On the implementation and performance of the (α, t) protocol on Linux

Anandha Gopalan
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, U.S.A
Email: axgopala@cs.pitt.edu

Sanjeev Dwivedi [*]
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, U.S.A
Email: sanjeev@cc.gatech.edu

Taieb Znati
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, U.S.A
Email: znati@cs.pitt.edu

Bruce McDonald [†]
Electrical and Computer Engineering Department
Northeastern University
Boston, MA 02115, U.S.A
Email: mcdonald@ece.neu.edu

Abstract

This paper details the design and implementation of the (α, t) protocol, a clustering and routing protocol for ad-hoc networks on Linux. Clustering and routing protocols that are developed, are normally tested using simulations. Without actual implementation, it is very difficult to perceive how efficient and effective the protocol would prove to be in the real world. The $(\alpha, t)-$Cluster framework deals with a unified approach to adapt dynamically to changing network topology. Nodes are organized into clusters depending on the ability to bound the probability of path failure due to node movement. This clustering scheme forms the basis for an adaptive routing strategy wherein routes within a cluster are maintained pro-actively and routes between clusters are managed re-actively. We conclude the paper by discussing an experimental study to evaluate the performance of the (α, t) protocol.

1. Introduction

The principal reasons for implementing wireless communications systems include support for terminal mobility [5], and more rapid, widespread access to communications services, without the need to construct and manage expensive cabling systems on the scale required for wired systems. In other words, the advantages of a wireless system are mobility, flexibility and cost savings. Most wireless networks currently in operation support untethered access for mobile communications devices by providing a wireless interface between the mobile devices and a fixed network of limited range *base-stations* (BS). Mobility is managed by allocating a limited set of communications frequency channels to each BS, and dynamically assigning a mobile device to a local channel as it moves from the coverage area of one BS to another. This infrastructured model is also referred to as *cellular* wireless communications.

The disadvantage of an infrastructured wireless network is that it requires a fixed infrastructure which constrains node mobility, thus limiting network deployability and increasing installation and management costs. To overcome these shortcomings, a class of infrastructureless wireless networks called ad-hoc networks emerged to fill the void. Ad-hoc networks offer an alternative to an infrastructured network whenever a fixed network is too expensive or infeasible to implement or less desirable due to cost, security or lack of flexibility.

It is often argued that in the present scenario, ad-hoc networks do not offer significant advantages because we do not have applications that can utilize them. [4] counters the argument by stating that, in the absence of infrastructure, the wireless devices themselves take on the functions yielded by them. Approaches towards infrastructure-less network solutions from large vendors like, Apple's rendezvous protocol [3], IETF's zeroconf protocols [25] and Bluetooth [9] are very important steps in this direction.

Routing in ad-hoc networks is a difficult problem and it involves a tradeoff between optimality and overhead. An ad-hoc network routing algorithm must be able to adapt

[*] The work was done while this author was a student at the University of Pittsburgh

[†] The (α, t) protocol was designed while this author was a student at the University of Pittsburgh

rapidly to topology changes to meet the performance demands of its users, without over-utilizing its resources. Clustering provides us with the means to balance the aforementioned tradeoffs. In a clustered ad-hoc network, the network is dynamically organized into partitions called *clusters* with the objective of maintaining a relatively stable and effective topology [15]. The membership and characteristics of each cluster may change dynamically over time in response to node mobility and is determined by the criteria specified by the clustering algorithm. Clustering in an ad-hoc network can be used to achieve several objectives, namely: support hierarchical routing, make the route search process for re-active routing protocols more efficient, support a hybrid routing strategy with different routing protocols that operate in different domains or different levels of the hierarchy.

Even though, theoretically, cluster-based protocols appear to be better, they have not been investigated much and as far as we know, no implementations of a cluster-based ad-hoc routing protocol is in existence. In the course of this research we have investigated a cluster-based protocol called the (α, t) protocol. The $(\alpha, t)-$Cluster framework is based on dynamic cluster organization, where the node mobility model is used to bound the probability of path failure over time. This effectively balances the competing demands for network resources and routing responsiveness. The (α, t) protocol has been designed with the features of both cluster-based and clusterless protocols and tries to avoid their shortcomings. At the same time, it introduces a new metric for cluster formation which allows intra-cluster routing to be more efficient. In this paper we have investigated, redesigned, implemented and analyzed the (α, t) protocol.

The (α, t) protocol might have far reaching significance towards QoS in ad-hoc networks, which has been largely non-existent from other implementations (schemes). This work is the first step towards the implementation of the (α, t) protocol and tries to build a proof of concept for the same.

This paper extends the work presented in [1] in the following areas: the protocol architecture, along with its different components and their interaction is elaborated; the clustering and routing algorithms of the $(\alpha, t)-$Cluster framework are explained in more detail; an extensive explanation of each module in the protocol implementation and the interaction between them is also provided.

The rest of the paper is organized as follows: Section 2 details the requirements of this implementation and the related work, Section 3 details the architecture used in this implementation, Section 4 talks in detail about the implementation, Section 5 talks about the experiments and results and Section 6 concludes the paper and identifies the areas for future work.

2. Requirements and Related work

A couple of requirements/system capabilities are required/called for in an efficient (and easy) implementation (all implementations thus far have implemented the following) [13]:

1. Finding out when a route is needed
2. Initiating a request
3. Queuing packets for an outstanding request
4. Re-injection of outstanding packet in the stream
5. Refreshing timers/validating routes

Ad-hoc networks have emerged in response to advances in hardware systems, availability of unlicensed radio spectrum, and frustrations over the costs and limitations of infrastructured wireless networks. As public cellular wireless system move into their third and fourth generations, wireless LANs have become important components of many corporate information infrastructures. Efforts have been under way to address many of the limitations of these emerging systems. Specification of the wireless MAC-layer protocol standard, IEEE 802.11, and the charter of the IETF MANET working group have reinforced the need for more flexible wireless networks, and thus a growing sub-field of wireless communications has taken hold, namely, wireless ad-hoc networking.

From the earliest adaptations of traditional distance vector routing proposed for the DSDV protocol [20] to sophisticated techniques that use information gathered from the GPS to report and estimate node position information for the purpose of efficiently building on-demand routes [14], the published work displays a wealth of varied and interesting techniques and ideas; however, a gap remains to be filled. Specifically, none of the schemes that have been proposed have been shown to perform well enough over a wide range of environments. Consequently, the question remains as to: "how to efficiently support routing that is responsive to a wide range of mobility patterns, and is scalable and one that can form the nucleus of a strategy capable of supporting QoS requirements in terms of throughput and delay?"

The problem of routing in wireless ad-hoc networks has motivated researchers and protocol designers to re-examine the basic tenets of adaptive routing as they have evolved over the past several decades. Challenges that were faced by early routing protocol designers, including limited bandwidth and unreliable communications links are being faced once again in the context of ad-hoc communications. However, in some ways the ad-hoc routing problem is more difficult. In particular, node mobility, asymmetric channel characteristics, and power constraints are added difficulties which must be addressed in order to implement a truly effective and commercially acceptable network architecture.

The structure of the Internet suggests that hierarchical routing is essential to achieve scalability. In ad-hoc networks, maintaining hierarchy (clusters) becomes more difficult due to the dynamic nature of the network. We believe that clustering can increase the scalability of ad-hoc networks by dividing the pro-active and re-active parts of the network into intra-cluster and inter-cluster domains.

Most of the literature on ad-hoc routing deals with re-active schemes. However, re-active schemes become extremely in-efficient when the network is subject to heavy traffic loads and high mobility. This leads us to the pro-active schemes. The main arguments against pro-active schemes are: periodic updates that requires bandwidth and processing, frequently using scarce resources to maintain routes that are seldom used.

As a result of the shortcomings present in both the re-active and pro-active protocols, it is apparent that a hybrid scheme is needed. Hybrid schemes contain the features of both these methods and hence can use a pro-active scheme for high mobility elements of the network while relatively immobile elements can communicate using re-active schemes.

The Destination sequenced distance vector DSDV routing protocol [20] is a pro-active routing protocol, where each routing entry is assigned a sequence number. This helps nodes to easily distinguish between old routes (one that is no longer valid) and a new routes. Ad-hoc on demand distance vector protocol (AODV) [19] is a re-active protocol, wherein routes are created and maintained as and when needed. When a source requests a route to a destination, the source broadcasts a route request message (RREQ). This request is re-broadcast by the other nodes until it reaches the destination. The destination on receipt of the RREQ message replies using a request reply (RREP) message, which is sent back to the sender using the reverse path that the RREQ took. Dynamic source routing (DSR) [12] is another re-active routing protocol. This protocol is very similar to AODV, but instead of re-broadcasting the request, nodes do a limited broadcast. A limited broadcast is when a node does not broadcast a request, but discards it if it has already processed the request. Zone routing protocol (ZRP) [10] is a hybrid routing protocol that divides the network topology into overlapping zones. Routing inside a zone uses the intra-zone routing protocol (IARP) and routing between zones uses a inter-zone routing protocol (IERP).

3. Protocol Architecture

The $(\alpha, t)-$Cluster framework supports a scalable routing infrastructure that is able to adapt to a changing network topology by dynamically organizing nodes into clusters and hence bounding the impact of routing overhead [16].

The $(\alpha, t)-$Cluster framework introduces a probabilistic metric to provide a bound on the availability of paths inside a cluster. This metric allows for the dynamic balancing of the trade offs according to temporal and spatial dynamics of the network. This is achieved by predicting the future state of the network links in order to provide a bound on the availability of paths inside a cluster. This well-defined metric captures the dynamics of node mobility. Using this metric, cluster organization can be made adaptive with respect to dynamically balancing the tradeoffs according to temporal and spatial dynamics of the network.

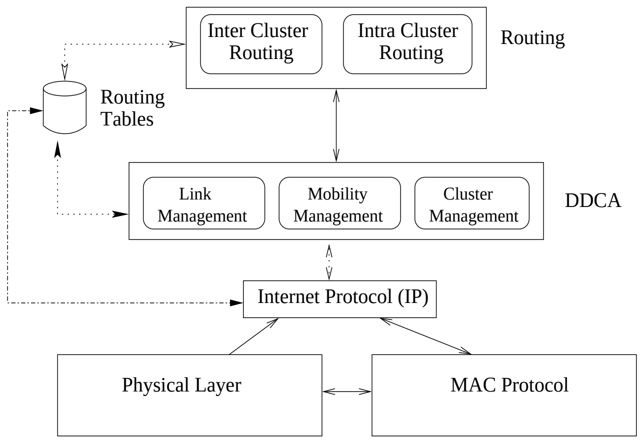

Figure 1. Overview of the design

The logical organization of the $(\alpha, t)-$Cluster framework is shown in Figure 1. The highest layer in the framework is the Internet Protocol (IP) that is responsible for creating IP packets and ensuring that they are sent out into the network towards their intended destination. If there is a route to the destination (information about the next hop), the IP packet is sent directly to the MAC layer, which in turn injects this packet into the network. If there does not exist any route to the destination, the packet is queued using the Netfilter framework (section 4.5) and the appropriate function that is registered is invoked to find a route to the destination. In case a route to the destination is found, this information is updated in the routing table.

The topological information regarding the next hop node for a packet is provided by the Intra and Inter-Cluster routing algorithms. These form the next layer in the framework and they rely on the clustering information provided by the Distributed Dynamic Clustering Algorithm (DDCA).

Intra cluster routing is done on a pro-active basis using a table driven pro-active routing algorithm. The $(\alpha, t)-$Cluster framework is flexible and independent of the specific intra-cluster routing algorithm and hence, any pro-active routing algorithm designed for ad-hoc networks can be used for routing within a cluster.

Inter cluster routing strategy tries to take advantage of the cluster topology and the intra-cluster routing tables. The

Inter Cluster Routing Protocol (ICRP) is a fully re-active cluster based routing protocol that discovers and maintains routes on an on-demand basis. In ICRP, the *parent* nodes (central coordinator for each cluster) of each cluster cooperate to control the route query process to avoid flooding the network.

DDCA is responsible for link management, mobility management and cluster management. DDCA logically partitions the network into clusters depending on the (α, t)-criteria for cluster formation. Each node in the network needs to be affiliated with a cluster. The information provided by DDCA is the basis for the routing protocols, since they differ based on whether a packet has to be routed within or outside a cluster.

At the lowest level in the framework are the physical and medium access control (MAC) layers that interact with each other. DDCA interacts with the physical layer in this framework to avail of the node-characteristics for calculating the link availability.

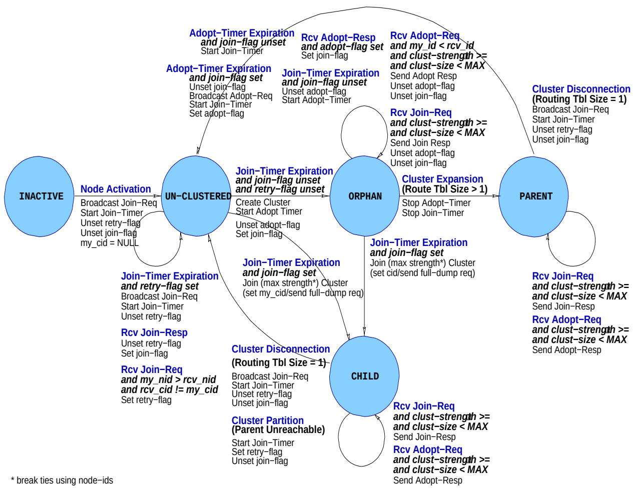

Figure 2. Description of DDCA events and states

3.1. DDCA Protocol Description

DDCA is an event driven algorithm which monitors the status of each node in order to maintain its cluster affiliation and current state. Each node can be in one of five states, namely, *inactive, un-clustered, orphan, child* and *parent*. The algorithm runs continuously and asynchronously on each active node in the ad-hoc network and forms the platform on which the intra-cluster protocol operates. DDCA provides ICRP with a clustered platform upon which it can operate the routing functionality.

The DDCA controls the cluster formation and using the set of states mentioned above, it provides the means for distributed control over the clustering process. A node cannot participate in routing until it is affiliated with a cluster and hence, as soon as any node becomes active, it tries to become part of a cluster. Once the node has associated itself with a cluster, the association is maintained until the node gets disconnected. A disconnected node tries to locate a feasible cluster; failing which, it forms a cluster of its own.

A scenario depicting the clustering decisions and other actions in DDCA is described briefly below. Each unclustered node seeks a feasible cluster by broadcasting a *join-request* message. If it receives no responses it creates a new cluster in which it is the only member, this type of a node is called an *orphan* node. To prevent adjacent un-clustered nodes from each creating new clusters, simultaneous requests are handled by forcing nodes with higher identifiers to back-off and try again. A node that receives at least one join-response message joins the maximum strength cluster from which a response was received. A node joins a cluster by changing its state, setting its cluster identifier (CID) and initiating an intra-cluster routing exchange with its neighbors. As a child, each node must process and respond to *join-request* messages and detect if it has become disconnected from the cluster, or if a cluster partition has occurred. The parent of every cluster is initially an orphan. Each orphan node periodically attempts to join an adjacent cluster until it detects that at least one child has joined its cluster. This can be detected by the reception of routing information and the subsequent increase in size of the intra-cluster routing table. Each parent node must process and respond to join-request messages and detect if it has become disconnected from its children. The complete set of events that can occur along with their interaction with the nodes in different states is given in Figure 2.

Routing inside a cluster is accomplished using a table driven pro-active routing protocol. The $(\alpha, t)-$Cluster framework is flexible and it can incorporate any pro-active table driven routing protocol like DSDV [20].

We have implemented intra-cluster routing by using a simple table driven protocol that keeps track of all the nodes in a cluster, and broadcasts messages to all these nodes when routing takes place. Each node, when it hears a *HELLO* from its neighbor adds that node to its list of *routable nodes*. Using *HELLO* messages, this list is updated periodically and stale entries are removed.

Another list, a list of *gateway nodes* is also maintained. Gateway nodes (also called border nodes) are nodes which are part of a cluster but can hear the broadcast from another cluster. These nodes play an important part in inter-cluster routing. When a node finds out that it is a gateway to another cluster, it broadcasts this to the other nodes in the cluster who update their list of *gateway nodes* appropriately. With the help of *HELLO* messages, this list is updated periodically and stale entries are removed.

DDCA guarantees that each node in a given cluster knows the addresses of all nodes currently affiliated within that cluster and the address of each external border node of that cluster.

3.2. ICRP Protocol Description

ICRP constructs routes on-demand and maintains them. Each node involved in routing maintains a cache of the nodes that it can reach via either intra-cluster or inter-cluster routing. The Inter Cluster route construction and maintenance protocol has four phases:

- Route Search
- Query Dissemination
- Route Setup
- Route Maintenance

Route search involves a query initiation by a source that requires a route to a destination that is neither in its cluster nor in its inter-cluster destination cache. The query messages are forwarded to all the gateway nodes of the cluster.

Query dissemination is the process by which the route search query is propagated through the network. Once a gateway node to a cluster receives a copy of the query, it first checks if it is a duplicate query. Duplicate queries are checked by first forwarding the query to the parent node and waiting for a reply. If the query is not a duplicate, it is again forwarded to all the gateway nodes of the cluster who again forward it to the other adjacent clusters. A cache is maintained regarding the reception of this query. Once a gateway node finds that the entry being requested belongs to its cluster it forwards the query directly to that node or if it itself is the object of the query, it starts processing the query.

Route setup is the phase when an actual route is setup between the source and the destination. Once the destination has been reached, the query terminates and no further queries are generated. The destination then updates its routing table, generates a query reply packet and sends it back to the node from which it received the query. This node in turn forwards the query back to the node from which it received the query. This continues until the originator of the query is reached. Once the query reply has been received by the originator, it updates its routing tables and the setup phase comes to an end. In case no reply is received within a timeout interval, the query is discarded from the cache.

Route maintenance is the process by which existing routes are maintained. In this phase each inter-cluster destination is checked periodically. Once a path remains inactive for time greater than a timeout value, the route is deleted. If a route is lost because the next-hop node is not available, a query is again initiated for that destination and a new path is setup if possible, from the point of disconnection.

4. Protocol Implementation

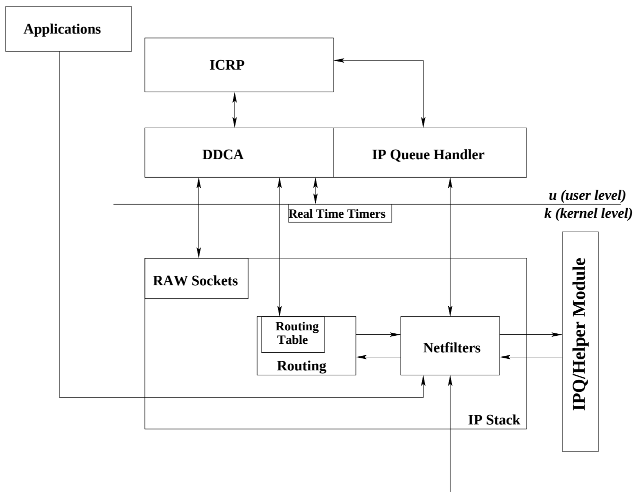

Figure 3. Architecture

A modular approach has been taken in this implementation in order to achieve a high degree of openness to potential changes in the individual components. The main components of the software was developed in C++ incorporating good Object Oriented Programming (OOP) techniques, except for some sections of the program which was written in C. The latter choice was motivated by the need to incorporate this component into the kernel (or link this component with the kernel at runtime).

The functional components of the protocol architecture are depicted in Figure 3. Each component has been implemented as a separate module and they export their interfaces for other modules to use to interact with this module. Each module has at least one global instantiation of itself, which can be used by other modules in the system to access the functionality of this module.

It was a design concern to keep this implementation as portable as possible. The implementation has been almost completely developed in user space. This allowed for a reduction in the overhead due to kernel interaction. Notice also, that as a result of this approach, debugging was made much easier. System dependent portions of the implementation were kept to a bare minimum (e.g: Timer functions, Kernel module). Porting this implementation to another platform just requires the system dependent modules to be re-implemented on that platform, while the rest of the implementation does not need to be changed.

A modular implementation allows for easier maintenance of the software. Due to the fact that modules interact with one another using the interfaces exported by them, any changes to the implementation of one module does not affect the other modules in the system.

The simple design on which the implementation is based is shown in Figure 3. Subsections 4.1-4.8 explain the functionalities of each component and the interactions between them.

4.1. Applications

Applications are the user level applications (usually applications built on top of the transport layer protocols like *telnet, ftp*) that want to initiate a connection and transfer data. These applications are unaware of the routing protocol or the underlying infrastructure that is being used. The interaction between the application and Netfilter (see section 4.5) is transparent to the application. Netfilter processes all the packets being generated by the application and passes these packets to the IPQ module. If a route exists to the destination, this packet is sent, otherwise, this packet is queued.

4.2. RAW Socket Interface

The raw socket interface provides the upper layer modules (DDCA and ICRP) with a platform to create and inject special type of packets into the network by bypassing the transport layer protocols (and to some extent, network layer protocols as well). This allows us to create our own packets with its own packet header and payload. This functionality enables us to create our own protocol type that is used by both *DDCA* and *ICRP*.

4.3. POSIX Real-Time Timers

The correct operation of DDCA depends on the timely delivery of the various timers in the clustering algorithm [16]. The granularity of the timer provided by the Linux kernel is not sufficient (officially Linux still does not have POSIX compliant Real-Time timers) for our purpose and also implementing multiple iterative timers in user space is not easy. This problem was solved by patching the Linux kernel with the POSIX 1003.1b compliant Real-Time timers patch [18].

4.4. DDCA

Distributed Dynamic Clustering Algorithm (DDCA) is the clustering algorithm on top of which the Inter Cluster Routing Protocol (ICRP) operates. DDCA creates the cluster and lets ICRP access the elements of the cluster through various interfaces that it exports. The interaction between ICRP and DDCA is limited to addition and removal of routing table entries from the kernel and ICRP receiving routing packets from DDCA.

DDCA uses RAW sockets to define a new protocol type (*IPPROTO_DDCA*) [2, 16], since it needs to create the IP packets, including the header by itself. DDCA also uses the Real-Time Timers functionality provided by the kernel for various timer based functions in the clustering algorithm. The DDCA module interacts with the kernel module using well-defined system calls that are listed in in Table 1.

Function	**Description**
add_gateway()	Adds the given gateway to the kernel list of gateway nodes.
remove_gateway()	Removes the given gateway from the kernel list of gateway nodes.
add_routable()	Adds the given routable node to the kernel list of routable nodes.
remove_routable()	Removes the given routable node from the kernel list of routable nodes.
change_state()	Changes the state of the node.
get_state()	Returns the current state of the node.
change_cid()	Changes the cluster id of the current node.
get_cid()	Returns the cluster id of the current node.

Table 1. Description of the system calls

4.5. Netfilter

Netfilter [23] is an architecture inside the kernel to filter packets based on various criteria. It defines *hooks*, which are well defined points in a packet's traversal through that protocol stack. At each of these points, the protocol will call the Netfilter framework with the packet and the hook number. The kernel module can then perform various operations on the packet before letting it pass further through the protocol stack. Additionally, it can also specify a verdict for the packet which can be one of *accept, reject, queue* or *steal*. If the kernel module has specified the verdict as *queue*, the IP packet queues up for processing by the IP Queue Handler module. Stealing the packet implies that the kernel does not bother about the packet anymore, it will be managed by the module that has registered to process it. Kernel modules that are once inserted into the kernel, become a part of the kernel and hence are able to access the *hooks* that are defined by the Netfilter framework. The *hooks* that are defined in IPv4 and illustrated in Figure 4 are:

1. NF_IP_PRE_ROUTING: Before an incoming packet is passed through the routing function.
2. NF_IP_LOCAL_IN: After an incoming packet has been passed through the routing function.

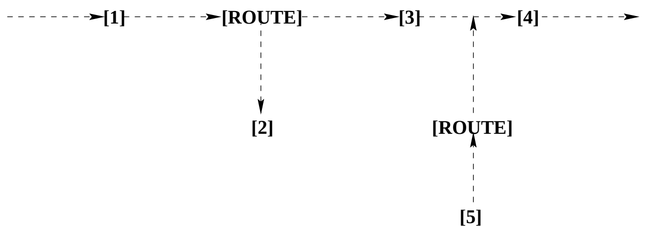

Figure 4. Packet traversal in the IP stack

3. NF_IP_FORWARD: After an incoming packet has passed through the routing function and is ready to be forwarded.
4. NF_IP_LOCAL_OUT: Before a locally originated packet passes through the routing function.
5. NF_IP_POST_ROUTING: After a packet has passed through the routing function.

The Netfilter interface is accessible at the user level through the $libipq$ library. This library can access the packets that are queued by the Netfilter based filters inside the kernel.

```
Algorithm 1: NF_IP_LOCAL_OUTPUT
Input: packet, interface
Output: Verdict
(1)   if a route exists
(2)       return Accept
(3)   else
(4)       return Queue
```

```
Algorithm 2: NF_IP_PRE_ROUTING
Input: packet, interface
Output: Verdict
(1)   if a route exists
(2)       return Accept
(3)   else if a connection to this desti-
      nation is still active
(4)       return Queue
(5)   else
(6)       return Discard
```

4.6. IPQ Helper Module

This kernel module is responsible for deciding which packets will be routed or dropped or forwarded. This module is inserted as part of the kernel before the (α, t) protocol starts to execute. The kernel module does not export any functions and all functions are internal to the kernel. Two functions, namely *output_handler* and *input_handler* are attached to the *Netfilter* interface via the *nf_register_hook* to handle outgoing and incoming packets respectively. The hooks that are registered for this implementation are: *NF_IP_LOCAL_OUTPUT* and *NF_IP_PRE_ROUTING*. The policy ingrained in these functions are given by the algorithms in algorithm 1 and algorithm 2 respectively.

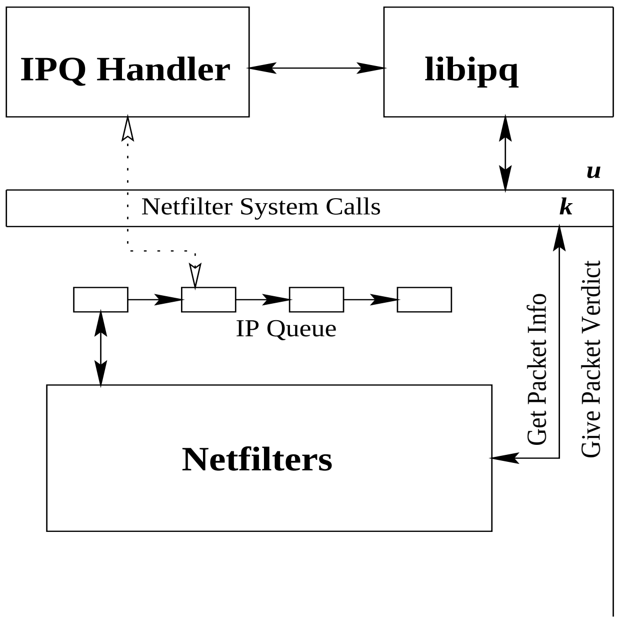

Figure 5. IPQ Handler and Netfilter interaction

The functions *init_module* and *cleanup_module* are also implemented in this module. The function *init_module* is invoked when the module is loaded into the kernel and this function switches on various protocol related activities. The function *cleanup_module* is invoked when the module is unloaded from the kernel and this function disables all the functionality related to the protocol and the system resumes its normal course of operation.

The routing algorithm implemented must be transparent to the user. Whenever a node tries to initiate a session with another node, for which there is no route available, the packets are queued and a search query is initiated. If the search succeeds before a timeout, the connection is established, otherwise a failure is returned. To make this operation transparent, the kernel module uses the interface provided by the Netfilter framework to register the functions that will be invoked when the specific hooks are encountered. Figure 5 details the interaction between the IPQ Handler and Netfilter using the $libipq$ library.

4.7. User Level Routing Table

After reading the implementation of *ospfd* in [17], it was decided to develop an interface to add and delete routing table entries from user space. A user level copy of the kernel level routing table is maintained as a queue. Any change made to the user level routing table is reflected in the kernel level routing table. The advantage of this scheme is that the kernel routing table need not be queried every time, only updates need to be sent to the kernel routing table. Routes are added and deleted from the kernel routing table by using the *ioctl* function call with the appropriate parameters and the *SIOCADDRT* and *SIOCDELRT* flags respectively. The routing table is maintained as a queue and each entry in the queue has the datatype *RTEntry*, which has the following structure:

```
typedef struct routingTableEntry
{
  unsigned long nid;
  unsigned long gw;
  unsigned long cid;
  unsigned long nextHop;
  int aliveSince;
}RTEntry;
```

The elements of the structure *RTEntry* are:

- nid: node id of the reachable node
- gw: node id of the gateway node associated with the node id: nid
- cid: cluster id of the reachable node
- nextHop: node if of the nextHop towards the reachable node
- aliveSince: indicates the time this node was alive

4.8. Packet Formats

This section lists the various packet formats used in this implementation.

Figure 6 shows the format of the DDCA Header packet. The important fields in this packet are:

- Protocol: Contains the type of the protocol that IP would use to identify the correct protocol stack to hand the packet over to. In our case, since there is no registered handler, IP looks for a RAW socket which has registered to receive a packet of this protocol type ($IPPROTO\_DDCA$).
- Source Network ID: Contains the ID of the node that initiated this packet.
- Source Cluster ID: Contains the cluster ID of the node that initiated this packet.

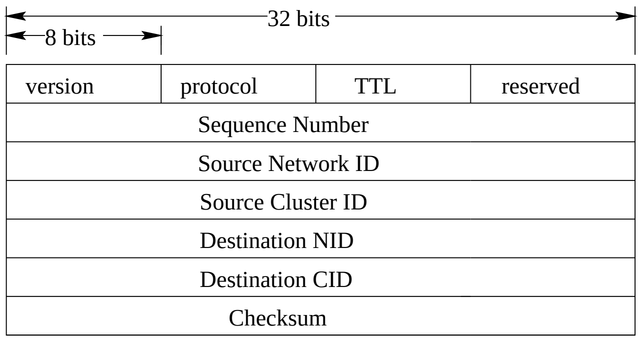

Figure 6. Format of the DDCA Header packet

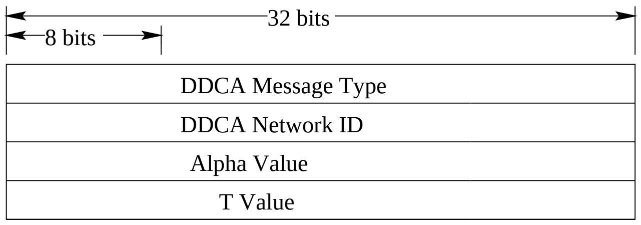

Figure 7. Format of the DDCA Body packet

Figure 7 shows the format of the DDCA Body packet. The important fields in this packet are:

- DDCA Message Type: Contains the message type to be sent. This is used by DDCA for its functioning.
- Alpha Value: Contains the value of α, the parameter used along with the value of t to test the link strength.
- T Value: Contains the value of t, the parameter used along with the value of α to test the link strength.

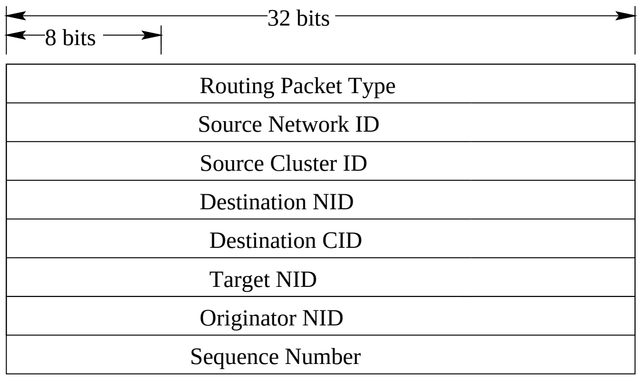

Figure 8. Format of the Query Body packet

Figure 8 shows the format of the Query Body packet. Some of the fields in this packet are similar to the fields in the DDCA Header packet (figure 6). The other important fields in this packet are:

- Routing Packet Type: Contains the type of the query, either RREQ (Route Request) or RREP (Route Reply).
- Target NID: Contains the NID of the node for which we need to discover a route.
- Originator NID: Contains the NID of the node where this query packet originated from.

4.9. User Interface to the protocol

For debugging purposes, it was important to have a method by which to access the status of the protocol in the kernel. Code in the kernel cannot be easily debugged with the help of a debugger. Tracing of code inside the kernel is also very difficult, since it is a set of functionalities not related to a specific process. Errors in the kernel are very difficult to track down and reproduce and these errors may result in a system crash, thus destroying much of the evidence that could have been used to identify the error. To ensure knowledge of the functioning of the protocol, we had two ways to access the information relevant to our protocol in the kernel.

4.9.1 Using *printk()*

printk is analogous to using a *printf* function at user level. The *printk* function call also takes in an optional argument, which is the *priority level*, that determines the emergency of the print statement. A high priority statement is printed on the system console (usually such statements are used for system emergencies, like hardware failures) while lower priority statements are used for purposes like devices failing to initialize and network failures.

4.9.2 Using the $proc$ file system

The massive use of *printk* statements to debug kernel code however, has one major drawback. It slows down the system considerably due to the fact that the *syslogd* daemon syncs its output files, thus causing a disk operation for every line that is printed. Another problem with using this scheme is that an error could cause a lot of messages to be printed on the console, thus making debugging harder.

It would be advantageous to be able to query the system and request the data that is relevant to the protocol, rather than continually producing output. The $proc$ filesystem offers such an alternative.

The $proc$ filesystem is a special, software created filesystem that is used by the kernel to export information to the world. Each file under the $proc$ filesystem is tied to a kernel function that generates the contents of the file on the fly when the file is read.

To create a file in the $proc$ filesystem [21], the kernel module implemented should implement a kernel function

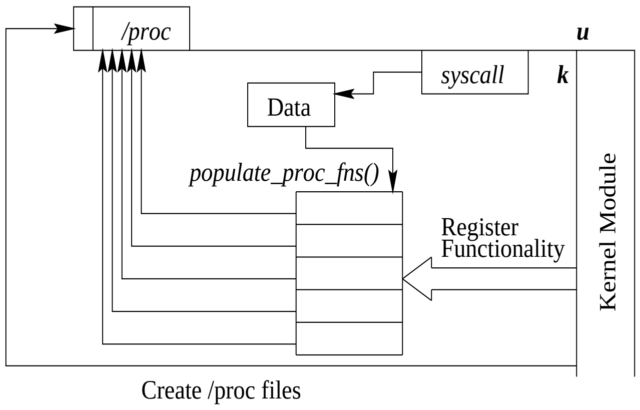

Figure 9. Module to create */proc* **entries**

Function	Description
show_gateways()	Prints all the gateway nodes for the current node in the decimal-dot notation.
show_routables()	Prints all the routable nodes for the current node in the decimal-dot notation.

Table 2. Utilities that can be used to observe the state of the system

to generate the data when requested. This function in turn must be registered to produce the data whenever that file is read.

When the kernel module is inserted into the kernel, it switches on the $(\alpha, t)-$Cluster functionality in the kernel and makes the system call functional. It also allows us to collect the kernel data that is relevant to our protocol by creating the $/proc$ files and also registering the respective functions to populate these files.

In particular, the following files are created in the $proc$ filesystem:

- */proc/at_routing/at_routable*
- */proc/at_routing/at_gateways*
- */proc/at_routing/at_state*
- */proc/at_routing/at_cid*

Table 2 highlights the utilities that provide an easy interface to the kernel information available through the $proc$ filesystem. These functions are used in conjunction with running the protocol to observe the state of the system.

5. Results

To test a hierarchical protocol like the $(\alpha, t)-$Cluster, we have to test both aspects of the protocol, the intra-cluster as well as the inter-cluster part of it. The experimental setup consisted of three nodes which were used to form a cluster (for intra-cluster testing) and two nodes used to form a cluster and another node acting as an orphan (for inter-cluster testing). Three different mobility models were studied.

- Static: The nodes are stationary.
- Group Mobility: The nodes move in groups, the rate of disconnections is not very high.
- High Mobility: The nodes are constantly on the move, leading to a very high rate of disconnections.

Experiments were conducted for the above three mobility models and two types of data analysis were studied.

- Goodput Analysis: This study measures the amount of data that had to be re-transmitted in order for the whole data to go through (e.g: A file).
- Data Rate Analysis: This study measures the sustained throughput that is achieved in the network.

The experiments were conducted with one node being stationary, while the other two nodes were moved around based on the mobility model (the mobility that was used was random mobility, where the nodes move as they want to, for example, laptops used in the experiment, were just carried around the department by the researchers conducting the experiment). The breaking point of the network was found (this was the point where inter and intra-cluster routes were being broken), and this knowledge was used in the group and high mobility models. During the experiments, the nodes were assigned different IP addresses that belonged to different domains. This was to make sure that the protocol worked irrespective of whether the nodes belonged to the same network or not (this is to mirror real life scenarios wherein, ad-hoc networks consist of nodes belonging to different networks).

5.1 System specifications

The nodes used in the experiments were Dell Latitude Laptops running Red Hat Linux (ver. 7.2). The Kernel version was 2.4.5 and the following modifications were added to the kernel:

- Kernel was patched with POSIX 1003.1b compliant Real Time Timers [18].
- Kernel was compiled with support for Netfilter.
- System calls were added to enable communication between the user level process and the kernel level module.
- PCMCIA package was added to be able to use the correct drivers for the wireless cards and to be able to utilize them in the ad-hoc mode [11].
- Iptables package was added for the *libipq* library which provides an interface for accessing packets queued by the kernel module [23].

The results were obtained by letting the nodes cluster and then by moving them around physically according to the mobility model. A file transfer was initiated between two nodes, one designated as the *sender*, the node responsible for sending the file, and the other designated as the *receiver*, the node responsible for receiving the file and also for logging the statistics. The size of the file transferred was varied from a very small file to a large file. The file to be transferred was broken down into packets of size 1500 bytes each.

The *sender* node is equipped with a *server* program, which is responsible for sending the file across. This program has a timer associated with it to calculate the time taken to send the file. The file is transferred continuously without any timeouts (regardless of the state of the connection). The *receiver* node is equipped with a *client* program, which is responsible for receiving the file. This program is also responsible for maintaining a count of the number of packets received and also the size of the packets received (this is to ensure that packets were not corrupted and wrong packets not received). The *client* does not sent acknowledgments for the packets that are received. Due to the frequent disconnections that can occur in ad-hoc networks, both the *server* and the *client* programs were written using RAW sockets. This does not use the services provided by the transport layer like acknowledgments and timeouts. This ensures that even if the connectivity between nodes is lost, the application does not time out.

5.2. Intra-Cluster evaluation

From Figure 10, we can see that the goodput in the static case is 100%. This is to be expected as the nodes are stationary and hence there is continuous connectivity. We can also conclude that the protocol is very tolerant to disconnections. Even as the file size increases (in the order of tens of megabytes), the protocol is able to maintain a good performance by ensuring that connections are restored quickly after a disconnection. The high mobility of the nodes helps in re-forming routes that are broken quickly due to the presence of other nodes that take the place of the previously disconnected node.

From Figure 11, we can conclude that the throughput remains constant in the static case. This is because the nodes

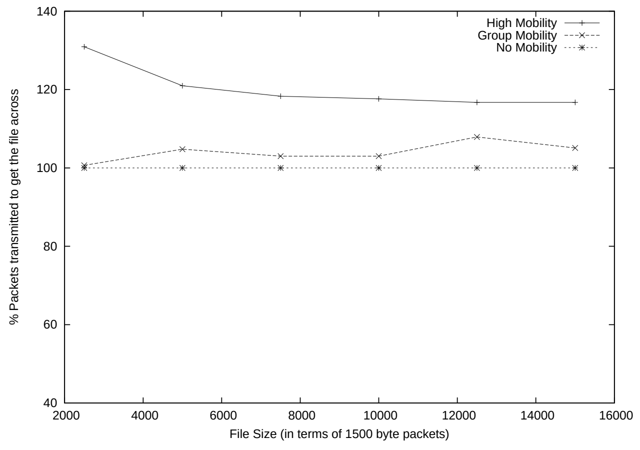

Figure 10. Intra-Cluster: Goodput Analysis

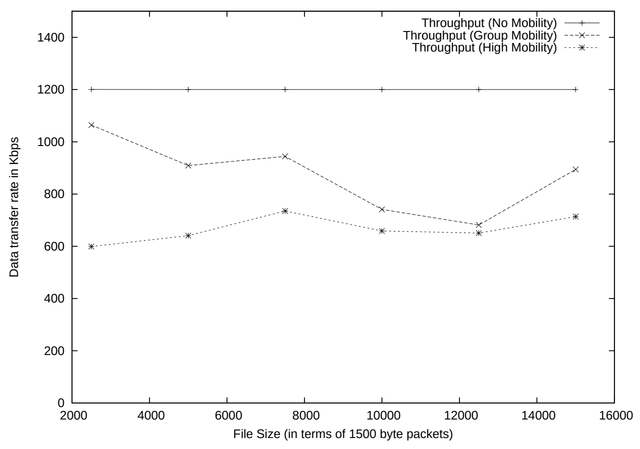

Figure 11. Intra-Cluster: Throughput Analysis

are stationary and connectivity is maintained throughout. The throughput falls significantly once disconnections are introduced into the network. There is a significant drop in the throughput in the high mobility case, but due to the fact that the protocol handles disconnections quickly, the throughput remains stable across different file sizes. This is very useful for applications that require a steady throughput (even though it is much lower than in the static case).

5.3. Inter-Cluster evaluation

From Figure 12, we conclude that the goodput in the static case is 100%. This does not come as a surprise as the nodes are stationary and hence have connectivity at all times. The goodput in the high mobility case has gone down considerable due to the increase in the number of disconnections, but we can see that the the protocol tolerates disconnections well and restores the connection soon. Even though the file sizes increase, thus giving a larger window

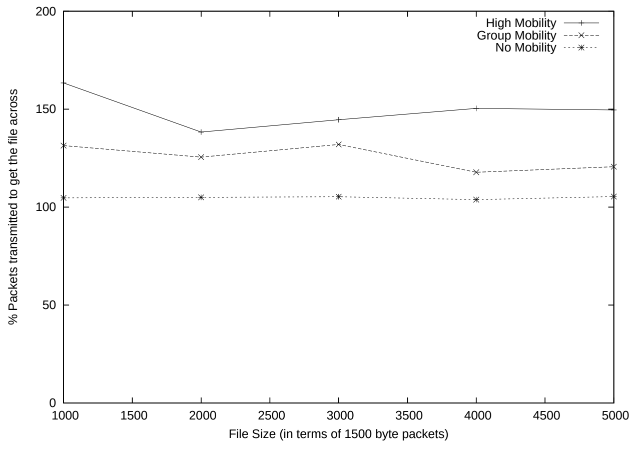

Figure 12. Inter-Cluster: Goodput Analysis

for disconnections to take place and hence lower the goodput, the protocol maintains a steady goodput.

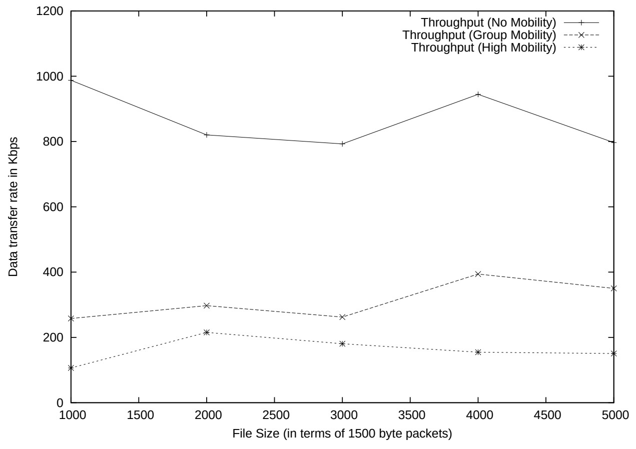

Figure 13. Inter-Cluster: Throughput Analysis

From Figure 13, we can conclude that the throughput for the static case averages well. The small dips and peaks can be attributed to transient environmental factors. When group mobility is introduced, we see that the data rate falls significantly, but the protocol handles these disconnections well, as can be seen from the fact that the data rate is quite stable. In the high mobility case, the data rate falls significantly due to a high number of disconnections, but overall the data rate remains stable.

6. Conclusions

The $(\alpha, t)-$Cluster framework was implemented on Linux and it was tested and evaluated. We believe this to be one of the few protocols for ad-hoc networks to have a proof-of-concept work. There is much work that still needs to be done in terms of tests, modifications and optimizations to make this protocol worthy of industry standards. The challenge is to be able to deploy MANET protocols in the industry so as to be able to test them out more rigorously which would help in improving the protocol.

We have to determine the overhead caused because of running DDCA continuously on these nodes. Careful analysis could determine how this protocol could be optimized and help reduce the number of packets transmitted, and also fix the optimum timeouts for the timers used.

Hybrid routing must be looked at in a whole new light when we talk about routing in Ultra-large scale networks and wireless sensor networks. Better models of routing need to be studied, models like data diffusion, content-based routing and information dissemination could be implemented and analyzed.

7. Acknowledgments

This work has been supported in part by the National Science Foundation (under grant no: 000073972).

References

[1] A. Gopalan, S. Dwivedi, T. Znati and A. B. McDonald. On the implementation of the (α, t)-Cluster Protocol on Linux. In *Proc. 37th Annual Simulation Symposium*, Apr. 2004.

[2] A.B. McDonald, T. Znati, A. Gopalan. (α, t) protocol specification. *Internet Draft*, August 2001.

[3] Apple Computer Inc. Rendezvou's Developer Page. *http://developer.apple.com/macosx/rendezvous/*, 2002.

[4] Charles E. Perkins. Ad-hoc networks. *Addison Wesley*, 2001.

[5] Committee on Evolution of Untethered Communications et al. *Evolution of Untethered Communications*. National Academy Press, 1997.

[6] W. P. I. Computer Science Department. Adding a system call. *http://fossil.wpi.edu/docs/howto_add_systemcall.html*, 2002.

[7] A. Demenshin. Demonstration of libipq usage. *http://aldem.net/netfilter/*, September 2001.

[8] H. Glenn. Linux IP Networking: A guide to the implementation and modification of the Linux Protocol Stack. *Masters Thesis http://www.cs.unh.edu/cnrg/gherrin/linux-net.html*, May. 2000.

[9] J. C. Haarsten. The Bluetooth Radio System. *IEEE Personal Communications Magazine, pp. 28-36*, February 2000.

[10] Z. J. Haas and M. Pearlman. The Zone Routing Protocol (ZRP) for Ad Hoc Networks. *Internet Draft*, August 1998.

[11] Jean Tourrilhes et al. MPL/GPL drivers for the Wavelan IEEE/Orinoco and others. *www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Orinoco.html*, April 2001.

[12] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad hoc Wireless Networks. In *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[13] V. Kawadia, Y. Zhang, and B. Gupta. System services for implementing ad hoc routing protocols. *International Workshop on Ad Hoc Networking*, 2002.

[14] Y. Ko and N. Vaidya. Location-Aided Routing (LAR) in Mobile Ad-Hoc Networks. In *Proc. ACM/IEEE MOBICOM*, Oct. 1998.

[15] C. R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 15(7), Sep. 1997.

[16] A. McDonald and T. Znati. A Mobility Based Framework for Adaptive Clustering in Wireless Ad-Hoc Networks. *IEEE Journal on Selected Areas in Communications (J-Sac), Special Issue on Ad-Hoc Networks*, 17(8), Aug. 1999.

[17] J. T. Moy. OSPF Complete Implementation. *Addison Wesley*, September 2000.

[18] S. Pather. POSIX 1003.1b timer patches for Linux. *http://www.rhdv.cistron.nl/posix.html*, August 2001.

[19] C. Perkins and E. Royer. Ad Hoc On-Demand Distance Vector Routing. In *Proceedings 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, 1999.

[20] C. R. Perkins and P. Bhagwat. Highly Dynamic Destination Sequenced Distance Vector Routing (DSDV) for Mobile Computers. In *ACM SIGCOMM*, pages 234–244, Oct. 1994.

[21] Rubini A. and Corbet J. Linux Device Drivers, 2nd Edition. *O'Reilly*, June. 2001.

[22] R. Rusty. Unreliable Guide to Hacking the Linux Kernel. *http://people.netfilter.org/ rusty/unreliable-guides/kernel-hacking/lk-hacking-guide.htmlz*, 1999.

[23] R. Rusty. Linux Netfilter Hacking HOWTO. *http://netfilter.samba.org*, 2002.

[24] H. Welte. The journey of a packet through the Linux 2.4 network stack. *http://www.gnumonks.org/ftp/pub/doc/packet-journey-2.4.html*, October 2000.

[25] A. Williams. Zero Configuration Networking. *Internet Draft*, September 2002.