

# VirtualConnection: Opportunistic Networking for Web on Demand

Lateef Yusuf, Umakishore Ramachandran

Georgia Institute of Technology  
{imlateef, rama}@cc.gatech.edu

**Abstract.** Social networks such as Facebook and Secondlife are popular. Wireless devices abound from affluent countries to developing countries. Social networks need wide area Internet support. Wireless interfaces offer ad hoc networking capability, without the need for infrastructure support. Web on Demand (WoD) aims to bridge the gap between ad hoc social networks (people in close proximity with shared interests) and ad hoc networking. A key requirement for WoD is transparent connection management. This paper makes three contributions: First, an abstraction called VirtualConnection for transparent connection creation and migration, with socket-like API; second, an implementation on iPAQs (Windows Mobile) as a user level library, and proof of efficacy of using this abstraction for realizing WoD; third, evaluations to establish the performance of this abstraction. In particular, we show the performance degradation due to virtualizing the connection is negligible.

**Keywords:** connection virtualization, connection management, communication library, opportunistic networking, ad-hoc networks.

## 1 Introduction

A young school boy is helping an old blind lady cross the street in a busy city street in India. The Good Samaritan driver in a pickup truck notices the stranded car with a flat tire on the shoulder of a U.S. highway and pulls up behind the car to help fix the flat. A local farmer in a Kenyan village train station overhears two foreign visitors discussing the possibility of bad weather ruining their vacation plans. He gives them his expert opinion on the weather based on experience and offers them useful travel tips.

All of the above are simple social examples of people weaving their own *ad hoc* local human web and offering or obtaining service from others: the old lady in the Indian city did not call the Red Cross; the stranded driver on the US highway did not call AAA; the foreigners in Kenya did not turn to The Weather Channel. How can we realize the electronic equivalence of such Web on Demand?

From affluent countries to the third world, mobile devices are becoming more prevalent. They are equipped with multiple network interfaces that could allow such devices in close proximity to communicate directly with one another. Social networking is becoming very popular and trendy allowing friends and family to stay connected. However, currently social networks as we know it (such as Facebook, Secondlife) require infrastructure support in the form of wide area Internet.

In this paper, we focus on a more restricted form of *ad hoc social network*, very much akin to the human social networks that we mentioned earlier.

Today, personal wireless devices (mostly in the form of cellphones) have penetrated deep into the societal fabric. Africa is reported to have the fastest growth rate for cellphone subscribers, and by one estimate [16] over 80% of the world's population has access to some form of a mobile wireless device. Given that most users, even in the third world, may have access to devices with wireless connectivity; could mobile wireless devices in close proximity form a web of their own and serve the needs of an ad hoc social network? This is the vision behind *Web on Demand (WoD)*. WoD complements the World Wide Web and leverages it when it is available. The most important aspect of WoD is that it democratizes personalized local access to services with a global reach. Specifically, our goal is service provision on wireless devices without reliance on infrastructure support. The vision behind web on demand is to bridge the gap between ad hoc social networks and *ad hoc* wireless networking.

A key requirement and feature of WoD is automatic and adaptive connection management among interacting devices. These devices typically have multiple network interfaces with different characteristics of power, bandwidth and range. Multiple applications also have heterogeneous requirements on bandwidth, time-sensitivity and quality of service. Examples of these network interfaces include Wi-Fi adapters for wireless networks, Cellular (GSM or CDMA) interface for wide area access, and Bluetooth, ZigBee [24] and IrDA for personal area networks. A number of applications including

web browsing, video streaming and file sharing are now available for use with these devices. A connection management layer could exploit commonalities among devices in WoD and present a virtual interface for interaction and data exchange. Such data dissemination is inherently peer-to-peer in nature: a mobile node may download varying file types such as an image, audio clip or streaming video from a peer and vice versa. Therefore, it is important to guarantee the stable and efficient transmission of information even in circumstances when a particular network interface may become unavailable, such as when users leave and enter a mobile hotspot for example.

The focus of this paper is to present the design and evaluation of a connection virtualization framework for use in realizing WoD. We provide a middleware stack for opportunistic networking among the heterogeneous interfaces that may be present on each device. The architecture virtualizes the connection between two ad hoc peers and automatically chooses a physical connection based on metrics such as power and delay, transparent to the application.

*VirtualConnection* is a system to hide the complexity and heterogeneity of these network devices while at the same time supporting the communication requirements of these applications.

The main goal of *VirtualConnection* is to provide middleware for adaptive management of connections. Based on prevailing context and by a fine-tuned switching mechanism, we can choose the appropriate wireless interface in an adaptive fashion without degrading the performance of the requests made by applications. The architecture is based on the maintenance of a *virtual socket*, a socket interface for client-server applications. Applications register data transfer requests with a virtual socket which transparently manages the network interfaces, by selecting the best interface for a particular data transfer request and transparently switching among interfaces when necessary.

*VirtualConnection* identifies the factors affecting the quality of a connection, including the data rate, power consumption of application, available battery, network transmission range supported by each adapter and the time-sensitivity of the data requests. It uses these factors to determine whether there is a new best connection available to use. The system requires little or no modification to existing application logic and so can be easily adopted. The middleware sits on top of the transport layer in the networking stack and handles the automatic connection and transfer of data between heterogeneous network interfaces; thereby freeing the programmer from the burden of having to self-manage those interfaces.

While there is previous work on opportunistic networking, few of them have focused on providing system support for abstraction of heterogeneous network devices in ad hoc scenarios. Delay Tolerant Networks (such as Huggle [9, 10]) support opportunistic networking between devices but focus on message ferrying and relay rather than ad hoc connections. Ad hoc overlay networks, MANETs and mesh networks (such as PeopleNet [6] and Mob [15]) provide targeted services such as bandwidth aggregation and information queries. Also a number of projects and applications use Bluetooth and uPnP to allow devices (cellphones, PDAs, GPS receivers, digital cameras, etc.) to be “discovered” and form a personal area network (PAN) for exchanging information. These are standalone solutions that typically use one of the network interfaces rather than a connection abstraction. Rocks [12], Tesla [20], Dharma [21], and Host Mobility Manager [22] provide abstraction for end-to-end connectivity management using multiple network interfaces to resume network connections during failures and disconnections. They do not however provide a mechanism for periodic selection of optimal network interfaces, nor do they support multi-hop data transfers through heterogeneous interfaces. *VirtualConnection* also incorporates recent results from works such as Coolspots [1] and Breadcrumbs [2] on energy management.

Via the *VirtualConnection* abstraction, the paper makes the following contributions:

- An architecture for providing adaptive connectivity to applications in the presence of heterogeneous network interfaces. The connection management supports transparent migration across the network interfaces whenever necessary.
- A system design and implementation of *VirtualConnection* as a user level library providing API calls that mimic the standard socket API to applications.
- An experimental evaluation of *VirtualConnection* using a WoD prototype that confirms the efficacy of the system design. Specifically, we show that the performance degradation of *VirtualConnection* is negligible.

Sections 2 and 3 describe the system architecture and connection management for our system. Sections 4 and 5 present *VirtualConnection* implementation details and results, respectively. Section 6 presents related work. Finally, Section 7 concludes the paper with future work.

## 2 System Architecture

The VirtualConnection architecture is designed to facilitate communication among participants of a WoD. VirtualConnection supports both single-hop and multi-hop transfer modes. In single-hop mode, there is a point to point connection between entities in a WoD while multi-hop mode allows forwarding of messages among entities in a WoD. Multiple VirtualConnections in a single participant allow the participant to simultaneously connect with other members of WoD through peered connections. For example in a single-hop VirtualConnection mode, taking advantage of the multiple interfaces available to it, peer A may be using Wi-Fi to download a file from peer B, while simultaneously running a chat with peer C using Bluetooth (Figure 1-(a)).

On the other hand, in a multi-hop VirtualConnection mode, even though Peer A and Peer C have no common adapter they can use for communication, they can relay their traffic through Peer B, which has the ability to communicate with both of them (Figure 1-(b)). This scenario will only occur if Peer B explicitly chooses to allow packet forwarding. Peer A can relay traffic through Peer B's Wi-Fi adapter, which in turn passes the requested data to Peer C using the Bluetooth adapter Peer B shares with Peer C. This multi-hop configuration illustrates one of the major advantages of cooperation among participating entities in a typical WoD.

The VirtualConnection abstraction is designed to guarantee connectivity during the lifecycle of a particular peer connection. The lifecycle of a connection can be divided into the following stages:

- *Connection Initiation*: This happens at the beginning when an application requests a connection with a peer. It includes the discovery of adapters on both peers, the selection of the adapter to use and the establishment of a control socket to monitor the progress of the connection.
- *Connection Maintenance*: Immediately after the connection has been established, the virtual connection enters maintenance mode. This requires an evaluation of the connection using the required metrics. If there is a better interface for communication, the connection can be migrated to the new interface. Also, when there is a disconnection from one interface, the connection will be automatically switched to any available interface. Any pending data is sent after the migration is completed, thereby avoiding data loss. If no interface is available to do the switching, the connection will be suspended and the application will be notified.
- *Connection Teardown*: This involves closing the virtual socket by closing all physical sockets involved in the connection. The network adapter can be disconnected to save power if there is no application requesting a connection.

VirtualConnection uses a simple abstraction of a *virtual socket* to manage each connection request. A virtual socket is the logical link that manages the sending and receiving of data packets between two peer endpoints. Each virtual socket can communicate with an endpoint of different transport types including TCP, UDP or Bluetooth. The virtual socket is responsible for receiving and delivering data to the applications. The virtual socket keeps logical send and receive buffers that correspond to those of the underlying transport protocols supported on the physical interfaces. The operations supported by the virtual socket are:

- **open(p, t, r)**: opens a new connection with a given peer  $p$ , using a transport  $t$ ;  $r$  is a Boolean measure of the time-sensitivity of the packets which can be true (real-time) or false (normal);  $p$  must include an initial address and address type to connect to.
- **send(d, c, p)**: sends  $c$  bytes from buffer  $d$  to the peer  $p$ .

Fig. 1. WoD Prototypes.

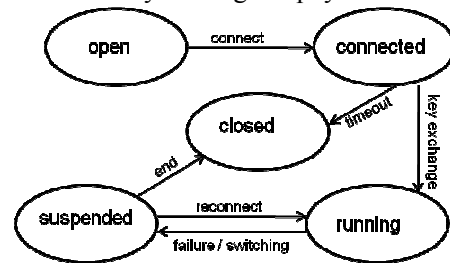
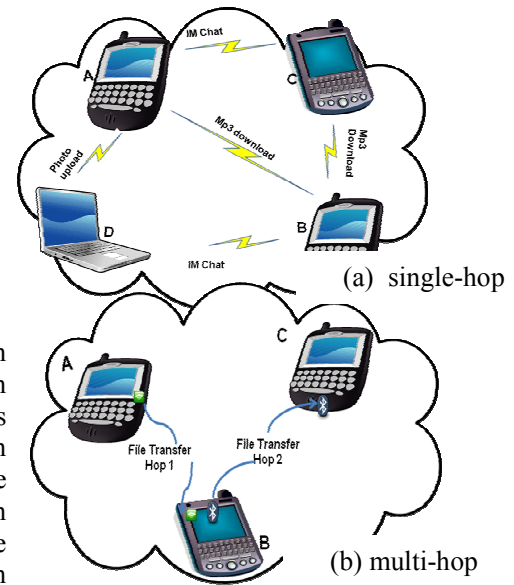


Fig. 2. State Diagram of VirtualConnection.

- **receive (d, c, p)**: receives  $c$  bytes from peer  $p$  into buffer  $d$ .
- **close()**: closes the virtual socket and removes the application from the application list.

The benefit of this API design is to resemble the native socket API as much as possible without supporting the full-range of the native socket API methods. The eventual goal of VirtualConnection is to use virtual socket to replace native socket calls by implementing a system-wide hook that traps operating system socket calls and extends them with virtual socket calls (similar to the Exokernel [19] approach). The WoD library will be interposed between the application code and the operating system. It will export the operating system socket API to applications and transparently use the WoD library in ordinary applications. This will allow the execution of existing applications without any modification to the application code.

**Fig. 3.** System Architecture.

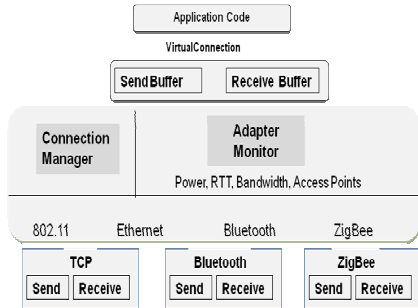


Figure 3 illustrates the architecture of the proposed system. The core modules in the architecture are the Connection Manager and the adapter monitor. The Connection Manager is responsible for adaptive network management. It creates a virtual socket on connection start and maintains the virtual socket throughout the lifetime of the connection. It is responsible for switching between adapters when there is a better option for the connection. The new connection option may be cheaper because it saves power and has a better throughput. If a real-time transport is requested, the Connection Manager uses the highest-bandwidth adapter and

**Table 1.** Events raised by the Connection Manager.

Event Type	Meaning
CONN_NEW	A new connection has been established
CONN_OFF	The active channel has lost connection
CONN_ON	The active channel has resumed connection
CONN_SWIT	A new active channel is proposed for the
CH	connection
CONT_DEL	One or more control socket has been removed
CONT_CHG	One or more passive channel has lost connection

does not attempt to switch the connection between adapters. This is to prevent performance degradation because of switching costs. The Connection Manager uses the logical abstraction of a *channel* to manage

connections between peers. The channel is the pipe between two virtual sockets and is responsible for the creation and deletion of physical sockets supported by the different adapters. While the architecture is general and extensible to any number of protocol stacks, the current implementation supports two types of system sockets by the channel: TCP and Bluetooth. Channels can be *active* or *passive* depending on whether they are currently involved in data transfers. Each channel has two background threads that continually estimate RTT and bandwidth values used in evaluation of the current connection between any two peer endpoints. Therefore, a single virtual socket may have multiple underlying channels corresponding to the identical types of physical sockets available between two given peers. Table 1 lists some of the events the Connection Manager generates during the lifetime of a specific connection.

The Adapter Monitor gathers statistics about the state of each adapter on the peer including type of adapter (Wireless 802.11, PPP, Ethernet, Bluetooth, ZigBee), current address of the adapter (IP address, Bluetooth address, ZigBee address), state of the adapter (On, Off, Connected, Disconnected, Sleeping, Transmitting, Idle), speed of the adapter, current throughput of the adapter, estimated bandwidth capability, battery level (0-100) of the mobile device and access points available at the adapter (Table 2). Adapter Monitor opens a connection between the two peers and transmits a random amount of data to estimate end-to-end throughput and bandwidth [2], [4]. The throughput is expressed in terms of the RTT of the data transfer.

**Table 2.** Messages reported by Adapter Monitor.

	Message Type	Description
<b>3 Connection Management</b>	INT_LIST	List of current interfaces
	INT_STATUS	Active connection status of an interface
	INT_OPSTATUS	Operational status of an interface
	INT_SPEED	Speed of an interface
	INT_BANDWIDTH	Bandwidth of an interface
	TH	
	ACTIVE_INT	Request the active interface
	INT_TYPE	Request the adapter type
	INT_PWR	Power consumption of adapter as a ratio of lowest adapter

In this section, we introduce the finer details of adaptive connection management. We begin by providing a step-by-step

elaboration of the activities, events and messages that are exchanged between peers during connection establishment and maintenance (Sections 3.1 and 3.2). We then describe how VirtualConnection determines the appropriate adapter to use for a given connection (Section 3.3). We conclude with a detailed description of connection establishment when VirtualConnection runs multi-hop mode (Section 3.4).

### 3.1 Connection Negotiation

Table 3 lists the different message types exchanged between peers during the lifetime of a connection. After an application invokes an open() call, a number of things happens under the cover. VirtualConnection retrieves the list of adapters (using the INT\_LIST request message shown in Table 2) and their network information including the INT\_TYPE, INT\_BANDWIDTH, INT\_STATUS, and INT\_OPSTATUS) from the adapter monitor. It connects to the initial address of the peer, passed through the open() call. VirtualConnection then attempts to establish a virtual socket with the desired peer by performing the following steps:

1. *Determine common adapters*: It exchanges the adapters' list (ADAPTERSLIST message) with the peer using the specified address and determines the identical types of adapters between the two lists (COMMONADAPTERS message).
2. *Establish the control sockets*: It creates a control socket for each and every common adapter between the two peers. The control socket is used to detect data connection failure and for the initial estimation of RTT and the bandwidth of the adapters. A UDP socket is used for IP-based adapters while a Bluetooth socket is used for Bluetooth adapters.
3. *Create a virtual socket*: It chooses the best network adapter and creates a virtual socket. The virtual socket is bound to the address of the chosen adapter.
4. *Exchange buffer size*: The peers exchange the size of their TCP receives buffer sizes (BUFFERSIZE message).
5. *Exchange network address*: The peers exchange the network address of the chosen adapters (NETADDRESS message).
6. *Creates connection cipher*: VirtualConnection establishes an encrypted identifier for the connection and then performs a key exchange for later authentication (CIPHER message).

Subsequent data transfer requests with calls to send() and receive() use the chosen adapter until an improved connection is available. When an application sends data, VirtualConnection keeps a copy of the data in an in-flight buffer similar to that specified in Rocks [12]. It increments the count of bytes sent when an application sends data and increments the size of bytes received when an application receives data. The size of the in-flight buffer is the sum of the size of its socket send buffer and the size of the peer's socket receive buffer. It uses a simple sliding window protocol to discard older data whenever new data arrives. Incurring an additional copy for virtualizing the physical connection is necessary to avoid data loss during migration from one physical adapter to another. We show that this additional copy has little impact on the actual performance measured with VirtualConnection (see Section 5.2).

**Table 3.** Messages exchanged by VirtualConnection.

Message Type	Description
ADAPTERSLIST	List of network adapters
CIPHER	Connection ID
BUFFERSIZE	Peer Receive Buffer Size
COMMONADAPTE RS	List of identical types of adapters
NETADDRESS	Active channel network address
SWITCHADDRESS	Begin Connection Switching

### 3.2 Connection Migration and Switching

When the Connection Manager raises the `CONN_OFF` or the `CONN_SWITCH` event (see Table 1), the virtual socket initiates the switching procedure. It creates a new physical socket at the new chosen adapter and then performs the following steps:

1. *Suspend all I/O calls:* It blocks all I/O operations (`send()`, `receive()`) until switching is completed
2. *Send a switch connection message:* it sends a message (`SWITCHADDRESS`) to the peer notifying a connection switch and the network address to be used for further communication. The message is sent through the control sockets of all common adapters determined at the initiation of the connection.
3. *Authenticate:* The peers authenticate by exchanging the connection cipher created during the execution of an `open()` call.
4. *Close existing socket:* It closes the previous physical socket and binds the virtual socket to the newly-created socket.
5. *Send in-flight data:* VirtualConnection retransmits any data that is pending in the in-flight buffer. It determines the amount of in-flight data it needs to resend by comparing the number of bytes that were received by the peer to the number of bytes it sent. This ensures no data is lost as a result of the switching process.

To handle failures, VirtualConnection implements the switching procedure on a `CONN_OFF` event differently from that of a `CONN_SWITCH` event. On a `CONN_SWITCH` event, VirtualConnection maintains a timeout for each stage of the switching procedure. The timeout corresponds to twice the average time measured for each stage of the switching procedure. It then aborts the switching procedure if the timeout expires and immediately tries to use the existing active channel for communication. However, no timeout is used in the case of a `CONN_OFF` event since there is no existing connection which can be used to process data transfers and the goal in that case is to try to reestablish connection as soon as feasible.

It should be noted that the application is completely unaware of this physical adapter switching; once the switching is complete, normal operation resumes using the new interface.

### 3.3 Determining the Optimal Connection

The Connection Manager retrieves metrics from the adapter monitor such as the state of the adapter, bandwidth, throughput, and battery level. Each metric has an assigned weight which is used to calculate the cost of connection with that adapter. The adapter with the lowest connection cost is chosen for a given connection. Since there has been extensive work on battery and power management, the current VirtualConnection architecture simply uses the results in [1] and [3] to reduce power degradation. CoolSpots [1] explores many policies for switching between Wi-Fi and Bluetooth adapters in order to enhance battery life. We adopt the “cap-static” policy which performs best among a range of data transfer benchmarks. The cap-static policy recommends a switching period of 250ms. Therefore, the evaluation of adapters to determine whether there is a “cheaper” adapter occurs every 250ms in our implementation as well, and switching is performed when two consecutive evaluations suggests a better connection. The evaluation requires retrieving metrics from the adapter monitor such as the state of the adapter (connected, disconnected) and the current power level. Adapter Monitor returns `INT_LIST` sorted by `INT_OPSTATUS`, then by `INT_BANDWIDTH`, and then by `INT_PWR`. When the measured power level is below 20% [3], the connection prompts a switch to the cheapest power adapter available (in the case of Wi-Fi and Bluetooth, the connection is switched to Bluetooth).

### 3.4 Multi-hop Configuration

VirtualConnection can run in one of two modes: *single-hop* and *multi-hop*. The multi-hop mode allows packet forwarding in situations which may otherwise be impossible as seen in Figure 1-(b). Even though Peer A and Peer C have no common adapter they can use for communication, they can relay their traffic through Peer B, which has the ability to communicate with both of them. This scenario will only occur if

Peer B explicitly chooses to allow packet forwarding. Peer A can relay traffic through Peer B Wi-Fi adapter, which in turn passes the requested data to Peer C using the Bluetooth adapter Peer B shares with Peer C. This multi-hop configuration illustrates one of the major advantages of cooperation among participating entities in a typical WoD.

VirtualConnection uses a modification of the Ad-hoc On-demand Distance Vector (AODV) [23] protocol to support multi-hop application-layer routing. For the example in Figure 1-(b), when Peer B intends to communicate with a Peer C whose route is not known, it broadcasts a Route Request packet (RREQ). Each RREQ contains an ID, sequence numbers, hop count, source and destination node addresses, source and destination address type, and control flags (Table 4). The RREQ is sent through all the available network interfaces Peer A shares with Peer B, which then propagates the RREQ to Peer C. When the RREQ reaches Peer C, a Route Reply packet (RREP) is generated and forwarded along the RREQ route back to Peer A. Each RREP contains destination sequence numbers; route lifetime, hop count, source and destination node addresses, source and destination address type, and control flags. To avoid broadcast storms, intermediate nodes, such as Peer B, simply drop duplicate RREQ.

The multi-hop multi-radio connection configuration can be divided into the following stages:

1. *Routing Initiation*: VirtualConnection attempts to make a connection to the destination address of the peer through each of its network interfaces. After a timeout and number of retries, multi-hop routing is enabled. Otherwise, if any of the direct connections is successful, direct routing is used.
2. *Route Discovery*: VirtualConnection tries to locate a possible route to the destination through the following steps:
  - The source node initiates the discovery by broadcasting an RREQ packet through all interfaces.
  - If any of the intermediate nodes has a valid route for the destination in its cache, the node will send an RREP to the source node. Otherwise, if the intermediate node does not know of such a route, it again broadcasts the RREQ through all of its network interfaces.
  - Any duplicated RREQ from the same sender and same sequence number is simply discarded.
  - When an RREQ reaches the destination node, it adds the route to its cache. It then generates an RREP which it sends through the Reverse Route back to the source. The destination responds to every new RREQ that it receives.
  - Upon receiving an RREP, the source adds the route to its cache. It then attempts to connect to the destination node.
3. *Data Transfer*: Once a valid route is discovered and a connection is established, VirtualConnection begins processing data request to and from the source node to the destination.

**Table 4.** VirtualConnection RREQ Header.

RREQ			
type	flags	Recvd	hopcnt
Destination Address			
Destination Address Type			
Destination Sequence Number			
Source Address			
Source Address Type			
Source Sequence Number			

## 4 Implementation

In this section we describe the salient implementation details of VirtualConnection. The implementation described here is written in C# 3.0. The system is a user library that exposes the API calls described in Section 2. We assume that the initial endpoints (IP Address, Bluetooth Address) of a peer requesting a new connection are known at the start of a connection. The implementation shares a single adapter monitor among all connections and a unique connection manager for each individual connection. The adapter monitor is a singleton class that is instantiated by the first connection request. The adapter monitor uses a background thread that monitors the status of all the adapters and raises one of three possible events: connected, disconnected, operational. Using the singleton class and raising events allow multiple applications to share the same adapter monitor, saving on system resources. The per-VirtualConnection Connection Manager subscribes to the adapter monitor events and initiates adapter switching or connection migration when any of the events are raised. The implementation uses the .NET *XmlSerializer* class to serialize messages, lists and objects used by the VirtualConnection.

#### 4.1 Interface Detection

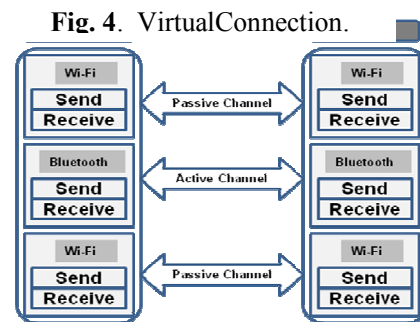
The OpenNETCF framework *OpenNETCF.Net.NetworkInformation.NetInterface* class supplies most of the needed statistics used by *VirtualConnection*. The list of all interfaces is retrieved using the *NetInterface.GetAllNetworkInterfaces()* function call. The class also supplies the speed, connection status, and operational status of the *NetworkInterface* among other properties. The Bluetooth radio's statistics are retrieved using the *InTheHand.Net.Bluetooth.BluetoothRadio* class. The RTT of the network interface is estimated using a simple ping packet. The bandwidth estimation involves downloading a random size byte array from the peer. Each interface address is represented by two fields: address data (a byte array), and address type (an enumeration).

#### 4.2 Connection Switching

On an *open()* *VirtualConnection* API call, the Connection Manager retrieves the list of adapters from the adapter monitor (the *INT\_LIST* message), exchanges the list with the peer at the address supplied at *open()* and then creates a new virtual socket to handle all data transfer requests on this new connection. It determines common adapter types and records their known addresses. The Connection Manager also creates a channel for each identical type of adapter (Figure 4). Each channel then creates a control socket that is used for exchanging control information; switch connection messages, connection cipher and adapter address changes. After the creation of the channels, the Connection Manager creates a globally unique identifier (GUID) which serves as the connection cipher. The GUID is then exchanged between the two peers and kept in a connection list, which is an array of existing connections. It selects an active channel and binds the channel to the virtual socket. Any invocation of *send()* and *receive()* uses the underlying system socket currently assigned to the active channel.

#### 4.3 Routing

The windows kernel selects the default outgoing network interface for IP packets. However, an application can bind a local socket to a specific network interface. After selecting the adapter used by the active channel, *VirtualConnection* binds the local socket to the address of the adapter. Data is then routed through the adapter to the destination. The multi-hop configuration uses a routing table that contains the destination address, next-hop and the network interface to use. In multi-hop mode, the Connection Manager initially tries a direct connection to the peer on an *open()* API call. If attempt fails, RREQ packets are then broadcasted to nodes in order to discover a path to the peer. If no path is found after a configurable timeout, an error status is reported. The Bluetooth RFCOMM protocol is used for data transfers over Bluetooth and TCP/IP is used for data transfers over Wi-Fi. They are both reliable transport protocols that do not require additional management for data loss.



### 5 Evaluation

We have implemented *VirtualConnection* prototype on Windows Mobile 6, as a user-level library. The implementation is developed using the .Net Compact Framework 3.5, the OpenNETCF Smart Development Library 2.2 and the Bluetooth library from IntheHand.Net using the C# programming language. The Bluetooth implementation currently supports the Microsoft Bluetooth stack. The Wi-Fi adapters are programmatically configured to ad hoc mode. The prototype WoD consists of three mobile devices and one laptop as shown in Figure 1. We use three windows mobile devices (two HP iPAQ *111 Classic Handheld*, and one HP iPAQ *hw6940*), running Windows Mobile 6.0 operating system and having both Wi-Fi and Bluetooth network devices. Each iPAQ has a with 1200 mAh, 3.7 Volt battery. The laptop has an inbuilt Wi-Fi interface. The Wi-Fi interfaces are all 802.11b devices with a raw maximum of 11Mb/s data rate. The Bluetooth interfaces are all Bluetooth 2.0 devices with a raw maximum data rate of 2Mb/s. Each device runs a download application or a chat application. The configuration creates peer wise connections between the members as shown in Figure 1.

We have done preliminary performance evaluation of our *VirtualConnection* library for use in WoD applications. The experiments we report below fall into three categories:

1. Micro measurements to quantify the set up cost of *VirtualConnection* and relate them to what would be incurred on a native protocol stack (Wi-Fi or Bluetooth). The micro measurement also includes connection migration cost when an adapter switch is required/desired.

2. Data transfer measurements to quantify the latency and throughput using VirtualConnection and relate them to corresponding results using Wi-Fi and Bluetooth.
3. Automatic connection switching measurements to show the VirtualConnection at work in switching among available interfaces, which is reflected in the observed latency for data transfer.

### 5.1 Connection and Switching Costs

We have measured the delay for establishing a connection between each peer wise connection in the WoD prototype. The experiments are repeated 100 times and the average values are recorded as shown in Table 5. We compare the values with that of using only a plain Bluetooth implementation and a plain Wi-Fi implementation.

The overhead incurred by VirtualConnection for connection setup is less than 600ms. It involves delay due to exchange of adapters' lists, chosen channel address and the connection cipher. The experiments show that VirtualConnection does not incur significant startup penalty. The overhead is minimal and acceptable for the additional functionalities provided by VirtualConnection for the intended scenarios where WoD would be deployed.

**Table 5.** Connection Costs.

Setup Stage	Costs(millisecon ds)
Socket Initiation	0.54
Adapters List Exchange	392.1
Transfer Adapter Determination	188.7
Connection ID	0.69
Control Sockets Establishment	2.68
Connection Establishment	7.5
Total Setup	585.6

every 250 ms and recommends switching after two consecutive positive evaluations. The automatic switching time is less than 200 ms and is considerably less than the delay that would be incurred for a manual migration of the connection between the peers for example.

**Table 6.** Switching Costs

Switching Stage	Costs (milliseconds)
Switch Message	3.28
New Channel Creation	96.13
Connection	7.51
Reestablishment	
Switch Message	3.28

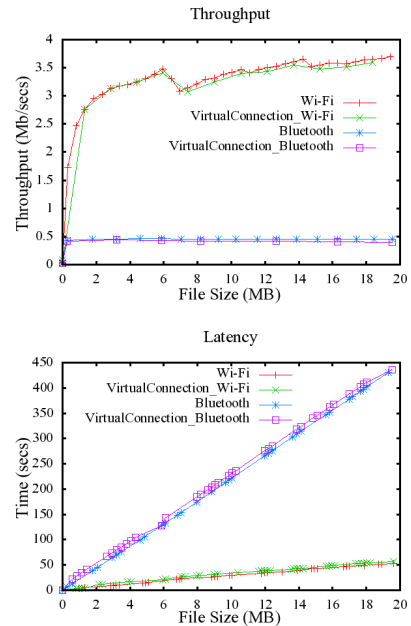
### 5.2 Throughput and Latency

We have measured the average throughput and latency of downloading a 20 MB file between two iPAQs in the WoD prototype: iPAQ A and iPAQ B. The first set of experiments measured the performance of VirtualConnection using a single adapter by disabling one of the adapters on both iPAQs. The results are shown in Figure 5. The throughput and latency values compare favorably with the results of native implementation of the same experiments on 802.11 and Bluetooth network interfaces. This is because the main cost incurred by VirtualConnection is the copying costs of maintaining an in-flight data buffer. The latency overhead is very small (an average of about 7.4ms for a 20MB file, less than 0.2% of native implementation) and imperceptible at the application level. The throughput is also close to measured values from native implementation.

Table 6 shows the switching costs incurred in switching between Wi-Fi and Bluetooth.

Most of the overhead incurred is due to synchronizing the two peers to agree on a new underlying adapter to use. Once the new socket has been established, the connection time is almost equal to that incurred for establishing a bare TCP or Bluetooth socket. The switching event takes about 600 ms to detect since the Connection Manager evaluates the connection

**Fig. 5.** Average throughput and latency of VirtualConnection over Wi-Fi, Bluetooth.



To demonstrate the switching between the two adapters, the connections start with a Bluetooth connection and an mp3 file (20MB) is downloaded by iPAQ A from iPAQ B. During the transfer, iPAQ B is gradually moved away from iPAQ A until the Bluetooth connection becomes unavailable and the system automatically switches to Wi-Fi (about 9 meters). Then iPAQ B is moved back to iPAQ A until the connection switches back to Bluetooth. The graph (Figure 6) shows how the observed latency change as the VirtualConnection switches back and forth between the two interfaces. Incidentally, these numbers, respectively correspond to the latency numbers recorded in the previous latency experiment with either Wi-Fi or Bluetooth (compare Figure 5-(b) and Figure 6).

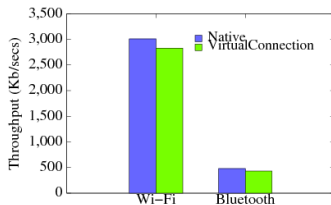
Figure 7 shows that the mean throughput in the WoD is very close to that obtained from native implementations. To calculate the mean throughput, we establish up to ten different simultaneous file transfers among peers in the WoD prototype.

### 5.3 Resource Usage

We measured the average battery level when a peer wise connection is run for as long as fifteen hours. The current battery level is determined

with the *PowerBatteryStrength* function in the *Microsoft.WindowsMobile.Status.SystemState* class. To reduce energy wastage, the Adapter Monitor actively measures the channel capacity of just the top two adapters. As an additional optimization

**Fig. 7.** Mean throughput in WoD prototype. The mean is computed from ten simultaneous file transfers among peers in the WoD prototype.

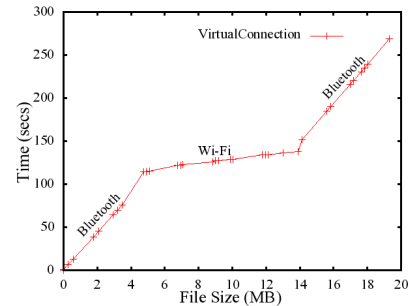


power incurred by the control sockets. In actual usage, VirtualConnection enhances battery lifetime since the cheapest power adapter (Bluetooth in this case) is preferred whenever possible. We demonstrate energy savings (about 67%) of VirtualConnection by measuring the battery level during a switching experiment similar to that explained in Section 5.2 but with a 1.5 GB file (Figure 8).

## 6 Related Work

There is a number of works that use heterogeneous wireless interfaces for data transfer. Many of these works are aimed towards energy efficiency by using another interface to complement Wi-Fi and save power (examples include Coolspots [1], and Wake-on-wireless [2]), while our work targets abstracting the heterogeneous interfaces to provide transparent connectivity in ad hoc scenarios. Projects such as Breadcrumbs [2] and Context-for-wireless [7], and Blue-Fi [25] allow the use of multiple network adapters for communication but focus on prediction algorithms to determine when to use the cheapest network adapters in terms of power savings. They require geo-location using GPS, cell IDs or Bluetooth access points and the use of the system by a mobile user along a predictable path. VirtualConnection is

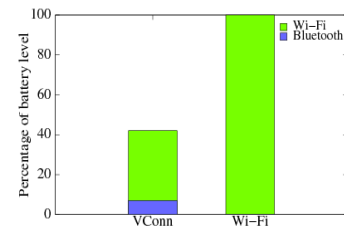
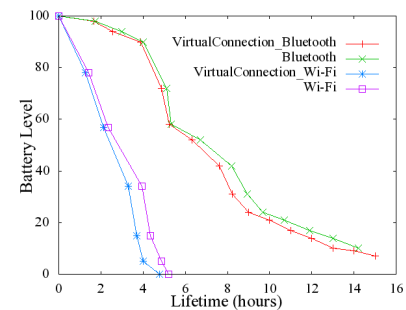
**Fig. 6.** Average latency using VirtualConnection with connection switching.



when the client device has low battery power (about 20% remaining), the adapter monitor discontinues active bandwidth estimation and

only monitors the connection and operational status of the adapters. The results show that additional power costs incurred by

VirtualConnection over the native implementation is minimal. The overhead is due to the extra power consumption for bandwidth estimation and additional



**Fig. 8.** Battery Lifetime of iPAQ 111 running VirtualConnection. The top figure compares the energy usage of VirtualConnection to that of the native implementations. The bottom figure illustrates energy savings using VirtualConnection with connection switching.

complementary to such systems, and can be leveraged by these systems to abstract their connection management.

Rocks [12], Tesla [20], Dharma [21], and Host Mobility Manager [22] allow connectivity management in mobile hosts but are focused on IP-based session management in client-server applications as an extension for Mobile IP.

Some other works (such as [13, 14, 17, 18]) focus on the cooperation between Wi-Fi and Cellular networks to improve performance in multiple radio scenarios. Our work does not require the cooperation between adapters and is an application layer solution. MultiNet [8] considers connecting a single Wi-Fi card to multiple access points to increase the capacity of the device and saving on energy. Our work is focused on using multiple adapters to connect to multiple access points.

Another set of related work includes opportunistic networking approaches such as Haggie [10, 11], and ad hoc networks such as PeopleNet [6] and Mob [15]. They are focused on delay tolerant networking and wide area social services, respectively. Our work is targeted towards ad hoc networks where the members are co-located and can exist as a standalone web or interact with other webs.

## 7 Conclusion

With the penetration of mobile wireless devices in the societal fabric from the affluent countries to the third world, there is a huge opportunity to use these devices for realizing what we call Web on Demand - a bridge between social networks and the wireless connectivity of mobile wireless devices. A key enabler for WoD is connection transparency so that the WoD can be realized with whatever connectivity is currently available and best suited for serving the need of the WoD. We have proposed an abstraction called VirtualConnection that does precisely this. It provides automatic pair-wise selection of the network interface based on quality of service and other parameters that are specifiable from the application level for the set of devices that need to form a WoD.

This paper makes several contributions. First it presents an architecture for realizing the VirtualConnection abstraction. The architecture is general and can accommodate any number and types of network interfaces. The connection management part of the architecture does three things: automatically selects the network interface given the needs of an application; continually monitors the interfaces to detect changes (such as disconnection, power usage, etc.); and triggers switching from one interface to another transparent to the application. Second, we have implemented the abstraction as a user level library providing API calls that mimic standard socket interface. The implementation runs on Windows Mobile 6 and currently supports two network interfaces, namely, 802.11 and Bluetooth. We have shown the efficacy of using this library by implementing a prototype WoD supporting two applications: file download and chat. Third, we have conducted preliminary evaluations to show that the performance of VirtualConnection is close to native implementation of sockets. The evaluations include costs involved in connection set up, switching interfaces, latency and throughput measurements for file transfer, and the automatic interface switching by the VirtualConnection. We are currently investigating appropriate mechanisms for extending the regular socket library of native operating systems with VirtualConnection (similar to efforts such Exokernel [19]) for allowing its use without requiring any change to the applications.

This paper only addresses the connection management aspect of the WoD vision. There are a number of problems that need to be addressed including name management for a number of WoDs a given device may want to participate in (based on interests), isolation and protection guarantees between such WoDs, etc. These issues are complementary to the transparent connection management problem that is at the heart of realizing the WoD vision, and represent possible future directions of our research.

## References

1. Pering, T., Agarwal Y., Gupta R., Want R.: CoolSpots: Reducing the Power Consumption of Wireless Mobile Devices with Multiple Radio Interfaces. In: Proceedings of the Annual ACM/USENIX International Conference on Mobile Systems, Applications and Services pp. 220--232 (2006)
2. Nicholson, A., Noble, B.: BreadCrumbs: Forecasting Mobile Connectivity. In: Proceedings of the 14th Annual ACM International Conference on Mobile Computing and Networking, (2008)
3. Ravi, N., Scott, J., Iftode, L.: Context-aware Battery Management for Mobile Phones. In: PerCom '08: Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications, Dallas, TX (March 2008)
4. Prasad, R., Murray, M., Dovrolis, C., Claffy K.: Bandwidth Estimation: Metrics, Measurement Techniques, and Tools, IEEE Network (June 2003)

5. Andersen, D., Bansal, D., Curtis, D., Seshan, S. Balakrishnan, H.: System Support for Bandwidth Management and Content Adaptation in Internet Applications. In: Proc. Symposium on Operating Systems Design and Implementation (October 2000)
6. Motani, M., Srinivasan, V., Nuggehalli, P.S.: PeopleNet: Engineering a Wireless Virtual Social Network. In: Proceedings of the 11th annual International conference on Mobile Computing and Networking, 243--257 (2005)
7. Rahmati A., Zhong, L.: Context-for-wireless: Context-sensitive Energy Efficient Wireless Data Transfer. Proceedings of the 5<sup>th</sup> International Conference on Mobile Systems, Applications and Services, 165--178 (2007)
8. Chandra, R., Bahl, P., Bahl, P.: MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card. In: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Hong Kong, China, 882--893 (March 2004)
9. Leguay, J., Lindgren, A., Scott, J., Friedman, T., Crowcroft J.: Opportunistic Content Distribution in an Urban Setting. In: Proceedings of the 2006 SIGCOMM workshop on Challenged Networks, 205--212 (2006)
10. Huggle [http://www.huggleproject.org/index.php/Main\\_Page](http://www.huggleproject.org/index.php/Main_Page)
11. Scott, J., Hui, P., Crowcroft, J., Diot, C.: Huggle: A Networking Architecture Designed Around Mobile Users. In Third Annual IFIP Conference on Wireless On-demand Network Systems and Services (WONS 2006), Les Menuires, France (January 2006).
12. Zandy, V., Miller, B.: Reliable Network Connections. In: Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking, Atlanta, GA, 95--106 (September 2002)
13. Salkintzis, A.K., Fors, C. Pazhyannur, R.: WLAN-GPRS Integration for Next-Generation Mobile Data Networks. IEEE Wireless Communications 9 (5), pp. 112-124.
14. Buddhikot, M.M., Chandranmenon, G., Han, S., Lee, Y.W., Miller S: Design and Implementation of WLAN/CDMA2000 Interworking Architecture. IEEE Communications 91
15. Chakravorty, R., Agarwal, S., Banerjee, S., Pratt I.: Mob: a Mobile Bazaar for Wide-area Wireless. In: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking, pp. 228--242 (2005)
16. Universal access – how mobile can bring communications to all (2005).
17. Bharghavan, V.: Challenges and Solutions to Adaptive Computing and Seamless Mobility over Heterogeneous Wireless Networks, Int. Journal on Wireless Personal Communications vol. 4 (2), pp. 217--256
18. Hsieh, H.Y., Kim, K. H, Sivakumar, R.: An End to End Approach for Transparent Mobility across Heterogeneous Wireless Networks, Mobile Networks and Applications, vol. 9(4), pp. 363--378 (2004)
19. Engler, D., Kaashoek, F., O'Toole J.: Exokernel: An Operating System Architecture for Application-Level Resource Management. In: Proc. 15th SOSP, Copper Mountain, CO, pp. 251--266. (Dec. 1995)
20. Salz, J., Snoeren, A.C., Balakrishnan, H.: TESLA: A Transparent, Extensible Session Layer Architecture for End-to-End Network Services. In: Proc. of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS), March 2003
21. Mao, Y., Knutsson, B., Lu, H., Smith, J.: DHARMA: Distributed Home Agent for Robust Mobile Access. In: Proceedings of the IEEE Infocom 2005 Conference, Miami (March 2005)
22. Peddemors, A., Zandbelt, H., Bargh, M.: A Mechanism for Host Mobility Management supporting Application Awareness. In: Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04) (June 2004)
23. Perkins, C., Royer, E. M., Das, S.: Ad hoc On-Demand Distance Vector (AODV) Routing. IETF RFC 3561 (2003)
24. ZigBee Alliance <http://www.zigbee.org/>
25. Ananthanarayanan, G., Stoica, I.: Blue-Fi: Enhancing Wi-Fi Performance using Bluetooth Signals. In: Proceedings of the Seventh International Conference on Mobile Systems, Applications, and Services (MobiSys'09), (2009)