Paradis-Net

A Network Interface for Parallel and Distributed Applications

Guido Malpohl and Florin Isailă

Institute for Program Structures and Data Organization, University of Karlsruhe, 76128 Karlsruhe, Germany Malpohl@ipd.uka.de, Florin@ipd.uka.de

Abstract. This paper describes Paradis-Net, a typed event-driven message-passing interface for designing distributed systems. Paradis-Net facilitates the development of both peer-to-peer and client-server architectures through a mechanism called "Cooperation". We introduce the programming interface and compare its mechanisms to active messages and remote procedure calls. Finally we demonstrate how the interface can be used to implement communication patterns typical for distributed systems and how peer-to-peer functionality can be mapped onto Paradis-Net.

1 Introduction

There is a growing interest in large-scale distributed systems consisting of a large number of cooperative nodes. Cluster computing, peer-to-peer-systems, grid computing and ad-hoc networks are examples of current active research areas. All these directions have in common the development of complex communication protocols. The architecture of these systems has recently shifted from the traditional client-server paradigm to decentralized cooperative peer-to-peer models and towards hybrid approaches combining both client-server and peer-to-peer paradigms.

Paradis-Net is a typed message-passing interface for distributed applications and operating system services. It is suitable for designing distributed systems for both high speed networks (e.g. Myrinet [1] or Infiniband [2]) or for relatively slow transport mediums (Internet). Paradis-Net offers a simple interface facilitating the implementation of complex communication patterns and abstracting away from particular network hardware.

Paradis-Net emerged from our experience in developing the Clusterfile parallel file system [3, 4] and addresses the problems of communication paradigms such as RPC and active messages, that have been described by the developers of *xFS* [5]. In a paper describing their experience with *xFS* [6], the authors identify the mismatch between the service they are providing and the available interfaces as a main source of implementation difficulties. We show in section 2 how our work on Paradis-Net addresses these issues.

The contributions of this paper are:

– Paradis-Net offers a low-level *transport independent* interface (for both, *user and kernel space*).
– Multi-party protocols are supported through the *Cooperation* mechanism, allowing several nodes to collaborate in order to serve a request.
– A Paradis-Net message handler can delegate the request to a remote handler. This mechanism is similar to *continuation passing*.

In addition to these points there are several notable details about the Paradis-Net interface:

– Paradis-Net and its Cooperation and continuation passing mechanisms match the needs of P2P overlay networks. (see section 4.2)
– The semantics of Cooperations suit the requirements of the RDMA protocol, which consists of memory registration and effective data transfer. Therefore, the low-level RDMA mechanism can be transparently exposed to the applications. (see section 4.3)
– For efficient utilization of SMPs, Paradis-Net has been designed as a multi-threaded library and offers a thread-safe implementation.

We have implemented Paradis-Net on top of TCP/IP sockets in user-level and in kernel-level. A user-level implementation for the Virtual Interface Architecture (VIA [7]) demonstrates that RDMA can be used transparently.

2 Related Work

Active messages (AM [8]) is a low-level message passing communication interface. An AM transaction consists of a pair of request and reply messages. Each request activates a handler associated with the message, that extracts the data from the network and delivers it to the application. The low-level interface of AM allows exposing the features of Network Intelligent Card (NIC) such as zero-copy RDMA to the applications. Paradis-Net can also map low-level NIC intelligence on the *Cooperation* communication abstraction (see section 4.3).

Remote Procedure Calls (RPC [9]) are a common standard for distributed client-server applications, e.g. NFS [10]. Both AM and RPC paradigms are suitable for client-server application due to their point-to-point request-reply nature. However, Wang et al. [6] found them unnatural for the multi-party communication needed by a peer-to-peer system: A point-to-point RPC call has to be followed by a reply from the liable peer. If the request is delegated to an another peer, the reply has to travel back on the same way, as shown in figure 1. On the other hand, Paradis-Net through its continuation passing mechanism, allows a direct reply from the last peer as depicted in figure 2. In general, for n delegations, RPC needs $2n$ messages, Paradis-Net only $n + 1$.

The Parallel Virtual Machine (PVM [11]) and the Message Passing Interface (MPI [12]) are used to specify the communication between a set of processes forming a concurrent program. With many communication and synchronization primitives they target the development of *parallel* applications following the SPMD paradigm and are not well suited for distributed system development.

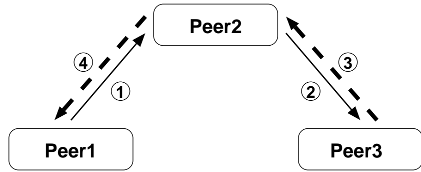

Fig. 1. Delegation through RPC

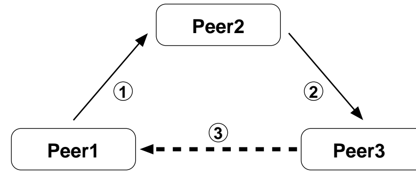

Fig. 2. Delegation in Paradis-Net

Solid arrow: Request – Dashed Arrow: Reply

General	
int initialize (*end_point* ep [])	section 3.1
void finalize ()	section 3.1
peer_id get_peer_id (**char** name[])	section 3.1
Communication	
int send(*peer_id* to, *msg_type* type, *coop_nr* nr, **void** ∗msg, **int** msg_size)	section 3.1
int forward(*peer_id* to, *msg_type* type, **void** ∗msg, **int** msg_size)	section 3.2
Handlers	
void set_handler(*msg_type* msg, **int** opt, *handler_fun* ∗handler)	section 3.2
void <*handler_fun*>(*peer_id* from, *msg_type* type, *coop_nr* nr, **void** ∗ msg, **int** msg_size)	section 3.2
Cooperations	
coop_nr start_cooperation(*rcv_desc* ∗rcvec, **int** vec_size)	section 3.3
int end_cooperation(*coop_nr* nr, **int** timeout)	section 3.3

Table 1. The Paradis-Net API

3 The Paradis-Net Architecture

Paradis-Net offers a peer-to-peer communication model in which every communication endpoint can be a server and a client at the same time. The Paradis-Net library is a layer between the application and the native network interface. Applications see a simple, uniform interface that is independent of the actual network technology used, thus easing the development of complex distributed protocols. Paradis-Net can be extended to support different network technologies. For a more detailed description of the API see our technical report on Paradis-Net [13].

3.1 Initialization, Peer IDs and Sending Data

The *initialize* method (see table 1) is needed to initialize the internal data structures and to open local endpoints. The *ep* argument contains the configuration options for the different network interfaces supported on this peer. On function return, the peer can be contacted by others. The inverse operation *finalize* closes down all the endpoints and releases the corresponding data structures.

Every Paradis-Net endpoint has a unique peer name expressed as an array of characters. This name usually consists of protocol and address information used to address other peers and to send messages to them. For example a TCP endpoint name is expressed as follows: *"tcp:<ip-address>:<service-port>"*. For convenience and performance reasons *get_peer_id* (see table 1) returns peer IDs that help refering to remote endpoints through handles rather then full endpoint names.

The Paradis-Net library offers only one explicit communication primitive: *send* (see table 1). This operation sends a typed message to the peer represented by the peer ID (*to*). *send* returns zero on success or an error number otherwise. Upon operation return the memory area that contains the message can be reused immediately.

3.2 Request Handlers

Paradis-Net does not offer a function to receive data from other peers. Instead, it uses handler functions that are called upon the arrival of a message. A handler function for a certain message type is set using the *set_handler* (see table 1) function. This event-driven mechanism requires an agreement between peers with respect to the message type they use. In contrast to Active Messages, Paradis-Net handlers are not limited in their execution time and can initiate calls to the library, including arbitrary *send* operations.

Handlers facilitate server implementation: Request reception and invocation of the appropriate user-defined handler is being taken care of by the library. A handler usually fulfills the request service and sends the reply back to the client.

The traditional client-server model can be expanded in Paradis-Net with the *forward* (see table 1) function. When called from within a handler, *forward* allows sending the message to a different peer and thereby also delegating the obligation to answer. The handler on the next peer will be invoked with the local *peer_id* that corresponds to the peer which was the original source of the request. Section 4.1 will give an example communication pattern the uses this operation.

3.3 Cooperations

The handler concept on its own is not convenient for the implementation of protocols involving several peers. For this reason Paradis-Net introduces "Cooperations". A Cooperation is a concept that defines a relationship between the outgoing requests and the incoming answers by creating a token that accompanies all involved messages. The function *start_cooperation* (see table 1) registers a Cooperation on the client side. The parameter (*rcvec*) describes the expected reply.

We will exemplify the life-cycle of a typical Cooperation in a client-server scenario: A Cooperation starts at the client by having the function *start_cooperation* registering the Cooperation and returning a token that represents the Cooperation. A Cooperation is registered using a *receive descriptor* containing memory

locations to store the replies. Receive descriptors also define criteria to distinguish between different message types and origins. Next, the client will send a request to the server and afterwards call *end_cooperation* (see table 1). This function blocks until the expected result is available. The token accompanies the request message on its way to the server by using the optional parameter of the *send* operation that attaches the token to the message. On the servicing peer, Paradis-Net will invoke the handler that has been assigned to the message type with the Cooperation token as a parameter (see the signature of handler functions in table 1). When the reply is sent, the Cooperation token is again attached to the message and travels back to its origin. On the client site, Paradis-Net identifies the reply as being part of a Cooperation by the type of the message. Although there is still the possibility to invoke a handler function upon the arrival of such a message, the library will first check if the attached cooperation token matches any of the currently active Cooperations on this peer. If this is the case, the service thread will store the message at the memory location that was declared when calling *start_cooperation* and afterwards wake up the thread that is waiting for the cooperation to finish.

4 Applications

In this section we will illustrate flexibility and simplicity of Paradis-Net in parallel and distributed file systems and P2P systems as well as its ability to transparently support remote zero-copy operations (RDMA).

4.1 Parallel and distributed file systems

Here are two examples that stem from our own experience developing the parallel file system *Clusterfile* [3] and the observations that were made building the distributed file system *xFS* [6].

Delegation. *xFS* illustrates the delegation pattern with respect to cooperative caching (see figure 2): A client (*Peer1*) reading from a file incurs a read miss in the local cache and sends a block request to the cache manager (*Peer2*) in order to retrieve the cached data from a different peer. The manager consults its map, finds the responsible cache server (*Peer3*) and forwards the request to it. The cache server then responds back to the client with the cached data. The same pattern can be also be employed for routing in a peer-to-peer system (section 4.2). Although this scenario appears to be simple, it is difficult to realize with traditional message passing interfaces. Wang et al. [6] demonstrate that RPCs are unsuitable to implement it because of the strict semantic imposed by the model.

On *Peer1* the method *start* registers a Cooperation, sends the request to a peer (*Peer2*) and waits for an answer.

```
void start(peer_id to, void *msg, int msg_len) {
  coop_nr coop;
  rcv_desc desc =                // receive descriptor
    { memory: NULL, size: 0,     // allocate memory for reply automatically
      type: REPLY,               // only accept messages with type "REPLY"
      options: RCV_FROM_ANY }; // accept messages from any peer

  coop = start_cooperation(&desc, 1);          // register cooperation
  send(to, REQUEST, coop, msg, msg_len);  // send request
   ...                                          // eventual computation
  end_cooperation(coop, 0);                    // wait for reply (no timeout)
}
```

Listing 1.1. Sending the request

The *start* function can be used for typical client-server communication as well. If *Peer2* would answer the request directly, *start* does not have to be changed at all, since the receive descriptor accepts replies from any peer, as long as the reply carries the Cooperation token issued by the local Paradis-Net library.

When *Peer2* receives the request from *Peer1*, Paradis-Net invokes the handler function *forward_handler*, which has been registered for messages with the type *REQUEST*. The handler first inspects the incoming message to find the peer responsible for answering the request and then forwards the message to it:

```
void forward_handler(peer_id from, msg_type type, coop_nr coop, void *msg,
                     int msg_len) {
  peer_id liable_peer = find_liable_peer (msg, msg_len);

  forward(liable_peer , msg_type, msg, msg_len);
}
```

Listing 1.2. Forwarding the request

The implementation of *forward_handler* ignores errors that might happen when forwarding the message. In the case of an error, *Peer2* could reply back to *Peer1* with an error message, or try to forward the request to a different peer.

The message from *Peer2* to *Peer3* carries the address of *Peer1* that will allow *Peer3* to identify the requester and to reply to him. On *Peer3* Paradis-Net calls the local handler function (*serve_request*, listing 1.3) with the peer ID of *Peer1* as first parameter, so that the handler function will not be able to see the mediator that forwarded the request.

```
void serve_request(peer_id from, msg_type type, coop_nr coop, void *msg,
                   int msg_len) {
  reply_msg reply; // This variable will hold the reply
  int reply_len ;  // The length of the reply
```

```
  fulfill_request (msg, msg_len, &reply, &reply_len); // Application specific
  send(from, REPLY, coop, &reply, reply_len);
}
```

Listing 1.3. Serving the request

Scatter/Gather. Scatter/Gather is a one-to-many communication pattern in which a peer sends requests in parallel to many peers and blocks waiting for all individual responses. Figure 3 illustrates the procedure.

Clusterfile [3] uses this pattern to contact several data servers storing stripes of a given file. Although the requests are sent out in a particular order, the order of the replies is arbitrary.

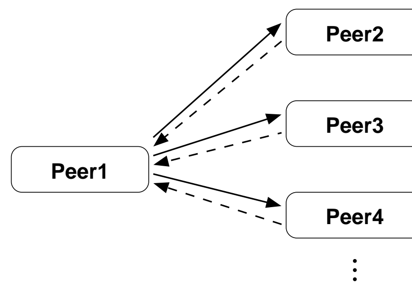

Fig. 3. The Scatter/Gather pattern
Solid arrow: Request – Dashed Arrow: Reply

The peers which play the server role in this pattern (*Peer2*, *Peer3*, . . .), define a handler function to process the request. This handler will, after assembling an answer, reply back to *Peer1*. The procedure accords with the one of *Peer3* in the delegation example and therefore the implementation is the same: see listing 1.3.

As an extension of the pattern, it is also possible to forward the request to a different peer using the *forward* function, akin to listing 1.2. This would result in a combination of the Scatter/Gather and the Delegation pattern.

For simplicity reasons we send the same message to every peer and expect reply messages of type *reply_type*. After sending the requests, *Peer1* will block in *end_cooperation* until all answers have been received.

```
reply_type* start(peer_id* peers, int num_peers, void* msg, int msg_len) {
  coop_nr coop;
  int i;
  // We allocate memory for all replies:
  reply_type *replies = malloc(num_peers * sizeof(reply_type));
  // init_desc allocates receive descriptors to accomodate the replies:
  rcv_desc *desc = init_desc(peers, num_peers, replies);

  coop = start_cooperation(desc, num_peers); // register cooperation

  for (i=0; i<num_peers; i++)              // send requests
    send(peers[i], REQUEST, coop, msg, msg_len);
  ...                                       // eventual computation
  end_cooperation(coop, 0);                 // wait for reply (no timeout)
  free(desc);                               // free the descriptors
```

```
  return replies;                                   // Success!
}
```

Listing 1.4. Sending requests

4.2 Structured P2P overlays

Three groups from MIT, Berkeley and Rice University joined their efforts in order to define a common three tier API for structured overlays [14]. The lowest tier 0 is the key-base routing layer, which provides basic communication services. Tier 1 provides higher level abstractions like distributed hash tables, while the applications at tier 2 use these abstractions in order to offer services like file sharing and multicasting. The researchers describe the tier 0 implementation of the four most influential structured overlay systems: CAN [15], Chord [16], Pastry [17] and Tapestry [18]. We will outline how the routing messages operations of tier 0 can be straightforward implemented using Paradis-Net.

An overlay node is identified by a *nodehandle* that encapsulates its network address (e.g. IP), in our case the unique peer name in Paradis-Net (section 3.1). Overlay nodes are assigned uniform random *node_ids* from a large identifier space by hashing their network addresses. The application objects are mapped on the same identifier space by computing a *key*. The objects are placed on the overlay nodes by assigning their keys to *node_ids* (for instance the longest prefix match in Tapestry). In order to efficiently sent a message from one node to the other, each node maintains a routing table. The routing table is used for choosing a next hop whose *node_id* is closer (for instance by using the Hamming distance) to the *node_id* of the destination. The routing strategy is system specific and is not discussed here. Given a node and a message to be routed, we assume the routing table delivers the *node_id* of the next hop.

At tier 0, two sets of API functions are proposed: routing messages and routing state access. The latter set refers to strictly local operations and is therefore not relevant for our discussion. The first set consists of three API functions: *route*, *forward* and *deliver*. The call *route(key K, msg M, nodehandle hint)* delivers the message M to the node storing the object associated with the key K. The optional argument hint specifies the first hop to be used. The route operation can be implemented using the continuation passing mechanism of Paradis-Net as a chain of handlers assigned to the message type $ROUTE$, running on the peers from the source to the destination. Each handler consults the local routing table in order to chose the next hop and then invoke the local *forward(key K, msg M, nodehandle nextHop)*, as provided by the application. This function may change K, M or $nextHop$, according to the application needs. Upon returning from it, *send* can be used for sending K and M to the node $nextHop$. At the destination, the up-call *deliver(key K, msg M)*, informing the application that a message for object K arrived, will be called from a Paradis-Net handler.

4.3 Transparent RDMA

The capabilities of direct access transport (DAT) standards such as the Virtual Interface Architecture [7], Infiniband [2] and Remote Direct Data Placement (RDDP [19]) reduce memory copy operations and minimize CPU demands when transfering data from the network interface to the main memory. The DAT Collaborative defines DAT requirements and standard APIs that include RDMA, memory registration, kernel bypass and asynchronous interfaces. RDMA allows direct memory-to-memory transport of data without CPU involvement. Memory registration facilitates the specification of granting access to the local memory regions used as destinations of RDMA operations. The kernel bypass eliminates unwanted kernel involvement in the communication path.

Paradis-Net cooperations fulfill the four characteristics of DAT protocols as validated by an implementation of Paradis-Net over VIA. Memory registration is implemented in the *start_cooperation* routine. A prerequisite for the use of RDMA is that the remote peer is known and identifiable through the *from* flag. When memory is registered, its handles, which are required for remotely accessing the memory, will be attached to the request along with the Cooperation token. The remote peer then has the information needed to directly write the reply into the client's memory without involving the CPU of the other machine.

Similarly, requests can be sent using RDMA and pre-registered buffers. In all cases, the kernel is bypassed through a complete user-level implementation over RDMA.

5 Conclusion

This paper introduced Paradis-Net, a low-level network interface which targets easier implementation of complex multi-party protocols. Paradis-Net emerged from our experience with parallel file systems and was motivated by the need of collaborative communication patterns.

To this end Paradis-Net introduces *Cooperations*, a mechanism that allows the user describing the result of collaborative work between several participating peers. We described the mechanism and its potential use by demonstrating how two common communication patterns used in parallel file systems can be implemented. Aside from distributed system development, we outlined how peer-to-peer functionality can be mapped onto Paradis-Net and how the library can transparently use remote direct memory access (RDMA) when available to increase performance.

References

1. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A gigabit-per-second local area network. IEEE Micro **15** (1995) 29–36
2. InfiniBand Trade Association: InfiniBand Architecture Specification Release 1.1. (2002)

3. Isailă, F., Tichy, W.F.: Clusterfile: A flexible physical layout parallel file system. In: Proceedings of IEEE Cluster Computing Conference, Newport Beach. (2001)
4. Isailă, F., Malpohl, G., Olaru, V., Szeder, G., Tichy, W.F.: Integrating collective I/O and cooperative caching into the Clusterfile parallel file system. In: Proceedings of the ACM International Conference on Supercomputing (ICS). (2004)
5. Anderson, T.E., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., Wang, R.: Serverless network file systems. ACM Transactions on Computer Systems **14** (1996) 41–79
6. Wang, R.Y., Anderson, T.E.: Experience with a distributed file system implementation. Technical Report CSD-98-986, University of California at Berkeley (1998)
7. http://www.viarch.org: VIA: The Virtual Interface Architecture. (1998)
8. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: A mechanism for integrated communication and computation. In: 19th International Symposium on Computer Architecture, Gold Coast, Australia (1992) 256–266
9. Nelson, B.J.: Remote Procedure Call. PhD thesis, Carnegie-Mellon University (1981)
10. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyone, B.: Design and implementation of the Sun network file system. In: Proceedings of Usenix 1985 Summer Conference. (1985) 119–130
11. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine. MIT press (1994)
12. The MPI Forum: MPI: A Message Passing Interface. In: Proceedings of the 1993 ACM/IEEE conference on Supercomputing. (1993) 878–883
13. Malpohl, G., Isailă, F.: The Paradis-Net API. Technical Report 2004/20, Universität Karlsruhe, Fakultät für Informatik, Germany (2004)
14. Dabek, F., Zhao, B., Druschel, P., Stoica, I.: Towards a common api for structured peer-to-peer overlays (2003)
15. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings of ACM SIGCOMM 2001. (2001)
16. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM Press (2001) 149–160
17. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware). (2001) 329–350
18. Zhao, B., Kubiatowicz, J., Joseph, A.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley (2001)
19. Internet Engineering Task Force: Remote Direct Data Placement Charter. (2002)