# The Cloud is the Router: Enabling Bandwidth-Efficient and Privacy-Aware Mobile Applications with Contrail

Microsoft Technical Report
MSR-TR-2010-89

Patrick Stuedi*, Iqbal Mohomed*, Mahesh Balakrishnan*
Z. Morley Mao†, Venugopalan Ramasubramanian*, Ted Wobber*

*Microsoft Research Silicon Valley, Mountain View, CA
†University of Michigan, Ann Arbor, MI

June 2010

## ABSTRACT

Collaborative applications running on 3G devices often rely on cloud-based servers for computation and storage. A peer-to-peer approach to building these applications can provide benefits such as enhanced privacy and bandwidth efficiency. We propose Contrail, an asynchronous network architecture that uses the cloud to relay messages between 3G devices. Contrail employs selective receiver-specific filters at sending devices to ensure that only relevant data consumes precious bandwidth. Our framework offers pull-based communications primitives suitable for mobile devices that are often either inactive or subject to poor network connectivity. Contrail enables robust mobile applications without making assumptions about the security of individual cloud providers. We have implemented Contrail within Windows Azure and demonstrate several sample applications executing across Windows Mobile devices.

## 1. INTRODUCTION

Smartphones connected to ubiquitous 3G networks are enabling new and diverse applications. One class of applications consists of *data sharing* services, where data generated by a device – location updates, photos, video clips and real-time feeds – is shared with other devices based on specific policies. These policies are usually based on the generated content; for example, a tourist backpacking through Europe with a smartphone may want to broadcast her location to friends who live nearby, receive tweets that mention her name, share videos that her family wants to see, or even receive a feed from her home's security camera if it detects movement.

Currently, such services are structured as client-server applications; all data generated by a device is uploaded via 3G to a central server which provides it selectively to other devices. The centralized approach allows data to be uploaded just once by a sender for multiple recipients, without requiring any of them to be online at the time. As an additional benefit, client-server applications do not require inbound connections into devices, allowing cell ISPs to provide greater security by blocking such connections.

However, the centralized approach can be inefficient if devices generate substantial data that no other device wishes to receive. In the example of the location-sharing application, the device will continually upload its location to the cloud even if nobody lives nearby, wasting network resources and battery life in the process. A second problem with the centralized approach relates to privacy: users expose all data to application servers and the third-party cloud providers hosting them, and potentially even to other customers of the same cloud [11]. Since the centralized server has to apply sharing policies on the data, simple encryption is not a viable solution. Privacy-preserving computing on encrypted data [3, 15] holds great promise, but is problem-specific and not yet widely deployed.

In this paper, we introduce Contrail, a new communication architecture for mobile data-sharing applications that eliminates the drawbacks of the client-server approach while retaining its positive attributes. Specifically, Contrail's goal is to enable applications that have the following properties:

- Efficiency: Data is uploaded by a device at most once, and only if it is explicitly requested by some other device.
- Non-Intrusiveness: Data arrives at a device only if it is explicitly requested from some other device.
- Privacy: Data sent by one device to another is not viewed by any other entity.

- Decoupling: Data can be sent from one device to another even if they are not simultaneously online.

Contrail provides these properties via two primary mechanisms. First, all communication is driven by the abstraction of sender-based content *filters*, which allow devices to explicitly request information from other devices. A device that installs a filter on a remote device receives data from it that matches the filter. Second, Contrail uses the cloud as a scalable message relay; devices periodically 'dial' into this infrastructure via 3G connections and use it to transfer encrypted filters and data to and from other devices.

Contrail's combination of sender-based filters and cloud relays achieves all four desired properties. First, data is uploaded once to the cloud if one or more filters on the sender installed by other devices matches it; the cloud caches this data and subsequently relays it to all recipients. Second, devices connect to the cloud as clients and pull data from it, not exposing themselves to inbound connections. Third, data and filters are encrypted end-to-end between devices, offering the cloud relays no visibility into either. Last, the cloud enables decoupled communication by buffering data and filters for offline recipients until they come online. In addition, Contrail's strategy of using the cloud purely for communication – as opposed to persistent storage or computation – enables robust applications that can easily fail-over to different cloud providers, or even resort to non-Contrail communication pathways.

While we center Contrail's design on 3G devices such as smartphones or netbooks, it has broad applicability in a world of diverse devices and connectivity options. Contrail can be a communication option for devices behind public WiFi hotspots, which often suffer from the same connectivity and bandwidth restrictions as 3G clients. Contrail also naturally supports delegate-based systems [5], where devices interact with trusted machines such as personal desktops behind home routers; these machines would simply be first-class Contrail end-hosts that use filters to pull data from smaller devices.

The contributions of this paper are the following:

- We describe a communication architecture for mobile data-sharing applications that offers important properties such as efficiency, non-intrusiveness, data privacy, and sender-receiver decoupling.
- We discuss the design of this architecture on commodity cloud platforms and mobile devices.
- We implement the Contrail cloud component on Windows Azure and its client component on Windows Mobile.
- We evaluate the latency, throughput, power and scalability characteristics of our Contrail implementation.

The remainder of the paper is organized as follows. Section 2 talks about our model of device trust and network characteristics. Section 3 describes the Contrail architecture, and Section 4 looks at the design of the individual components within it. Section 5 talks about the different applications that Contrail enables. Section 6 describes our implementation and Section 7 evaluates it. Section 8 provides related work, and Section 9 concludes.

## 2. THE Contrail MODEL

### 2.1 Trust Assumptions

Contrail assumes that devices trust each other. For example, when one device sends data to another device, it trusts the recipient to not redistribute that data. Similarly, we assume that devices do not impersonate each other. We expect this trust to result from social contracts; for example, if the owner of the sending device knows the owner of the receiving device. In concrete terms, we assume the existence of a social graph where the existence of a link between two users implies that they trust each other completely.

We assume that it is unsafe to reveal data to a cloud provider in unencrypted form. We do assume, however, that cloud providers are reliable and do not lose inflight data. We assume that individual cloud providers can become unavailable due to new 'failure modes', such as quota overages, payment delays, increased rates and changing business alliances. We assume that at least one operational and feasible cloud provider exists at any given point of time.

### 2.2 Network Model

Most deployed 3G networks are designed for shortlived client-server interactions. As a result, IP addresses assigned to end-hosts can change frequently and routing state is short-lived. Most networks do not support inbound network connections, partly due to the threat of phone-based malware but also to avoid expensive peer-to-peer traffic. Additionally, upload capacity is usually more constrained than download bandwidth.

We expect 3G devices to be frequently offline due to limited battery and network coverage. Consequently, NAT traversal services designed to tunnel IP packets are of limited use with 3G end-hosts, since two devices will rarely be online at the same time.

## 3. THE Contrail SYSTEM

Contrail consists of two primary subsystems: 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/IP connection to the cloud-based messaging layer via 3G (or a WiFi hotspot). Contrail's basic operation can be described in simple

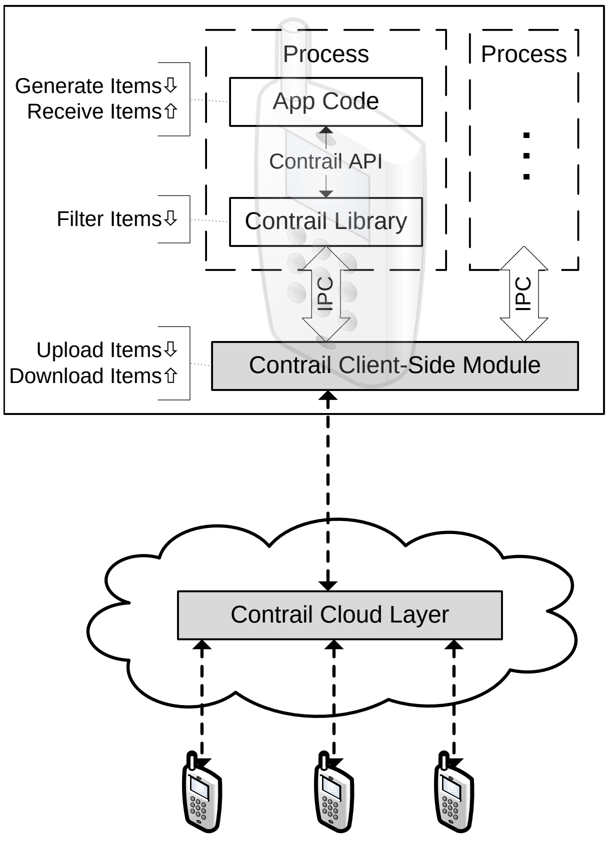

**Figure 1: Contrail Architecture**

terms: data sent from one device to another device 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.

Applications on devices use Contrail by linking to a library that exposes several functions for sending and receiving data (see Figure 1). Internally, this library uses IPC to communicate with the client-side module running on the device. A single client-side module runs on each device in a separate process, and is shared by multiple applications.

In this section, we describe the overall architecture of Contrail, along with the abstractions and API it exposes to applications. For now, we treat the client-side module and cloud-based messaging layer as black boxes; in the next section, we will describe their internal design in detail.

## 3.1 Abstractions

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 the actual photograph in the payload, and attach metadata pairs to it such as (“date”, “12/4/2009”) 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; later, we will describe the semantics of matching ItemIDs on items generated by a single application.

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.

An end-point installs *filters* on other end-points to receive items from them. In the absence of installed filters, no end-point in Contrail can send data to any other end-point. In its most general form, a filter is simply a function that accepts an item as input and returns true or false. In this paper, we restrict ourselves to filters that are simple conjunctions of comparison tests on key-value metadata: for example, a filter used by the photo-sharing application could return true on an item if the value for “location” matches “San Francisco, CA”, or if the value for “size” is greater than 1024. Contrail filters can be extended to be arbitrarily expressive queries on item data or metadata. In the limit, they can consist of arbitrary code, though this requires additional security mechanisms that are beyond the scope of this paper.

## 3.2 The Contrail API

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 applications 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 delivered, 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. This results in all the installed filters 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 client-side module via IPC, along with a list of destinations, corresponding to the end-points that installed

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

**Figure 2: Contrail API**

the matching filters.

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.

## 3.3 Security

In Contrail, message flow between devices originates with the installation of filters that specify, at the sender, what data is desired and by which receivers. However, secure communication between devices first requires the establishment of trusted channels. Contrail channels are uni-directional and offer both integrity and confidentiality. Although all channels transit the cloud by design, the cloud can neither tamper with nor read the contents of messages. Similarly, the cloud cannot tamper with message headers without risking detection. Channel end-points are identified by {DeviceID, PortID} pairs. Channels have exactly one sender and at least one receiver: they are either point-to-point or one-to-many. The latter permits certain efficiencies by allowing messages to be encrypted once for all receivers, but it also permits any of the several receivers to impersonate the true sender. This impersonation threat is not important to us because we assume that client devices are trusted.

We expect devices to hold *a priori* a white-list of DeviceIDs with which communication is anticipated. In many scenarios, all applications across a device can share a single white-list. While it can be occasionally desirable for applications to maintain individual sets of correspondents, we will assume a single per-device white-list here to facilitate discussion.

Each device stores a public key pair that it uses to receive and distribute the symmetric keys that underlie secure channels. We do not mandate a specific mechanism for mapping between DeviceIDs and public keys, however we assume that one exists. Existing tools for performing this mapping, such as public key infrastructures, are well-known in the literature. Alternatively, we can rely on manual key distribution or stipulate that DeviceIDs be derived from associated public keys.

Prior to first communication between devices, the sending device must construct a channel by a sending a *key distribution* message to its intended recipients. This message contains the source and destination PortID as well as newly-generated symmetric keys to be used for integrity and confidentiality on the channel; it is encrypted with the public key of each intended recipient, and signed by the sender. Upon receiving a key distribution message, each receiver must ascertain that the originator of the message is acceptable to the application controlling the local destination port. Channel keys must, as usual, be of limited duration and are therefore refreshed by periodic key distributions.

We expect Contrail white-lists to express the social graph of devices (and therefore users) in the system. In that sense, a device can 'friend' another device if they mutually add each other's identifiers to their respective white-lists. We assume that the social graph is undirected; i.e., trust relations are symmetric. Since the white-lists represent the social graph, *all communication between devices in Contrail travels along links in the social graph.* In other words, a device in Contrail will receive data or metadata from some other device only if that device is its friend. Additionally, a device's white-list, and in particular the set of devices from which it will accept filters, can be made available to the cloud. This can help prevent an unknown rogue device from spamming another device with filters.

We would like to reiterate that a Contrail device receives no messages by default. In order to receive data items and keys from other devices, it must first install filters on them. To receive filter installation requests, it must include devices in its filter white-list that are allowed to install filters on it.

## 3.4 Extended Functionality

Contrail's design provides the flexibility to add network functionality that can be useful for mobile applications.

### *3.4.1 Data Caching*

Contrail extensively caches uploaded items, retrieving them when required using a concatenation of the DeviceID, the PortID and the ItemID. Consequently, items generated by an end-point are uploaded only once to the cloud the first time they match a filter; on subsequent filter matches of the same item, the client-side module directs the cloud to retrieve the item from its cache and route it to the receiver, without requiring a new upload from the sender. The items are cached in encrypted form; the cloud can satisfy a send using a cached copy only if the recipient has the decryption key.

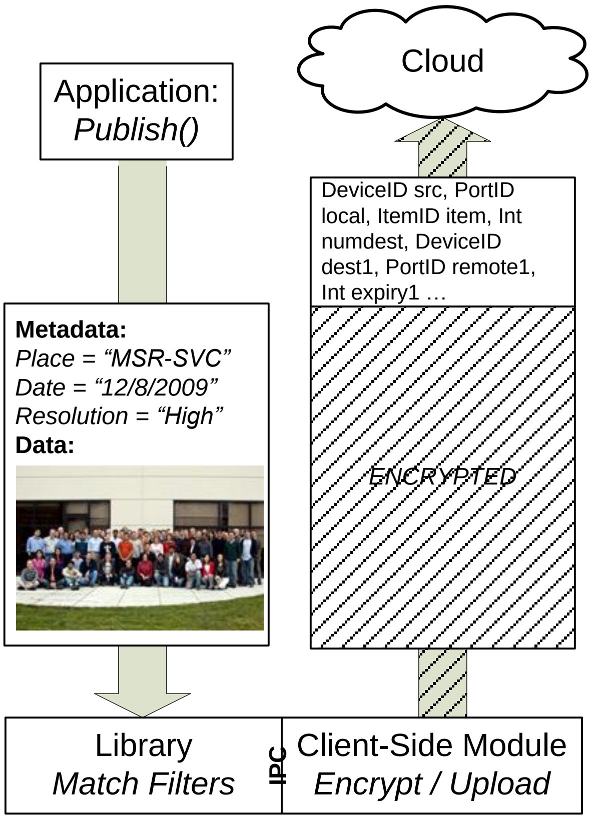

**Figure 3: Path of an Item through the stack**

#### *3.4.2 Data Obsolescence and Expiration*

Contrail allows an application to define obsolescence relationships between items by assigning them the same ItemID. For example, a device might send multiple Contrail items to an offline receiver that represent different versions of the same document; when the receiver comes online and connects to the cloud, it receives only the latest version of the document as opposed to each successive version.

In similar vein, Contrail also allows applications to define data expiration times. Receivers can specify expiration times on the filters they install on senders, passing in an extra optional parameter to the *InstallFilter* call. This ensures that a receiver connecting to the cloud after a long hiatus is not flooded with old messages.

#### *3.4.3 Publisher Groups*

In many cases, a user may be interested in receiving data from a subset of her friends that satisfy some criteria; for example, users who are in a particular location. An obvious way to achieve this functionality is for the user to install an appropriate filter on each of her friends' devices; however, this approach may become cumbersome if she has hundreds of friends.

To allow for more efficient filter installation, Contrail allows devices to join *groups*. A group is represented by a GroupID. We overload the *InstallFilter* call so that devices can install filters on a GroupID. When a device installs a filter on a GroupID, that filter is routed to all its friends that have joined the group. Subsequently, the installing device receives messages generated by its friends in the group.

In the example of the photo-sharing application, all Microsoft employees could join a group called "Microsoft". Now, any user can install a filter on the group "Microsoft"; the filter will be installed on all her friends who belong to the group, and pull in any matching photos from them. Alternatively, the membership of the group could be dynamically determined; i.e., a group called "Mountain View" could consist of all devices currently in Mountain View. Devices would individually join or leave this group based on their current location.

Note that our definition of a group is different from the usage of the term in traditional networking: a Contrail group defines a set of devices that *send* data, whereas an IP Multicast group is a set of nodes that *receive* data together. Also, Contrail groups retain the property that all traffic travels between friends on a social graph. Devices can use groups only to receive data from their own friends. We believe that more general functionality can be easily implemented at the application level; i.e., if a user wants to retrieve photos from friends of friends who currently belong to the "Mountain View" group, her friends could duplicate her filter and install it themselves on the group, forwarding any results back to her.

## 3.5 Reliability and Flow Control

By default, Contrail does not provide end-to-end reliability; it uses reliable TCP/IP connections to transfer data to and from the cloud, and assumes that the cloud will not drop any in-flight data. Applications that require stronger reliability guarantees can implement their own reliability mechanisms, installing appropriate filters that catch acknowledgments and retransmissions.

For example, consider a photo-sharing application with two devices, $A$ and $B$, where $B$ has installed a filter on $A$ requesting specific photos, say those whose "location" tag equals "Mountain View, CA". The scenario we are concerned with is when $A$ sends $B$ a photo which gets lost in transit.

One option is for device $A$ to install a filter back on $B$ to catch acknowledgments or negative acknowledgments; $B$ can then create and publish ACK or NACK items that are routed back to $A$. Device $B$ can then install additional filters (or modify existing ones) to catch retransmissions meant for it; for example, by matching items whose "retransmit" property equals $B$.

An issue with delegating reliability to the application is that it has no means of detecting data loss. A sender cannot distinguish between the case where data is lost and the case where data is merely delayed in the cloud due to the receiver being offline. As a result, the sender could retransmit data even if it hasn't been lost. However, Contrail's caching and obsolescence mechanisms mitigate this problem; each retransmission of an item is satisfied by the cloud's cache, and the receiver only obtains the latest retransmission since all prior transmissions are made obsolete by it. As a result, a retransmission operation simply involves a small notification by the sender to the cloud.

Applications that want increased reliability can also utilize *multi-homing.* A device that is about to go offline for a long period (due to a user forgetting her charger on a trip, for example) will not be online for end-to-end retransmissions if in-flight data is lost. In such cases, the device can upload an item to multiple Contrail instances running on different cloud providers. The mechanics of multi-homing are simple: the application deals with multiple Contrail client-side modules as it would with different conventional network interfaces.

Contrail's flow control mechanism is simple; if the cloud has too many in-flight items backed up for an offline receiver, it simply returns a failure code to the sending client module, which in turn relays it back to the application as a return value to the *Publish* call. The application can then try to publish the data again after some interval. We expect applications to use expiry and obsolescence functionality effectively to eliminate large backlogs in the cloud.

## 4. Contrail DESIGN

Thus far, we have treated the client-side module and the cloud layer as black boxes. In this section, we look at the internal design of both these subsystems.

### 4.1 Contrail Client Module

The Contrail client-side module consists of two separate components: a *pull* component that polls the cloud periodically for inbound messages and a *push* component responsible for uploading outgoing messages. Each of these components connects to the cloud periodically, 'sleeping' in between connections by not using the wireless radio. Applications can individually set the frequency with which these connections occur.

The pull component exposes a *polling-interval* parameter to applications, expressed in milliseconds; setting this allows the application to regulate polling frequency. More frequent polling results in lower latencies for message delivery but uses up power and bandwidth. Since the Contrail module is shared by multiple applications, it chooses the lowest polling interval requested across all applications. Once a connection has been made, it persists as long as there is data left to be downloaded. Once the cloud reports an empty queue of incoming items to it, the pull component goes back to sleep after waiting for a fixed time interval. The time interval it waits for on an empty queue is a second parameter, called the *idle-timeout.*

Similarly, the push component exposes a *batch-size* parameter; this corresponds to the maximum number of outbound items per application that are allowed to queue up before the Contrail module connects to the cloud. As with the pull component's *polling-interval* parameter, this value can be set independently by each application; the module uploads all outbound items across applications when the *batch-size* value for any one of the applications is exceeded. A default time-out value between connections ensures that items are eventually transmitted even if the *batch-size* is never exceeded by any application.

By default, the push and pull components operate independently, driven by their respective parameters. A simple optimization involves combining their operation, so that the module both pushes and pulls data to the cloud during a single connection. More sophisticated optimizations that leverage the properties of the wireless hardware are possible; for example, utilizing knowledge of radio power-down policies to minimize power usage [1, 12].

When the push or pull components connect the cloud, they exchange *messages* with it. There are three types of messages: data messages encapsulating items, filter installation messages, and key distribution messages.

The basic format of a data message is shown in Figure 3. 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.

When the client-side module receives an item via IPC from the library running in the application process, it encapsulates the item within a data message with the appropriate header and adds it to an outgoing queue. The push component then connects to the cloud and uploads the message, either immediately or after some interval if the *batch-size* parameter is not equal to 1. We will shortly describe the data message's path in the cloud.

Similarly, the pull component receives data messages from the cloud, from which it extracts the encapsulated item and relays it via IPC to the library. In this case, the data message typically has only one destination in the header, i.e., the DeviceID and PortID of the current device.

### 4.2 Contrail Design 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. Consequently, it is important to understand the common features – and restrictions – of emerging cloud platforms.

### *4.2.1 What makes a Cloud?*

Cloud platforms such as Microsoft Azure and Google AppEngine mandate a three-tier architecture on developers (Amazons EC2 does not enforce separation between the web and the compute tiers, but does have explicitly separate storage services). The first tier consists of front-facing webservers that accept and manage connections from clients. Application code executes in a second tier of stateless compute nodes, with all persistent data stored within a separate storage tier. The stateless nature of the compute tier allows such platforms to easily scale out code written by inexperienced developers; each incoming request can be load-balanced to any compute node, allowing throughput to be ramped up simply by adding more machines to the system. The storage tier is separately scaled out using more complex protocols that partition and replicate data in order to provide fault-tolerant and scalable storage.

Current cloud platforms provide multiple storage tiers with different interfaces, performance and persistence levels. Common to all three major platforms are queueing services (e.g., Azures Queue Storage and Amazons Simple Queue Service) and object stores with put/get interfaces. Each storage tier exposes a name-space to compute nodes that allows them to identify units of storage. Storage tiers can be persistent (e.g., Amazon's Elastic Block Storage) or volatile (e.g., AppEngine's *memcache*).

In addition, cloud platforms are invariably geo-distributed, allowing services to be replicated or partitioned across multiple geographically distant data centers. Clients attempting to access geo-distributed cloud services are transparently directed to their closest data center through region-specific DNS entries. Within the cloud infrastructure, the webserver and compute node handling a particular request are usually collocated in the same data center to ensure low latency; however, the state modified or accessed by the request can exist at a storage node in a remote data center. In the simple example of a cloud-based email service, the mailbox data of a European user resides in a storage node in Europe; however, when she visits the US, her requests are directed to a webserver and compute nodes in a US data center, which subsequently access the storage node remotely.

To summarize, contemporary cloud platforms exhibit three properties of interest to us:

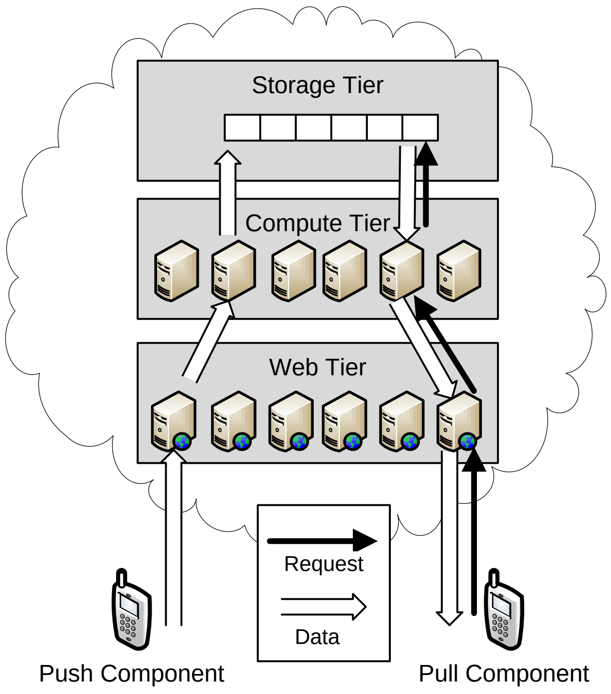

**Figure 4: Contrail's cloud design.**

- Applications must be built according to a three-tier architecture with a web tier, a compute tier and a storage tier.
- The compute tier is stateless and any request can execute on any compute node.
- The storage tier is distributed across multiple data centers, and any compute node can access any unit of storage, irrespective of location.

### *4.2.2 Contrail Cloud Design*

Current cloud providers offer different development models for developers, ranging from general compute nodes such as Amazon's EC2 to constrained three-tier platforms such as Google's AppEngine. Since Contrail is designed as a cloud service capable of executing on any three-tier platform, it can be deployed on more general cloud platforms as well. In Contrail, the web tier is responsible for accepting connections from devices. As described earlier, the client module on each device periodically initiates a connection – for pushing data or pulling it – to a webserver, which we will call the gateway for that device during that connection. If this is the first time that the device is connecting to the cloud infrastructure, the gateway executes a 'join' operation on a compute node, which then creates a queue in the storage tier for that device. 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 devices.

When the gateway receives a data message from the client module on the device, it executes a 'send' operation at a compute node with the message as a parameter. For each destination specified on the data message, the compute node locates the queue of the destination device (using its DeviceID as the queue name) and appends the message to it. Before it appends the message to each destination device's queue, it changes the header so that it contains just that device as the sole destination.

When a device initiates a pull connection into the cloud, the gateway executes a 'retrieve' operation at a compute node with the DeviceID of the connecting device as a parameter, and waits for the response. The compute node locates the queue corresponding to that DeviceID and retrieves the messages in it. It checks each message to eliminate those that have passed their expiry time, or have been made obsolete by later items in the queue. The remaining messages are relayed down to the pulling device.

A device initiating a connection to the cloud for pulling items is always directed to the data center closest to it. The gateway and compute node reside in that data center, while the device's queue can potentially be in a different data center (from the third property of cloud platforms). Such a scenario can arise due to user movement; for example, if the user resides on the US west coast but is currently traveling in Europe. The compute node that empties the device's queue on a pull request can potentially relocate the queue to the local data center (from the US to Europe, in the example), based on policies aimed at predicting the user's home location.

Similarly, a device initiating a connection to the cloud for pushing items is also directed to the closest data center. The compute node in that data center updates the queues of destination devices directly, even if those queues reside in remote data centers. For example, if a device in Mountain View, CA pushed an item to one in Ann Arbor, MI, the item would first travel to a gateway and then a compute node in a local data center in california, which would then directly update the destination device's queue in a mid-west data center.

## 5. APPLICATIONS

Contrail makes it easy for developers to build data sharing applications for mobile devices that provide the four properties outlined in Section 1: efficiency, non-intrusiveness, privacy and decoupling. These applications can be built by specifying appropriate Contrail filters and parameters; Table 1 shows some examples. In this section, we first elaborate on these applications, and then describe the design of a specific Contrail application.

### 5.1 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. For real-time audio or video, applications can set the *polling-interval* parameter to 0 and set the *idle-time* parameter to a non-zero value, ensuring that outgoing items are immediately dispatched to the cloud; later, in the Section 7, we will evaluate the effectiveness of this approach. In addition, applications 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, for example.

**Content Sharing/Searching:** Contrail is useful for sharing bulk data across users: for example, photographs or videos. We have already described the operation of a photo-sharing application designed using Contrail. An application that wants to let users search their social network for content would simply have each user install temporary filters with very short lifetimes on each of their friends. Interestingly, each query can be propagated at the application-level by recipients of the filter installing it on their own friends, thus implementing P2P search on the social graph.

**Location Notification/Sharing:** Contrail enables privacy-aware location-based applications; for example, an application that requires parents to be notified when their children leave a particular GPS range. This is achieved simply by having the parents' devices install filters on their children's phones. Another example involves an application that wants to alert users when their friends are at some fixed location; each user's phone can install filters on her friends' phones with the appropriate location information. Extending this application to one where users are notified when their friends are near their current location is also easy; the user's phone has to periodically replace the filters on her friends' phones with updated ones.

**Sensor Aggregation:** 3G 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,

| Application | Filter | *polling-interval* | *idle-timeout* |
|---|---|---|---|
| Chat | destination==“Bob” | 5 sec | 0 sec |
| Photo-Sharing | resolution==“high” AND tag == “family” | 60 sec | 0 sec |
| Location Notification | location==“Mountain View, CA” | 600 sec | 0 sec |
| Video streaming | destination==“Bob” | 0 sec | 10 sec |

**Table 1: Examples of filter and Contrail parameters for different applications**

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.

## 5.2 The Location Notification Application

We built several applications using Contrail. In this section, we describe the details of the location notification application mentioned above. 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 (call her 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 5 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 checks the installed filter on the location item, and pushes it 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 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 that of the four corners. 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.

While expiration does not come into play in the specific case of a parent tracking her child, it does play a major role in this application. For example, if Alice installed a similar filter on her friends to facilitate meetups, she would specify an expiry time of a day for any location updates that her friends send her, so that she is not flooded with old location updates from friends that have become irrelevant.

## 6. IMPLEMENTATION

We have implemented a prototype of Contrail with the client-side module running on Windows Mobile and the Contrail cloud layer running inside Windows Azure. Our prototype supports filters that are conjunctions of comparison operators on metadata properties. It includes end-to-end encryption, though we did not implement key distribution messages. Also, we did not implement the *batch-size* knob; it is fixed to a value of 1, which means that data is always pushed out immediately to the cloud.

The client-side module is implemented as a Windows Mobile background service. At its core it is listening on a TCP server socket for *OpenPort* requests relayed by the Contrail library from local applications. Upon receiving an *OpenPort* request it launches a thread to process messages from and to that particular port. The client-module stores a hashtable with all open ports and their corresponding open connection. Items received from applications are encapsulated within data messages and enqueued inside the module. Messages received from the Contrail cloud layer are forwarded to the right local application connection using the PortID information in the message header. The client-side module maintains two separate threads to communicate with the contrail cloud layer, corresponding to the push and pull components described in section 4.1.

Windows Azure cloud services are composed of a set web roles and a set of worker roles. A web role runs inside the IIS webserver and a worker role is an arbitrary windows program. Each web or worker role is running as a separate virtual machine inside one of the Azure datacenters. Virtual machines might run on separate physical machines or share one physical machine.

The Contrail cloud layer consists of many worker role instances, where each worker role is hosting two separate TCP servers: an incoming TCP server receiving messages from mobile devices and an outgoing TCP server transmitting messages to mobile devices. Additionally, each worker role hosts a message process-

**Alice**

```
PortID localPort = OpenPort("any_port", null);
SetPollingInterval(localPort, 30);
SetIdleTimeout(localPort, 0);
/* App-defined function that creates filter
   to match locations within Mountain View */
Filter momfilter = create_mtnview_filter();
/* Install filter on the "location_update" port
   on child's remote device */
InstallFilter(localPort,momfilter,
              remotedevice,"location_port");
/* Alice receives location updates from child's
   phone if he leaves the bounds of Mountain View */
Item msg = ReceiveItem(localPort);
if(msg!=null)
/*child has wandered out of Mountain View!*/
    freak_out();
```

**Alice's Child**

```
PortID localPort = 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(localPort, msg, "mycurlocation");
sleep(1 minute);
}
```

**Figure 5: Location notification application using the Contrail API**

ing thread which processes newly arrived messages and passes them to persistent storage. Together, those three components (TCP servers, message processing and persistent storage) implement the three tiers (web tier, compute tier, storage tier) described in Section 4.2.

A common sequence of steps for a data message traversing the cloud layer is as follows. The message is received by the incoming TCP server and placed into an in-memory queue. The message processing thread dequeues the message from the queue and stores it persistently using the Azure blob service. Blobs are one of several storage services provided by Azure, apart from queues and tables. Blobs can contain a much larger amount of data than queue messages or table entries, which is suitable for Contrail where messages can have arbitrary size. Moreover, blobs can be addressed directly, a feature we used to implement item obsolescence (see Section 3). In Windows Azure, blobs are organized in containers. We dedicate a container to a contrail device name. Each message is stored in a separate blob inside the destination device's container. The name of the blob is formed by a concatenation of the DeviceID, the PortID and the ItemID. This guarantees that no two messages from different ports will ever interfere with each other.

A message is kept in persistent storage until either the message's expiry time is reached, or an outgoing TCP server reads the message from storage and transmits it downstream to the destination device. In most of the cases the outgoing TCP server of a message runs within a different worker role than the TCP server where that message has entered the cloud. An outgoing TCP server becomes active if a remote device connects to it and requests messages for a certain device name. The TCP server will then lookup the blob container of the corresponding device name and transmit any message found in one of the blobs. Once a message has been transmitted it is deleted from the persistent storage.

## 7. EVALUATION

We have evaluated Contrail using our prototype implementation. Our evaluation is divided into two parts, micro-benchmarks and scalability. In the micro-benchmarks, we show that Contrail offers applications a trade-off between latency and battery lifetime. We also show that Contrail can saturate 3G upload speeds, and offers a trade-off between throughput variance and battery lifetime. In the scalability part of our evaluation, we validate the ability of Contrail to scale to large numbers of clients by simply using more resources in the cloud. All our experiments are on a real implementation of Contrail running on Windows Azure; for clients, we use a mix of laptops tethered to 3G Windows Mobile phones, and the phones themselves.

## 7.1 Micro-Benchmarks

### 7.1.1 Latency

We first show that the end-to-end latency provided Contrail depends on the actual values of *polling-interval* and *idle-timeout*, as described in Section 4.1. In this experiment, we study the latency of sending a Contrail item between two 3G devices. Our experimental setup consists of a client/server application running on a laptop tethered with a 3G mobile phone. To simplify time measurement both client and server are running on the same laptop (we verify that this setup provides results identical to having the client and server on separate devices, detailed subsequently). Initially, the client installs a filter on the server matching items including the client's DeviceID as as part of the item's metadata. After the filter is installed and the server has received the corresponding callback, the server periodically publishes a Contrail item which matches the client's filter. Thereby, we make sure a new message is sent only after the previous message has been correctly received. We measure the time between the server publishing an item

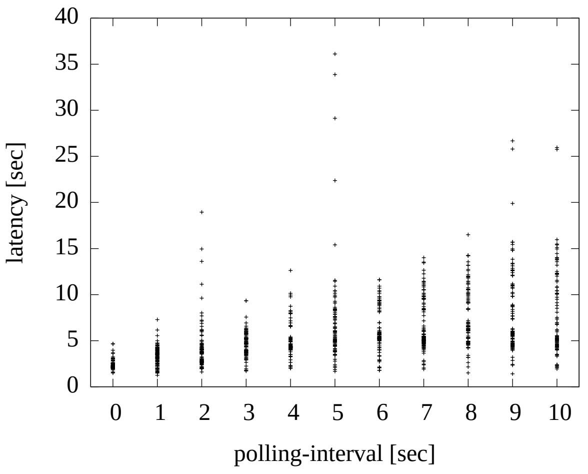

**Figure 6: Contrail latency: lower values of *polling-interval* result in lower end-to-end latency between two 3G devices**

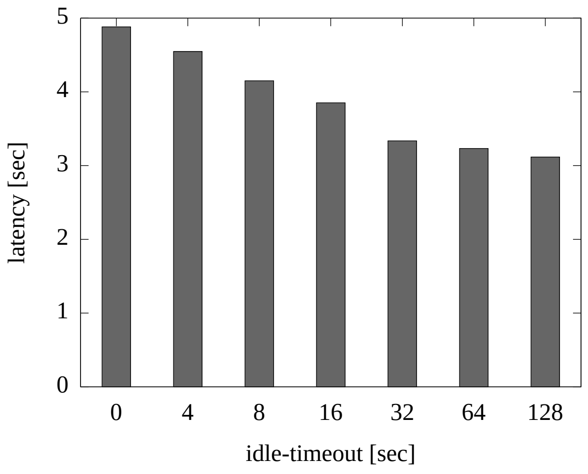

**Figure 7: Contrail latency: longer *idle-timeout* values result in improved end-to-end latency between two 3G devices**

and the client receiving it.

Figure 6 shows the measured latencies for different values of *polling-interval* while keeping *idle-timeout* zero. Each point corresponds to one single message sent between the client and the server, for a given *polling-interval*. Initially, as the *polling-interval* is small (1 second), the latency is spread around 4 seconds. Note that the latency includes the time to upstream the message, the time to downstream the message and the time to process the message inside the cloud. As we increase the *polling-interval*, the latency is increasingly spread over a larger range. For a *polling-interval* of 8 seconds for instance, the latency shows values from 1 second up to almost 15 seconds. In the worst possible scenario, each the message sent by the server arrives at the cloud just a moment after the client receiver has last polled the cloud. In the best possible scenario a poll from the client would just pick the message as it arrives.

Figure 7 shows how the end-to-end latency is affected by the *idle-timeout* parameter. The setup matches the one of the previous experiment, except that we keep *polling-interval* 4 seconds while varying *idle-timeout*. The experiments confirms the intuition that a higher *idle-timeout* leads to lower latencies. As the connection between the mobile phone and the cloud is kept open during the *idle-timeout* seconds after it is established, any message arriving during this time interval will be received immediately.

We also ran experiments measuring the end-to-end roundtrip time of a Contrail item in the situation where the sender and receiver were on two separate laptops, both tethered with a 3g phone. The roundtrip time corresponds to the time it takes for a Contrail item to be transmitted between sender to the receiver, and back from the receiver to the sender. As expected, the roundtrip time we measured in those experiments was roughly twice the latency of the corresponding one-way latency experiment.

### 7.1.2 Battery Lifetime

The previous set of micro-benchmarks clearly indicate the latency improvement that is achieved by having a lower *polling-interval* or a higher *idle-timeout*. However, this comes at the cost of reduced battery lifetime as the mobile device's radio is being used more frequently. To get a clear picture of this trade-off, we measured power consumption for various values of the *polling-interval* parameter. The results shown in Figure 8 were obtained on an HTC Touch Cruise device running Windows Mobile 6.1 using the device's 3G radio. The line marked 'Always On' corresponds to the power-hungry configuration where the client-side module continuously maintains a connection to the cloud (by setting *polling-interval* to 0 and *idle-timeout* to a large value). The Contrail application running on the phone was setup to receive a continuous series of small messages. The messages were being sent from a laptop every 2 seconds. For each parameter combination, we completely recharged the device and ran the experiment until the device shutdown due to lack of power.

As we consider longer durations of *polling-interval*, we observe that the device is able to stay running for longer periods of time. Specifically, setting *polling-interval* to 90 seconds results in the device lasting for twice as long versus when *polling-interval* is set to 5 seconds (8 hours versus 4). As expected, the 'Always On' case results in low battery lifetime; as the sender is publishing messages every 2 seconds, the receiver receives a continuous

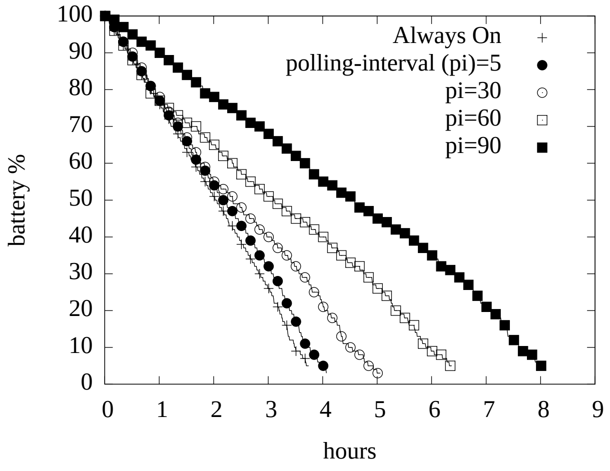

**Figure 8: Battery Consumption: Higher values of *polling-interval* result in improved battery lifetime (measured on a Windows Mobile phone).**

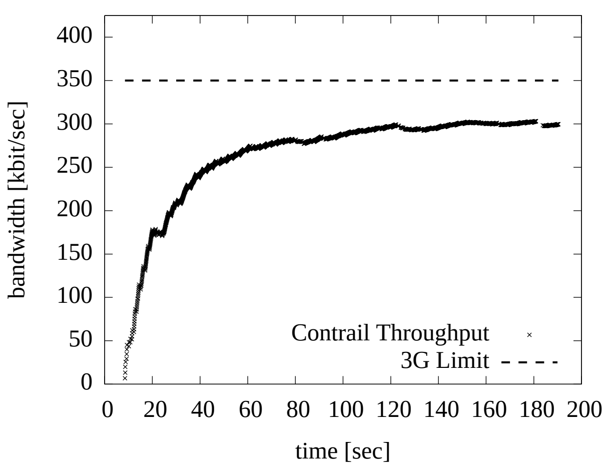

**Figure 9: Contrail throughput between two 3G devices is limited by 3G upload bandwidth**

stream of messages and never sleeps. We observe that the price of low latency is reduced battery lifetime.

### 7.1.3 Bandwidth

We ran another set of experiments to study the data rate at which two Contrail instances – both attached to a 3G network – can communicate with each other. Similar to the latency experiment the setup consists of a client and a server, both having the necessary filters installed so that they can exchange Contrailitems with each other.

In the first experiment, the server injects 2000 items in one batch, causing the Contrail module to upload the items as fast as the 3G connection permits. Figure 9 shows the data rate at which messages are received by the client over time. As can be observed, the data rate at the client approaches 300 kbit/s after a warm-up time. This rate is only slightly below the maximum 3G upload capacity we measured, which was around 350 kbit/s. The warm-up time occurs since initially not enough messages are available at the cloud to saturate the client's download capacity. To avoid the client entering the Contrail sleep phase we were using a large *idle-timeout* and a *polling-interval* of zero.

A reasonable data rate is important for applications transferring bulks of data between phones, like, e.g. pictures or small movie clips. Another type of potential applications for Contrail are streaming applications, like video, audio, or sensor streaming. Those applications typically send data at a fixed rate. We ran one experiment to see how fixed data rates are affected by the *polling-interval* and *idle-timeout* parameters. Our setup differs from the previous experiment in that the server injects a 12.5 KB message every second, resulting in a constant 100 kbit/sec data rate. Figure 10 (Left) shows the bandwidth the client sees over time when using a *polling-interval* of two seconds. As can be observed, the throughput is interrupted, which is due to the Contrail client module entering a sleep phase whenever no more messages are available to download. A more or less constant throughput can be achieved, however, by increasing the *idle-timeout* parameter. This is shown in Figure 10 (Right). A *idle-timeout* of 2 seconds is enough to make sure the TCP connection between the Contrail client module and the cloud will not terminate as long as there are messages being received at the cloud. Remember that the TCP connection only closes if no message is received for at least *idle-timeout* seconds.

### 7.1.4 Discussion of Micro-Benchmarks

Our latency, battery life and bandwidth micro-benchmarks illustrate the trade-offs for the *polling-interval* and *idle-timeout* parameters exposed by Contrail. These two parameters constitute an explicit mechanism provided by Contrail to alter communication performance. The appropriate setting for these parameters is an application-specific or user-preference policy decision. We conjecture that further research on adaptive policies may make it easier to build Contrail applications, and would be interesting future work.

## 7.2 Scale

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 experi-

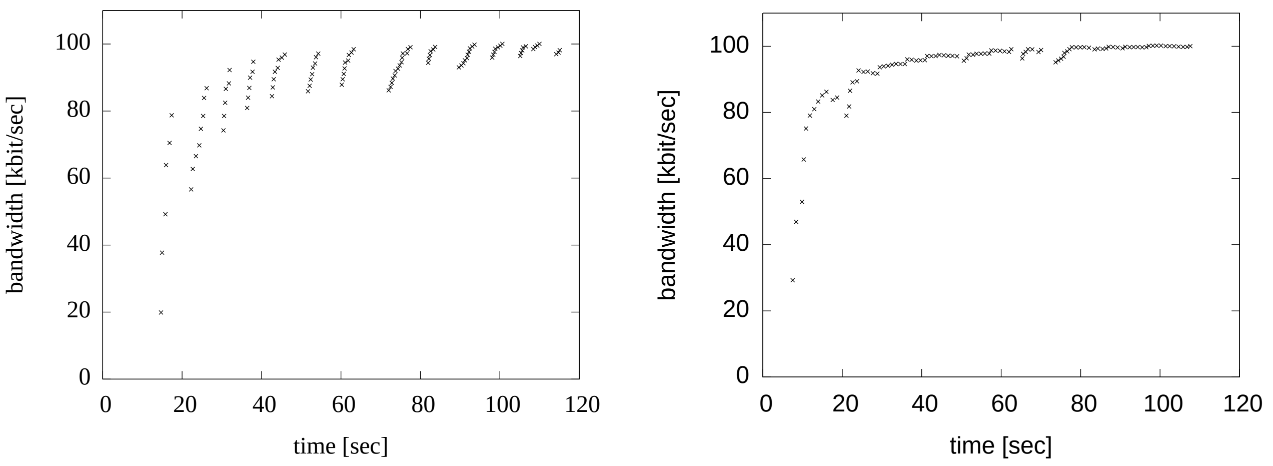

**Figure 10: In cases where the receiver's download capacity is not saturated, setting an appropriate *idle-timeout* provides applications consistent throughput. (Left: idle-timeout: 0 secs, Right: idle-timeout = 2 secs)**

ment where we varied the number of clients that were simultaneously connected to the cloud under two conditions: one where all load was being handled by a single Azure instance, and another where 10 instances were used to handle traffic. In this experiment, the clients ran on server-class machines with good network connectivity. Each client sent a message to itself every 5 seconds. The *polling-interval* parameter was also set to be 5 seconds. Figure 11 shows the average end-to-end message latency across users. We see that while a single instance can easily handle 10 simultaneous clients, supporting 100 clients at the same time results in degraded performance (an average message latency of 6 seconds instead of 3.9 seconds). However, if we support the same 100 clients with 10 instances, performance improves (average message latency of 3.4 seconds). When we ran 980 clients simultaneously on a single instance, we observed extremely poor performance as expected. However, when we increased the number of instances to 10, we were able to support 980 clients with slightly degraded performance (average message latency of 9 seconds). While this is not shown in the graph, we attempted to support 980 clients with 20 Azure instances and observed a performance improvement (average message latency of 6.5 seconds). These results indicate that the elastic nature of the cloud provides a scalable routing fabric for Contrail applications.

## 8. RELATED WORK

Content-based Publish/Subscribe [6] 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 [10].

Prior work by Ford et al. [4] has investigated naming and interconnection schemes for personal mobile devices. Haggle [17] is a network architecture for mobile devices that includes addressing and routing. MobiClique [8] 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 [2, 18] 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 [7]. Contrail can also leverage hierarchical power management techniques [16, 14]. In addition, Contrail can be easily enhanced to support upload and download priorities for data [9]; 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 [13, 5]. Contrail is complementary to such techniques; it provides a networking layer that can be used to interconnect devices and delegates.

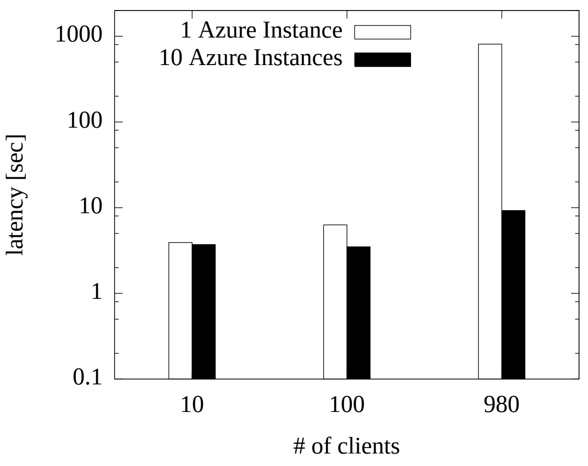

**Figure 11: Contrail scalability: adding more Azure instances enables Contrail to support more clients without performance degradation.**

Privacy-preserving computing techniques already enable specific functionality such as keyword search [3, 15]. 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 filters.

## 9. CONCLUSION

As 3G devices grow in power and functionality, the ability to share data seamlessly across them is increasingly valuable. Contrail enables data-sharing applications that are bandwidth- and power-efficient, non-intrusive, privacy-aware and supports decoupled communication between devices with non-overlapping periods of connectivity. In this paper, we showed that two primary mechanisms used by Contrail – filters and cloud-based relays – can be used to construct applications that have these properties. We show that a Contrail implementation on the Windows Azure platform leverages the scaling ability of the cloud to support large numbers of clients while retaining good latency and throughput.

## 10. REFERENCES

[1] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.

[2] 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.

[3] 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.

[4] C. L. S. R. F. K. Bryan Ford, Jacob Strauss and R. Morris. Persistent Personal Names for Globally COnnected Mobile Devices.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. Qureshi and J. V. Guttag. Horde: separating network striping policy from mechanism. In *MobiSys*, pages 121–134, 2005.

[10] 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.

[11] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM Conference on Computer and Communications Security*, 2009.

[12] M. Rosu, C. Olsen, C. Narayanaswami, and L. Luo. Pawp: A power aware web proxy for wireless lan clients. In *6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2004.

[13] 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.

[14] 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.

[15] 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*, pages 44–55, 2000.

[16] 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*, pages 261–274. ACM New York, NY, USA,

2005.
[17] 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*, 4717:391, 2007.
[18] 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*.