for users are stored in the cloud and proxies only relay filter installation requests between devices as permitted by these. Our assumption is that the cloud will honor these white-lists. As a result, devices cannot be spammed with filters by unknown rogue devices.

Privacy is ensured via device-to-device encryption: the cloud sees only encrypted payloads. Our strategy for encrypted communication is not novel; we use simple off-the-shelf techniques. We use public key encryption to exchange symmetric keys between devices, which are then used for encrypting all messages. For example, if Bob wants to send messages to Alice, he first sends her a message encrypted with her public key, so that only someone with her private key can decrypt it. That message contains a symmetric key which is used for all future messages (since symmetric encryption is faster and uses less energy on a smartphone than public key encryption).

For messages meant for multiple recipients, we encrypt the payload with a freshly generated symmetric key and then include this symmetric key as well in the message, encrypted separately with each recipient's public key. For example, if Alice is sending a photograph to Bob, Charlie and Donald, the outgoing message consists of the photograph encrypted with the new symmetric key, along with three versions of the symmetric key, encrypted with Bob's, Charlie's and Donald's public keys respectively. These per-message symmetric keys are cached and reused if many messages are sent to the same set of people.

In some applications, users may want to authenticate messages, ensuring that they did indeed originate from the apparent sender and were not tampered with. To handle this, Contrail computes a hash of the payload of each message and signs it with the sending user's private key.

Contrail does not provide privacy of inter-device relationships; through the white-lists, the cloud knows which devices (and which users) are talking to each other, even if it does not know what they are talking about. In the context of a social network, this amounts to the cloud knowing who your friends are. We think this is an acceptable trade-off: white-lists enable a spam-free system resistant to denial-of-service attacks (a critical property for resource-constrained devices), but require users to reveal their friend lists to the cloud.

4 The Contrail System

As described, Contrail consists of a client-side module that executes on each device and a messaging layer that runs in the cloud. In this section, we delve into the details of these two components.

4.1 Contrail on the Phone

Identifiers in Contrail: The basic unit of data in Contrail is an *item*. An item is defined as the combination of a payload and application-defined metadata. While metadata can be in any form, the default option in Contrail is to represent it as a hash-table of key-value pairs. For example, an item used by a photo-sharing application would store

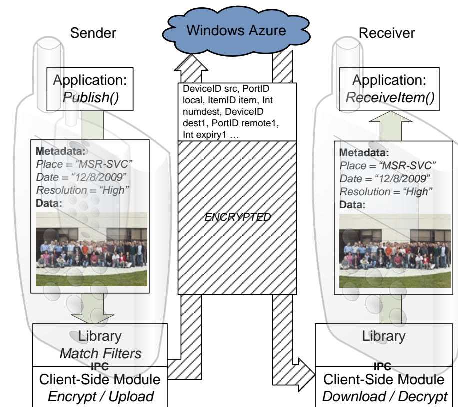


Fig. 1: The path taken by a data item through the Contrail stack.

the actual photograph in the payload, and attach metadata pairs to it such as ("date", "9/19/2010") and ("location", "San Francisco, CA"). Each item has an application-specified *ItemID*. The ItemID does not have to be unique across items generated by different applications; applications can set the same ItemID for different items (such as different versions of a document) to indicate that the later one makes the other obsolete.

A Contrail *end-point* is a pair consisting of a *DeviceID* and a *PortID*. The DeviceID is a globally unique identifier similar to a DNS name that is assigned to each client-side module. The PortID is a locally unique identifier used to multiplex traffic across different applications on the same device.

Contrail API: Contrail provides a library for applications running on the mobile device. The library offers to following API:

```
OpenPort(PortID local, Callback cb)
Publish(PortID local, Item itm, ItemID iid)
InstallFilter(PortID local, Filter f, DeviceID dest, PortID remote)
ReceiveItem(PortID local)
```

To use Contrail, an application creates an end-point by calling the $OpenPort$ function, specifying a PortID and a filter installation callback function. Once the application opens a port, other end-points – i.e., other instances of the application on different devices with open ports – can try to install filters on it, in order to receive data from it. These filters are delivered to the application via the filter installation callback. When a filter is received by the application, the application can either accept or reject it, by returning $true$ or $false$ from the callback, respectively.

To actually send data to other end-points, the application calls the $Publish$ function with an item as a parameter; see Figure 1. This results in all the installed filters on that port being evaluated on the item. The evaluation of the filters is performed by the Contrail library, within the application's own process. If the item is matched by one or more filters, it is transferred by the library to the shared module via IPC, along with a list of destinations corresponding to the end-points that installed the matching filters.

The shared module in turn constructs a data message with the item as the payload and uploads it to the cloud.

The basic format of a data message is shown in Figure 1. The header of the data message includes the source end-point information, the ItemID of the encapsulated item, the number of destination end-points, and routing information for each destination. The routing information for each destination consists of the (DeviceID, ItemID) pair as well as the expiry time of the item for that destination. Expiry times are destination-specific since we believe their utility to be driven by receivers that don't wish to receive stale data.

To install filters on other end-points, the application uses the $InstallFilter$ function. Once it has installed filters, the application can receive messages by calling the $ReceiveMessage$ function, which blocks for incoming items. The Contrail library also supports asynchronous interfaces for receiving messages; we omit these for brevity.

Push vs Pull: In addition to these interfaces, Contrail allows applications to tune the behavior of the shared module. For many applications, the shared module can simply keep a connection constantly open to the cloud; this is how push notifications work for the iPhone e-mail client, for example. For others, keeping a connection open constantly can be wasteful. If the application receives data at fixed, long intervals (a message every hour, for instance), or does not care about minimizing end-to-end latency, it may prefer the shared module to connect and disconnect periodically.

To support such applications, Contrail exposes two parameters. The *polling-interval* parameter, expressed in milliseconds, allows the application to regulate the frequency with which the shared module polls the cloud for new messages. The *idle-timeout* parameter specifies how long a connection is allowed to remain idle before it is torn down. Creating connections more frequently and keeping them open longer results in lower latencies for message delivery at the cost of energy and bandwidth. Since the shared module is shared by multiple applications, it chooses the lowest *polling-interval* and longest *idle-timeout* requested across all applications.

4.2 Contrail in the Cloud

The Contrail messaging layer is designed to run on any generic cloud provider; this flexibility allows for applications to switch between cloud providers when faced with faults and security issues. The only assumption Contrail makes about the cloud infrastructure it runs in, is that it provides an object store accessible through a put/get interface. Today most cloud providers (e.g., Microsoft Azure, Google AppEngine, Amazon AWS) do provide such a service.

Connecting with the Cloud: When a Contrail device connects to the cloud, it is directed to a randomly chosen application server (in Azure, these are called *worker roles*). We call this application server the proxy for that device during that connection. If this is the first time that the device has connected to the cloud, the proxy creates a message queue for the device in the storage tier. The name of this queue is simply the DeviceID of the connecting device. The purpose of the queue is to hold incoming data items and filters sent to the device from other Contrail end-points.

Upon accepting the connection from the device, the proxy updates a central map with the status of the device. This map has an entry for each device, including whether it's

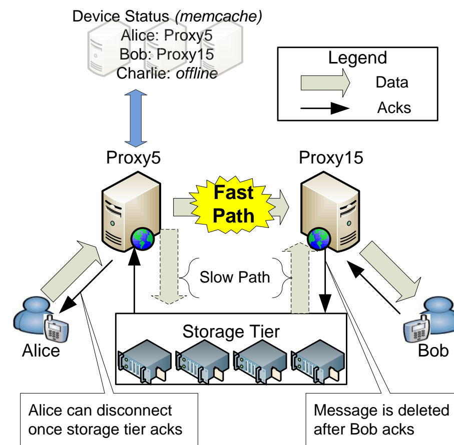


Fig. 2: Contrail implementation: data travels between proxies on a fast path for online devices and a slow path for reliability and offline devices.

currently online or offline, along with its current proxy if it's online. The map is stored in an in-memory storage service such as memcached; since Azure does not currently have such a service, we implemented our own over standard worker roles.

Relaying messages: If the connecting device has a message to send to another device, the proxy first checks the device map. If the receiver device is online and connected to the cloud, the proxy of the sending device opens a connection to the proxy of the target device and transfers over the message (we call this the *fast path*). The destination proxy then relays the message to the target device.

In parallel, it also writes the message to the queue of the target device in the storage tier (the *slow path*). This happens whether the target device is offline or online. When the target device is offline, writing it persistently allows the device to retrieve it at a later time; when it is online, it ensures that the message will be reliably delivered without requiring the sending device to stay online. Once the message is persisted in the storage tier, the proxy sends back an acknowledgment to the sending device. This lets the sending device delete the message from its buffers and go offline if required, with the guarantee that the message will be eventually delivered to the recipient.

Delivering messages: To receive messages from other devices via the fast path, the proxy listens for connections from other proxies. When the device first connects, the proxy also checks for incoming messages in the storage tier sent via the slow path while the device was offline. When a device successfully downloads a message, it sends back an acknowledgment to its proxy that triggers the deletion of the message from the storage tier. This ensures that messages are not stored forever in the storage tier.

Contrail ensures reliable delivery once the sender receives an acknowledgment, assuming that the cloud's storage tier does not suffer data loss and that the receiving

Alice

```
PortID localP = OpenPort("any_port", null);
SetPollingInterval(localP, 30);
SetIdleTimeout(localP, 0);
/* App-defined function to a filter
   matching locations within Mountain View */
Filter filter = create_mtnview_filter();
/* Install filter on remote port with
   id equals "location_update" */
InstallFilter(localP,filter,
             remotedevice,"location_port");
/* Alice receives location updates from
child's phone if he leaves Mountain View */
Item msg = ReceiveItem(localP);
if(msg!=null)
/*child has left Mountain View!*/
    freak_out();
```

Alice's Child

```
PortID localP = OpenPort("location_port", null);
while(true)
{
/* Alice's phone determines her location using GPS */
Location current_location = get_current_location();
Item msg = new Item();
AddMetadataToItem(msg, "location",current_location);
/* Publishing with same ItemID "mycurlocation"
   every time makes previous location
   updates obsolete */
Publish(localP, msg, "mycurlocation");
sleep(1 minute);
}
```

Fig. 3: Code for child-tracking application using the Contrail API.

device eventually connects to the cloud. The message is not removed from the sender's buffer until it is persisted on the cloud's storage tier, as indicated by the acknowledgment to the sender. It is not removed from the storage tier until it has been acknowledged by the receiver. Failures of the sender and receiver proxies or disconnections of the devices from the cloud can result in duplicate uploads and downloads of messages, but not loss.

5 Applications

Contrail makes it easy for developers to build social network applications that are decentralized yet efficient. We built several applications using Contrail, including location-tracking, photo-sharing, folder-sharing and chat. In this section, we first describe the design of the location-tracking application, and then elaborate on other possible applications.

5.1 The Location Notification Application

Here, we describe the details of the location notification application. The goal of this application is to notify users when the location of their friends satisfies some fixed condition; for example, as mentioned previously, a user Alice may want to know if her child is outside a threshold distance from his school, or if a friend she planned to meet at the mall has reached there. We will describe how Contrail allows such an application to be built in a manner that conserves bandwidth and power without sacrificing privacy, using filters as well as functionality such as item obsolescence and expiry times.

Figure 3 shows the pseudo-code for the location notification application. At a high level, this application uses filters in the following manner: Alice's device installs a filter on her child's device that includes the condition to be checked. The application running on her child's device periodically publishes his location as an item. Contrail on the child's device checks the installed filter on the location item, and pushes the item to the cloud if it matches. Importantly, each matching location update is published using the same ItemID ("mycurrentlocation" in the figure), making previous updates obsolete; as

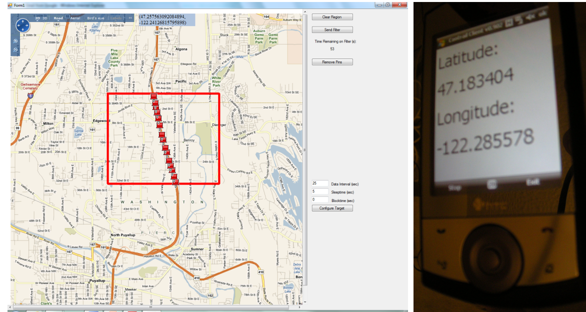


Fig. 4: Contrail application for selective location sharing.

a result, if Alice's device connects to the cloud after a prolonged disconnection, she receives only the latest location update.

In the pseudo-code, we omit the details of the filter. In our example, the filter is a bounds check on the location item's latitude and longitude. We represent the Mountain View area as a box with four corners, each of which has a latitude and longitude. Our filter is a conjunction of comparisons between the current coordinates and the bounds of the box. While our current implementation is restricted to such filters, Contrail can easily support more complex queries; for example, we could compute the distance of the current coordinates from a fixed point and check it against a threshold.

This application can also be used to notify users of their friends' location within a specific area. For example, Alice may want to know Bob's location, but he may choose to reveal it to her only when he's within the Microsoft campus. Figure 6 shows our location-tracking application in such a scenario. Alice installs a filter on Bob's phone asking for his location within a specific part of Seattle, which he accepts. On the right is Bob's phone generating location updates, and on the left is a computer where Alice is tracking Bob's location. As can be seen, Alice views Bob's location only when he is within the bounds specified.

5.2 Potential Contrail Applications

Real-Time Interactive: Applications such as chat, collaborative document editing, audio/video-conferencing and real-time games can be built easily using Contrail. Currently, such applications use either centralized servers (e.g., Google Wave) or – as in the case of Skype – leverage application-specific peer-to-peer networks on the wired Internet to tunnel traffic from and to 3G devices. To set up a chat session involving two or more people, for example, the application would simply have each participating device install filters on the other devices.

In addition to the obvious benefit of privacy, real-time applications benefit from Contrail's upload and multicast efficiency — a web-cam could stop uploading if nobody is watching it, or upload a stream just once for multiple viewers. Contrail's semantic efficiency properties are also useful to such applications; they can set expiry times

on outgoing items, ensuring that receivers do not get stale video frames, for example. Similarly, they can set up obsolescence relationships, ensuring that the receiver only receives the latest video frame or the latest version of a document.

Content Sharing: Contrail is useful for sharing bulk data items such as photographs or videos. Simple sharing is trivial to implement in Contrail; users can accept filters from their friends to enable sharing and then tag new media with the appropriate metadata. An application that wants to let users search their social network for existing content – as opposed to continuously receive new content – would simply use temporary filters with very short lifetimes and re-publish existing content through these filters. Interestingly, each query can also be propagated along the social graph at the application-level if recipients of the filter install it on their own friends, thus implementing P2P search on the social graph. Contrail's main benefit for content sharing applications is privacy, since the content metadata is not exposed to third parties.

Sensor Aggregation: Mobile devices can be viewed as sensors from which data can be aggregated, processed and queried (for example, phones being used to track traffic). Contrail is a great fit for sensor aggregation applications, since filters can be used to construct arbitrary aggregation topologies that save bandwidth and enforce privacy. For example, all Microsoft employees at the Silicon Valley campus could transmit their GPS locations to a local Microsoft server they trust, which then knows their individual locations; in turn, this server could transmit anonymized or aggregated data to a public server. This example would require the local Microsoft server to install filters on employee devices, and the public server to install a filter on the Microsoft server. As such, this example shows that a Contrail instance can include trusted machines in addition to edge devices.

Can Facebook be built using Contrail?

An interesting question for Contrail is whether it can support the same kinds of applications currently found on centralized services such as Facebook. We believe that most of these applications are easy to build on Contrail. For instance, message walls are simple to implement — Alice can install a catch-all filter on Bob's device that is evaluated on all new status updates. Facebook-style commentary threads for individual status updates seem difficult to achieve at first glance, since users can view comments made by each other on a common friend's wall even if they aren't each other's friends; for example, if Alice comments on Bob's status update, all of Bob's friends can view her comment.

In Contrail, communication between non-friends can be achieved by having users republish information at the level of the application. For example, to allow all of Bob's friends to view Alice's comment on his status update, consider a scheme where each user installs two filters on their friends: one to get status updates, and another to get comments. Now, Alice gets Bob's status update (along with all his other friends) via the status update filter; she then publishes a comment that only Bob gets via the comments filter. Bob then publishes the comment as a status update to his wall so that everybody else gets it.

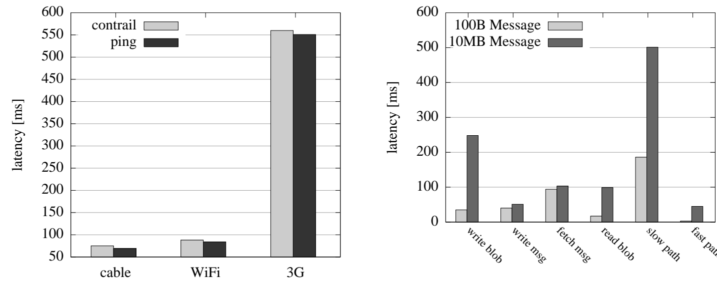


Fig. 5: a) Contrail's end-to-end latency between devices is close to network latency. b) Contrail's overhead in the cloud on the fast path (right-most bars) and the slow path (5 left-most bars).

6 Evaluation

We have evaluated Contrail using our prototype implementation. All our experiments are on a real implementation of Contrail running on Windows Azure. For clients, we use Windows Mobile phones connected to 3G networks, laptops tethered to these phones, and (for scaling experiments) instances in the Amazon EC2 cloud.

The first part of our evaluation focuses on the Contrail cloud-based messaging layer. We show that it provides good performance in terms of end-to-end latency and throughput. We also show that it is highly scalable. The second part of our evaluation focuses on the edge device; we show that Contrail's sender-side filters do not have a high computational overhead. We also evaluate the impact on the edge device of Contrail's tunable parameters.

6.1 End-to-End Latency

Figure 5a shows the end-to-end latency for an item to travel from one laptop to another via Contrail over different networks: when directly attached to a home cable network, when accessing that cable network over WiFi, and when tethered to a 3G phone. Both laptops are in the same physical location and the size of the message is 400 bytes. To understand what fraction of the observed latency was Contrail overhead, we also measured network-level ping latency from one of the devices to a ping server located near the Azure data center hosting the Contrail instance. The resulting graph shows that Contrail's end-to-end latency is limited almost entirely by latency on the network. Contrail itself adds no more than 5 to 10 ms of latency overhead.

Where is this extra latency used up? To find out, we instrumented the path of a Contrail message through the cloud using the Azure Diagnostics tracing framework. In Figure 5b, we show the measurement results for two different message payload sizes, of 100B and 10MB respectively. All the numbers shown are averages taken from 10 samples; we found the differences between each sample to be very small.

To understand Figure 5b, recall that messages in the Contrail cloud follow two separate paths: a fast path via a direct TCP connection between proxies when the

communicating devices are both online, and a slow path that involves persisting the message to disk. The right-most bar in Figure 5b shows the latency on the fast path. This number is crucial; it determines Contrail's latency overhead between two online devices. As can be observed, the latency overhead of a message on the fast path lies slightly below 50ms for a 10MB packet, and is around 4ms for a 100B message; this corresponds to the overhead observed in the previous end-to-end latency graph (Figure 5a).

The four left-most bars in Figure 5a show latency on the slow path. The 'write blob' stage refers to the time it takes the sender proxy to persist a message to the cloud's storage tier (in this case, Azure Blob Storage). The 'message write' stage refers to the time taken to update the queue of the offline recipient with a pointer to the message in the blob store.

6.2 Contrail Scalability

Next, we show that Contrail can scale to large numbers of client devices simply by adding more application servers (or Azure worker role instances) in the cloud. An important value proposition for cloud computing is the notion of elasticity. As load increases, additional computing resources can be harnessed to prevent degradation in the user experience. In the case of Azure, the unit of scaling is an instance, which corresponds roughly to a single virtual machine. We conducted an experiment where we varied the number of clients that were simultaneously connected to the cloud. The experiment was performed under three conditions: where message traffic was being handled by 1, 2 and 10 Azure instances. In this experiment, the clients ran on Amazon EC2 machines (in their US-West Coast facility). We used 100 small EC2 instances and ran 10 clients per instance, after verifying that running 10 clients per machine would not saturate the resources of one instance. Each EC2 client sent a message via Contrail – running in the Azure cloud – to itself every second. Figure 6a shows the average end-to-end message latency across users. We see that while a single instance can easily handle up to 200 simultaneous clients (average round-trip message latency of under 80ms), supporting 300 clients at the same time results in degraded performance (an average message latency of over 200 seconds). However, with 2 Azure instances, we can support up to 400 simultaneous clients (77ms for 300 clients and 87ms for 400 clients). With 500 clients, we start to notice performance degradation (over 200ms), while 600 simultaneous clients result in very high message latency. Finally, we observed that with 10 Azure instances, we were able to support at least 1000 simultaneous clients (78ms). These results indicate that the elastic nature of the cloud provides a scalable routing fabric for Contrail applications. Contrail is a trivially partitionable cloud application: as additional clients use Contrail, performance can be maintained by increasing the number of cloud instances.

6.3 Contrail Throughput

Apart from end-to-end latency on small items, we are also interested in knowing the data rate at which two Contrail clients can communicate. In this experiment we measured throughput of two different scenarios. *Online throughput* is the data rate at which two

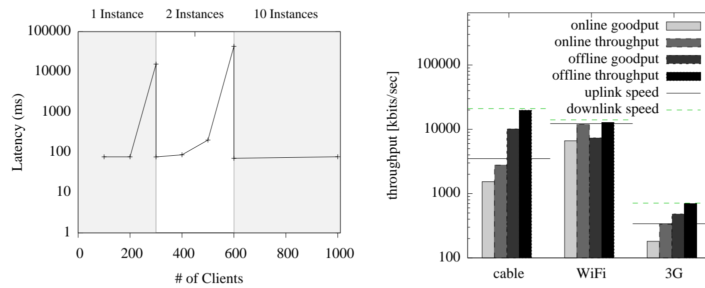


Fig. 6: a) Contrail can scale to thousands of clients simply by adding more server instances in the cloud. b) Throughput and Goodput between two Contrail devices.

devices can communicate if both devices are connected to the cloud simultaneously. *Offline throughput* is the data rate at which a device can receive data waiting for it in the Contrail cloud's persistent storage; this is data sent to the cloud while the receiver device was offline.

Figure 6b shows both online and offline throughput for the case where two laptops are attached to a) a cable network, b) a WiFi network, and c) a 3G network. The meanings of throughput and goodput in the figure are standard: one measures the total bytes transferred per second and includes the overhead of Contrail's headers and serialization mechanisms, while the other measures only the payload bytes transferred per second.

We can see in Figure 6b that Contrail's raw throughput reaches the network limit for all three network types. For online throughput, we are limited by the sender's uplink bandwidth, since the sending device is actively transferring data even as the receiver consumes it. For offline throughput, we are limited by the receiver's downlink bandwidth, since the cloud is able to send data at a fast enough rate.

The figure also shows that Contrail's goodput is much lower than its throughput. This is a limitation of our current implementation, which uses XML serialization of data messages (mainly because it is the only serialization mode natively supported on the Windows Mobile SDK). In the future, we expect to implement custom binary serialization to reduce the gap between goodput and throughput.

In Figure 7a, we evaluate the performance impact of item granularity. The Contrail implementation does not fragment items across multiple messages; each item is sent in a single Contrail message. As a result, applications must decide at what granularity to use items; for example, an application sharing a collection of photos could bundle them all into a single item, or send each photo individually as a separate item.

Accordingly, Figure 7a shows the transfer time of a) a 10MB file when both Contraildevices are attached to a cable network, b) a 10MB file if both sender and receiver are connected to a WiFi network, and c) a 100KB file for the case where both devices are using a 3G network. For all three configurations, smaller items result in lower transfer times up to a point; this is because the messaging infrastructure of Contrail behaves like a store-and-forward network, reading a message to completion before forwarding it to the receiver device. Consequently, the smaller the items, the

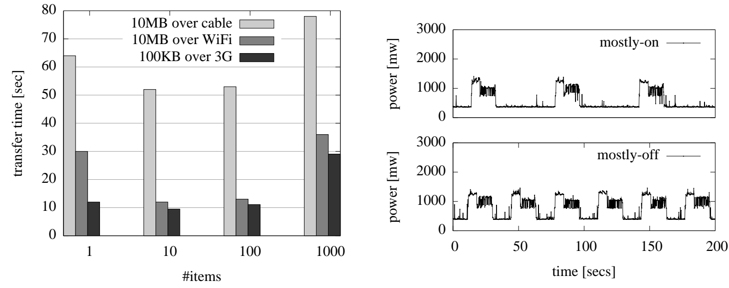


Fig. 7: a) Item granularity: smaller items result in better performance up to a point. b) Contrail uses less power when the connection is kept mostly on (top) as opposed to mostly off.

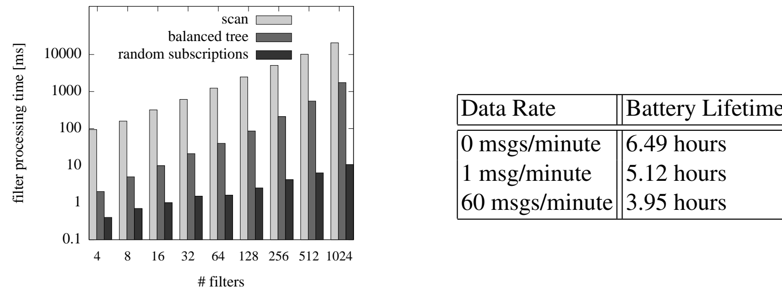
faster the receiver device starts downloading useful data. Beyond a point, however, smaller items give worse performance, since each message comes with its own headers.

6.4 Energy Consumption and Filtering

In the next set of experiments we study the effects of different options for a Contrail client to communicate with the cloud. As explained in Section 4, the Contrail API lets the application choose proper values for *polling-interval* (pi) and *idle-timeout* (it). Together, these parameters control how frequently the device opens a connection to the cloud and how long it keeps this connection open. Our initial hypothesis was that a longer value of *idle-timeout* would result in higher battery usage but lower message latencies, since the device would stay connected to the cloud for longer periods of time. We tested this hypothesis using a mobile phone running Windows Mobile 6.1. We intercepted the main power cycle between the battery and the phone and measured the instant power consumption using a dedicated power monitor [2].

Figure 7b shows power consumption of two different configurations, one where the polling interval is zero but the idle-timeout is 60 seconds (corresponding to tearing down and re-opening a connection immediately, once a minute), and another one where the polling interval is 30 seconds and the idle-timeout is 0 (establishing a connection every half-minute and tearing it down immediately). Essentially, the first case corresponds to having the connection open almost constantly (mostly-on), while the second case corresponds to creating short-lived connections periodically (mostly-off). The y-axis of the figure corresponds to the instant power consumption and the x-axis refers to time the experiment is running. We are not sending or receiving any data in this experiment.

The figure shows that for both configurations the mobile phone manages to enter a low power state: in the mostly-on case, this state occurs while the connection is on, whereas in the mostly-off case it occurs when the connection is off. This indicates that keeping a connection open does not come with a significant energy penalty. Also, keeping the connection open allows the phone to receive Contrail messages immediately, as opposed to the mostly-off case where it has to wait for a connection



Data Rate	Battery Lifetime
0 msgs/minute	6.49 hours
1 msg/minute	5.12 hours
60 msgs/minute	3.95 hours

Fig. 8: a) Filter execution time on a contrail mobile phone. b) Filtering data reduces messages and extends battery lifetime.

to be opened. This result suggests that – at least on this particular hardware – keeping a connection open is always the better strategy.

Despite this result, Contrail still supports the option to configure *idle-timeout* and *polling interval*. Our rationale is that different mobile devices may show different characteristics when it comes to energy consumption. In addition, certain applications may expect messages only at fixed intervals – for example, if a user is receiving updates from a 3G-enabled temperature sensor – or may prefer to only download the latest version of some data instead of all intermediate versions.

Next, we evaluate the feasibility of Contrail's sender-side filters. Evaluating filters on edge devices may seem infeasible when we consider that it is not uncommon for users on a social network website to have hundreds of friends (which might translate to an equivalent number of installed filters for each application). In this experiment, we study how fast Contrail can match all these filters when a new data item is generated on the mobile phone. We use a specific type of filter in our experiments: conjunctions of equality checks.

The matching time depends heavily on the matching algorithm and the actual set of filters that need to be matched. We study three cases. In the first case, we keep the filters in a list and iterate through the list every time a new item is generated. As can be observed from Figure 8a (label 'scan'), this approach very quickly results in a matching time of several seconds if the number of filters is large. In a second case we implemented a well known matching algorithm that uses a tree data structure to store the filters [4]. We generated filters in the worst possible manner which would cause the algorithm to visit every node in the tree while matching a data item. From Figure 8a (label 'balanced tree') it can be seen that the tree-based matching algorithm reduces the average matching time to a value below one second for 512 filters. In a third case, we used the same matching algorithm, but this time with randomly generated filters. The matching time in this case is just a few milliseconds, even for 1000 filters. This is because the algorithm mostly only traverses one path from the root of the tree to a leaf, where a leaf stores all the filters matching a particular data item.

Lastly, Table 8b present some measurements to show the energy consumption on a Contrail device at different data rates. Clearly, reducing messages improves battery

lifetime by a large amount. Thus, Contrail's filtering mechanisms can help applications minimize their battery consumption.

7 Related Work

Content-based Publish/Subscribe [8] is a well-known paradigm that uses content filters to route messages from publishers to subscribers. Contrail filters are similar to those used by Pub/Sub systems and offer similar benefits, such as decoupled transmission and bandwidth efficiency. However, Contrail uses filters for one-to-one and one-to-many communication between trusted, known devices. In contrast, Pub/Sub is aimed at scaling communication between anonymous sets of publishers and subscribers who do not know each other directly. Many of the results from the Pub/Sub literature on efficient filter matching apply to Contrail as well. Content filters are also to be found in replication frameworks [13].

Prior work by Ford et al. [9] has investigated naming and interconnection schemes for personal mobile devices. Haggle [18] is a network architecture for mobile devices that includes addressing and routing. MobiClique [11] explores opportunistic communication between devices on a social graph. All these projects are focused on settings where devices do not necessarily have ubiquitous 3G connectivity; as a result, many of the design decisions involve cooperation between proximal devices.

Contrail is an example of an Off-By-Default [5, 19] network architecture; devices have to install filters on each other to enable communication.

The design of the Contrail client-side module is related to work on efficient polling strategies for phones [10]. Contrail can also leverage hierarchical power management techniques [17, 15]. In addition, Contrail can be easily enhanced to support upload and download priorities for data [12]; for example, if a user wants to prioritize her tweets over her video uploads.

Privacy-aware architectures for mobile devices typically rely on trusted delegate machines for computing [14, 7]. Contrail is complementary to such techniques; it provides a networking layer that can be used to interconnect devices and delegates.

Privacy-preserving computing techniques already enable specific functionality such as keyword search [6, 16]. Contrail is complementary to these solutions; it is possible that applications will push simple functionality into the cloud using privacy-preserving techniques while retaining more general functionality on edge devices in the form of Contrail.

8 Conclusion

Building decentralized, privacy-aware social networks on smartphones is a daunting task; devices are often disconnected and have tight budgets for energy and bandwidth. Contrail is a communication platform that makes it easy for developers to build decentralized social network applications. Contrail enables efficient, privacy-aware applications that trigger communication between devices only when strictly necessary. It achieves this via two mechanisms: sender-side filters that reside on edge devices and cloud-based relays that provide reliable, secure communication between devices.

References

1. Diaspora. http://www.joindiaspora.com.
2. Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor.
3. Privacy-aware and highly-available osn profiles. In *6th International Workshop on Collaborative Peer-to-Peer Systems (COPS 2010)*.
4. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 1999. ACM.
5. H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default. In *Proc. 4th ACM Workshop on Hot Topics in Networks (Hotnets-IV)*. Citeseer, 2005.
6. D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. *Lecture notes in computer science*, pages 506–522, 2004.
7. R. Cáceres, L. Cox, H. Lim, A. Shakimov, and A. Varshavsky. Virtual individual servers as privacy-preserving proxies for mobile devices. In *Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, pages 37–42. ACM, 2009.
8. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
9. B. Ford, J. Strauss, C. Lesniewski, S. Rhea, F. Kaashoek, and R. Morris. Persistent Personal Names for Globally Connected Mobile Devices.
10. D. Li and M. Anand. Majab: improving resource management for web-based applications on mobile devices. In *MobiSys '09: Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 95–108, New York, NY, USA, 2009. ACM.
11. A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot. Mobiclique: middleware for mobile social networking. In *WOSN '09: Proceedings of the 2nd ACM workshop on Online social networks*, pages 49–54, New York, NY, USA, 2009. ACM.
12. A. Qureshi and J. V. Guttag. Horde: separating network striping policy from mechanism. In *MobiSys*, pages 121–134, 2005.
13. V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association.
14. N. Sadeh, J. Hong, L. Cranor, I. Fette, P. Kelley, M. Prabaker, and J. Rao. Understanding and capturing peoples privacy policies in a mobile social networking application. *Personal and Ubiquitous Computing*, 13(6):401–412, 2009.
15. E. Shih, P. Bahl, and M. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 160–171. ACM New York, NY, USA, 2002.
16. D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, 2000. S&P 2000. Proceedings*, 2000.
17. J. Sorber, N. Banerjee, M. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM New York, NY, USA, 2005.
18. J. Su, J. Scott, P. Hui, J. Crowcroft, E. De Lara, C. Diot, A. Goel, M. Lim, and E. Upton. Haggle: Seamless networking for mobile applications. *Lecture Notes in Computer Science*, 2007.
19. H. Zhang, B. DeCleene, J. Kurose, and D. Towsley. Bootstrapping Deny-By-Default Access Control For Mobile Ad-Hoc Networks. In *IEEE Military Communications Conference (MILCOM) 2008, San Diego, November 17-19, 2008*.