Tools for Collaboration in Metropolitan Wireless Networks

G. Sibley and V. S. Sunderam

Math & Computer Science
Emory University
Atlanta, GA
30302
{gsibley | vss}@emory.edu

Abstract. This paper presents RRNAPI, a toolkit for accessing network performance in Ricochet Radio Networks. It is envisioned that these tools are a necessary stepping stone in developing frameworks for distributed computing in wireless networks, in particular, extending Collaborative Computing Frameworks (CCF). Many forms of traditional distributed computing require a reliable network, however these programs do not extend to wireless networks because reliable connectivity is not generally possible with wireless networks. This paper explains connectivity issues in Ricochet Networks, presents practical solutions to these problems, and explains a toolkit that can be used when developing distributed applications to address these problems.

Keywords – Wireless, Ricochet, Collaborative Computing

1 Introduction

Today we witness the proliferation of wireless technologies such as IEEE 802.11a/b LAN [10], Bluetooth [6], and Ricochet [11] to name a few. While these technologies and the wireless networks they comprise promise to free users from the desktop, they also pose problems for many network applications. In particular, software that requires consistently reliable networks begins to fail. Many software design techniques used in distributed computing are among those that traditionally require a reliable network. In the context of distributed computing, the terms traditional software or legacy software refers to and includes such projects as Legion [5], GLOBUS [4], HARNESS [2], CCF [9], PVM [16], etc. This whole suite of distributed computing has no inherent mechanism to handle faults of the size and scope that occur in fixed wireless networks. Even in wired networks, it is impossible to achieve 100% reliability. Wireless networks fare much worse; environmental conditions as common as rain or fog can impede communication channels. Faults, or *trouble spots*, last an in-determinant amount of time, typically measured in seconds as opposed to milliseconds. Thus, for legacy software to operate on a wireless network it will need some mechanism for handling trouble spots.

2 Ricochet Radio Networks

Ricochet Radio Networks are micro-cellular based networks in metropolitan areas. Small (shoe box) sized radios are scattered throughout a city atop utility poles at approximately 5 per square mile. These radios have effective coverage areas that overlap creating nearly complete coverage. Environmental features such as buildings and

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 395–403, 2001.

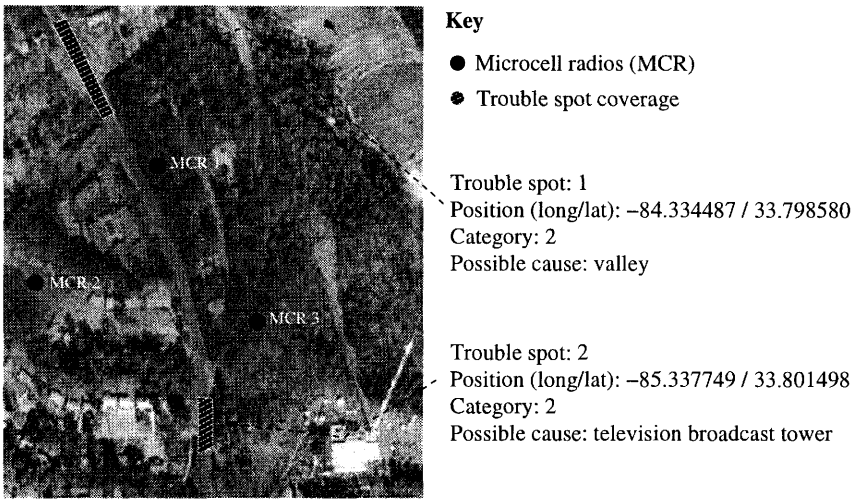

Fig. 1. Typical Ricochet Network and exemplar Trouble spots

bridges will create shadows within which coverage lacks. Further, areas without adequate number of pole-top radios will obviously lack coverage. Poletops relay data to Wireless Access Points (WAPs) which route these packets using a proprietary protocol to a Network Interface Facility (NIF) then to the Network Operations Center (NOC) and finally onto the Internet.

2.1 Trouble Spots

Trouble spots are localized and temporal. That is, a trouble spot exists at a particular position, for a particular amount of time (possibly infinite). (See fig. 1). It is important to note that there are two types of trouble spots; those that are anomalous, and those that result from lack of radio coverage. This later type are not faults per se, rather they are persistent characteristics of fixed wireless networks, while the former are atypical network failures (for example, packet loss due to interference from a passing school bus or from weather conditions.) In section 6 *persistent* trouble spot prediction is discussed.

3 Need for Application Level Awareness

It is desired that loosely connected nodes participating in a distributed computation be tolerant of trouble spots, insofar as trouble spots do not permanently disrupt (or indeed destroy) the distributed application. Ideally, a reliable protocol, such as TCP/IP [7], CCTL [14], or an extension thereof, could handle faults at the transport layer. However, implementing a reliable transport layer on fixed wireless networks may not be a solution. Disconnection times are possibly very long in wireless networks [15], so while the transport layer may still be working, the application above it may not be. However, there is significant information available to an application concerning connection quality. Thus, a useful protocol would handle indeterminantly long disconnections, as well as be aware of the quality of service provided by the network. These issues are of paramount interest to an application that runs *on top* of the transport layer.

Todays wireless networks are inherently faulty to such a degree that transport protocols alone cannot adequately address problems that arise in distributed applications. Therefore, distributed applications must act intelligently by managing network usage with regard to the networks current state. Our experience [11] with Ricochet Networks suggests that as much as 20% of the time a modem is in service is spent within trouble spots that last longer than one second. While most transport layer protocols can handle faults of this nature, they do not offer the application the ability to easily assess the situation (i.e. quantify such metrics as proposed in [12] and discussed in section 3.1. Thus, if necessary, (as is in Collaborative Computing Frameworks [1]), protocols for network control should be embedded within the application layer. To be useful, these protocols should offer an application knowledge about network connectivity. Next we examine how to determine quality of connection.

3.1 What Metrics Are Useful?

From our previous work in evaluating Ricochet Radio Networks [12] three metrics have been selected as useful when determining quality of connection. These are latency, packet loss, and signal strength. Latency and packet loss can each be calculated using datagram sockets and simple echo requests. The time period over which these statistics are generated should of course be variable to accommodate different applications. Signal strength is a quantity measured in decibels that in Ricochet Radio Networks typically ranges from -30 dB to -110dB. This information is available via queries sent to Ricochet modems AT commands. A modem gathers this data from all poletops it can detect.

3.2 Other Useful Information

Poletops provide other information [8] beyond signal strength such as their latitude and longitude, and their color. Among other things, color specifies whether or not a radio is a WAP (color is not an immediately useful property). Latitude and longitude can be used to give rough estimates of user position. The more radios are visible, the more accurate that estimation becomes. Temporal positional data can also be used to make estimates of the users heading. One can see how positional information is useful for connectivity predication as it relates to application level awareness.

We have discussed two general types of trouble spots in Ricochet Radio Networks, how to assess the severity of a trouble spot, and why it is necessary to offer this information in an application level interface. Next we look at RRNAPI, a toolkit to serve this task.

4 Ricochet Radio Network Application Program Interface

The Ricochet Radio Network Application Program Interface was developed in order to enable programs that wish to take intelligent action based on network performance. The API offers 21 functions (see appendix A). Programs instantiate RRNAPI by calling `RRN_init()`. This function must be called before any other RRNAPI function; `RRN_done()` is called when the application is finished with the API. The time from initialization to completion constitutes a 'session'. Depending on how the `mode` parameter is set the API spawns a thread that handles IO from the modem, the upper level API, the RRNAPI Server, and session files. This thread updates the client application

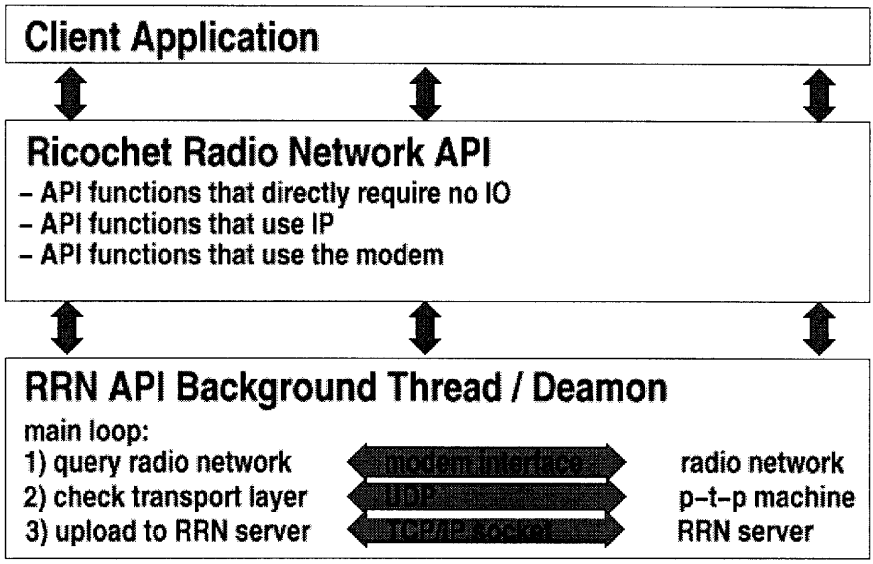

Fig. 2. Functional diagram for RRNAPI

at programmer specified intervals with 'snapshots' (see appendix B for a list of important data structures) of the current network statistics. These snapshots can be saved and loaded to and from disk or sent as updates to the RRNAPI Server.

Applications can choose the update threads behavior via the mode parameter to `RRN_init`. These flags (see fig. 3) can either update the server or not (`RRN_DO_UPDATE`), use the modem to query the network or get this information from a file (`RRN_USE_MODEM` - this is useful for developing the set of internal functions that communicate with the modem directly), continuously cache the session to disk (`RRN_RECORD`), attach to a modem that is online or off-line (`RRN_MODEM_MASTER`), and specify whether or not to perform the transport layer check using UDP (`RRN_CHECK_TL` - turning this off will render useless further calls to `RRN_packet_loss` and `RRN_latency`). The default initialization mode for `RRN_init` will update the RRN Server, open the modem off line, check the transport layer, and cache the session.

`RRN_CHECK_TL` enables the `rrn_check_udp_thread`. This thread uses ICMP echo request packets to estimate latency and packet loss. The time period over which these calculations are performed and the interval between checks is set by `RRN_config_tl_check`. This function also allows the application to set a latency threshold that specifies a maximum time period after which latency is considered to be infinity. This is useful for programs that require performance within set parameters. The default values are 10 ICMP packets over 10 seconds with a an infinite (`INT_MAX`) threshold. Error codes are set in the global rrn_error. The values of errors can be seen in figure 3. For a complete listing of RRNAPI see Appendix A.

5 Technical Notes

RRNAPI works only with Ricochet Radio Networks, and is therefore useful to only a small niche of wireless applications. While this may seem like a drawback at first, it is

```
RRN Error Codes:

#define RRN_CONNECT        1
#define RRN_NO_DIAL_TONE   (2   | RRN_ERROR)
#define RRN_NO_ANSWER      (4   | RRN_ERROR)
#define RRN_NO_CARRIER     (8   | RRN_ERROR)
#define RRN_RING           16
#define RRN_BUSY           (32  | RRN_ERROR)
#define RRN_IO_ERROR       (64  | RRN_ERROR)
#define RRN_INIT_ERROR     (128 | RRN_ERROR)
#define RRN_MODEM_ERROR    (256 | RRN_ERROR)
#define RRN_TIMEOUT_ERROR (512 | RRN_ERROR)

RRN Location Modes:

#define RNN_RSSI_MID       2
#define RNN_RSSI_NEAR      1
#define RNN_RSSI_AVG       0

RRN Initilization Modes:

#define RRN_DO_UPDATE      1
#define RRN_USE_MODEM      4
#define RRN_USE_STDIO      8
#define RRN_RECORD         16
#define RRN_MODEM_MASTER   32
#define RRN_MODEM_SLAVE    64
#define RRN_CHECK_TL       128
```

Fig. 3. RRNAPI Error Codes.

important to note that the API's usefulness stems from the ability to do three things: 1) evaluate transport layer performance, 2) measure signal strength, and 3) estimate client location. Both one and two above are provided by most wireless devices (for example, Sprint PCS [13], 802.11b [10] compliant cards, and CDPD [3]). However, point three is not provided nor accessible in most wireless devices. This is the only limitation to extending RRNAPI to a more general wireless device API. Gathering location information is trivial and accurate with the use of GPS.

The use of different devices, and indeed different modems of the same make, results in different data sets for the *same* areas in a network. Thus, data gathered from Metricom GS modems is not equivalent to data from Sierra Wireless modems or Novatel modems; nor are results from one Metricom GS the same as those of another Metricom GS. While the ability to generally model network performance is important, there is a need to examine these models in light of which hardware is in use.

5.1 Extending RRNAPI

If GPS devices become pervasive in mobile electronics, then extending RRNAPI to work on a broad range of devices is possible. One should note that the fore mentioned issue of differentiating between types of modem is more imperative when the difference is not between modems, but between entire technologies (e.g. the difference between wireless LAN and Ricochet are vast when compared to the differences between one make of Ricochet card and another). Thus, for a wireless API to be useful (in the sense that RRNAPI is useful) it must include location information.

RRNAPI is designed with a modular back end to support different modem hardware. Modem specific functions in RRNAPI are defined separately. Anyone wishing to implement RRNAPI with a modem other than the Ricochet GS serial modem will have to rewrite the functions in `modem.c` and compile it with `rrnapi.c`. Later, these internal functions will be dynamically loadable shared objects. Further, it may be useful to have the API attempt to detect the modem make of an interface and then load an appropriate function set.

Currently, RRNAPI requires Glib, a POSIX threads implementation, a Linux host, and a working Ricochet GS Serial Modem. Porting RRNAPI to other Unix platforms should be straightforward. Porting to Microsoft Windows will require some effort.

6 Future Work

The RRNAPI offers not only an interface for mobile clients to access network performance, but also the ability to query a remote database that contains past network performance. Ricochet Radio Network Daemons and Ricochet Radio Network Terminate Stay Resident (RRND's and RRNTSR's) could be run on client computers that continually update an RRN Server with network statistics. This server holds temporal data on the networks performance from the clients point of view. Given enough data it should be possible to create an accurate picture of network performance at some time in the future. This allows for the prediction of trouble spots. All programs that use the RRNAPI get their network data (packet loss, signal strength, etc.) via a thread that queries the modem and network at application specified intervals. There is also the option to have this thread send network performance data to the RRN Server. Such a database opens many possibilities for research. What techniques would be used to model performance? What accuracies are possible? Is it possible to predict temporal performance of the network? Can network load be detected? These questions require further investigation.

A Appendix: RRNAPI Listing

The RRNAPI is available from `http://vector.mathcs.emory.edu/rrnapi/`

`RRN_init()`	Input: modem device name, mode bits. Output: returns zero on success. Sets `rrn_error` otherwise. Must be called before the API is used. Connects to the modem and starts background IO threads.
`RRN_done(void)`	Input: void. Output: return zero on success. Sets `rrn_error` otherwise. Called to close the API.
`RRN_set_rrn_file()`	Input: string name of file Output: void Set file for session.
`RRN_load_rrn_file()`	Input: pointer to a linked list of snapshots. Output: same list. Loads previous session from file into a linked list of snapshots.

`RRN_save_rrn_file()`	Input: pointer to a linked list of snapshots. Output: same list. Saves session into file of snapshots.
`RRN_poletop_count()`	Input: a list of poletops. Output: how many poletops are on the list. If input is null, returns the number of poletops currently visible to the modem.
`RRN_rssi()`	Input: poletop list, calculation mode. Output: a radio signal strength indication (rssi). Uses a list of poletops to calculate rssi. `mode` specifies how to calculate the rssi; use `RRN_RSSI_AVG` for an average of all poletops, or `RNN_RSSI_NEAR` to get the strength of the closest (actually, this is the strongest, not necessarily the closest) poletop.
`RRN_connection_status()`	Input: void. Output: an error code.
`RRN_network_id()`	Input: void. Output: Ricochet Network Number.
`RRN_firmware_version()`	Input: void. Output: string. Returns the modems software version.
`RRN_hardware_version()`	Input: void. Output: string. Returns the modems hardware version.
`RRN_poletops()`	Input: void. Output: linked list of `RNN_poletop_t`'s. Returns a list of poletops sorted by location that are 'visible'.
`RRN_print_poletops()`	Input: list of `RRN_poletop_t`'s. Output: prints to stdout. Prints a list of `RRN_poletop_t`'s.
`RRN_set_update_timeout()`	Input: a time in milliseconds. Output: void. Set how often to query the modem, check the transport layer, and update the server.
`RRN_packet_loss()`	Input: void. Output: a percentage. See what percentage of ICMP echo request packets were lost over a set time period.
`RRN_latency(void)`	Input: void. Output: time in milliseconds. Check the average latency over a set time period.
`RRN_set_heading()`	Input: a `RRN_vector_t` pointer. Output: void. RRNAPI maintains a heading. This is a vector from the last location known to the current location with a velocity magnitude.

`RRN_location()`	Input: a mode. Output: a lat. long. location. Calculate likely user location based on radio locations and signal strength. mode 0 = weighted average of active poletops. (default). mode 1 = strongest, and most likely closest, poletop. mode 2 = center without respect to signal strength. These modes are `RRN_RSSI_AVG`, `RRN_RSSI_NEAR` and `RRN_RSSI_MID`.
`RRN_print_modem()`	Input: a RRN_modem_t. Output: void. Print modem information.
`RRN_modem_info()`	Input: void. Output: pointer to a RRN_modem_t. Get modem information. This does NOT query the modem, it just sees what `RRN_init` found when the API was started.
`RRN_config_tl_check()`	Input: number of ICMP packets to send, interval between packets, threshold for latency. Output: void. Establishes the behavior of `rrn_check_udp_thread`.

B Appendix: Data Structures

```
typedef struct _vector_t{
  double base_long;
  double base_lat;
  double mag_long;
  double mag_lat;
}RRN_vector_t;

typedef struct _poletop_t{
  double latitude;
  double longitude;
  int    strength;
  int    color;

}RRN_poletop_t;

typedef struct _udpinfo_t{
  unsigned int latency;
  unsigned int latency_threshold;
  float        packet_loss;
  unsigned int delta;
  unsigned int width;
} RRN_udpinfo_t;
```

```
typedef struct _snapshot_t{
  GList                 *poletop_list;
  struct timeval        *timestamp;
  RRN_udpinfo_t         *tl;
  RRN_vector_t          *heading;
} RRN_snapshot_t;
```

References

1. S. Chodrow, S. Cheung, P. Hutto, A. Krantz, P. Gray, T. Goddard, I. Rhee, and V. Sunderam. CCF: A Collaborative computing frameworks. In *IEEE Internet Computing*, January / February 2000.
2. J. Dongarra, A. Geist, J. Kohl, P. Papadopoulos, and V. Sunderam. Harness: Heterogeneous adaptable reconfigurable networked systems. In *High Performance Distributed Computing*, 1998.
3. Wireless Data Forum. Cdpd system specification release 1.1, 1998. URL: *http://www.wirelessdata.org/develop/cdpdspec/index.asp.*
4. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997. URL: *http://www.globus.org/research/papers.html.*
5. A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds Jr. A synopsis of the legion project. Technical Report CS-94-20, Department of Computer Science, University of Virginia, June 1996. URL: *http://www.cs.virginia.edu/ legion/CS-94-20.pdf.*
6. The Bluetooth Special Interest Group. The Official Bluetooth Website, 1999. URL: *http://www.bluetooth.com/developer/whitepaper/whitepaper.asp.*
7. E. A. Hall. *The Core Internet Protocols: The Definitive Gude.* O'Reilly & Associates, 2000.
8. Metricom Incorporated. Ricochet Technology Overview, 1999. URL: *http://www.ricochet.com/ricochet_advantage/tech_overview/.*
9. R. J. Loader and J. S. Pascoe. Future Directions of The CCF Project. Technical report, The University of Reading, Department of Computer Science, 2000. Available by request (in press).
10. IEEE White Paper. Ieee 802.11b standard. Web Page, 2000. URL: *http://www.wlana.com/learn/80211.htm.*
11. J. S. Pascoe, G. Sibley, V. S. Sunderam, and R. J. Loader. Mobile Wide Area Wireless Fault Tolerance. Technical report, University of Reading and Emory University, 2001.
12. J. S. Pascoe, G. Sibley, V. S. Sunderam, and R. J. Loader. Mobile Wide Area Wireless Fault Tolerance. Technical report, University of Reading and Emory University, 2001.
13. Sprint PCS. Sprint pcs developers forum, 2000. URL: *http://www.developer.sprintpcs.com/.*
14. I. Rhee, S. Cheung, P. Hutto, A. Krantz, and V. Sunderam. Group Communication Support for Distributed Collaboration Systems. In *Proc. Cluster Computing: Networks, Software Tools and Applications*, December 1998.
15. G. Sibley. Ricochet Network Personal Communications. Technical Report 1100-01, Department of Math and Computer Science, 2000. Emory University.
16. V. S. Sunderam. Pvm: A framework fo parallel distributed computing. *Concurrency, Practice and Experience*, December 1990.