Verteiltes Rechnen unter Mach

Andreas Polze

Humboldt-Universität zu Berlin
Institut für Informatik

apolze@informatik.hu-berlin.de

ABSTRACT

Mach ist ein *Microkernel*-Betriebssystem, das an der Carnegie Mellon Universität im Rahmen eines DARPA Forschungsprojektes entstand. Mittlerweile sind kommerzielle Mach-basierende Systeme für eine Vielzahl von Architekturen verfügbar. Die bekanntesten sind Mt.Xinu, NeXTSTEP und OSF/1, weitere Mach-basierende Betriebssysteme sind angekündigt.

Mach bietet leistungsfähige, netzwerktransparente nachrichtenbasierte Interprozeß-Kommunikation (IPC) über Ports an. Der *Mach Interface Generator* (MIG) ist ein Werkzeug zur Generierung von Rümpfen für Klienten- und Server-Programme, die über Ports kommunizieren. Wir wollen anhand eines kleinen Klient/Server-Beispiels wichtige Prinzipien von Mach-IPC und die Verwendung von MIG demonstrieren.

1. Einleitung

Seitdem Betriebssysteme *Multitasking* anbieten, suchen Anwendungsprogrammierer nach Wegen, verschiedene Programme miteinander kommunizieren zu lassen. Komplexe Probleme lassen sich dann durch kooperierende Prozesse elegant lösen. Im Umfeld des Betriebssystems UNIX existieren daher eine Reihe von Mechanismen zur Interprozeß-Kommunikation. Beispiele sind:

- UNIX *pipes*
- System III *named pipes*
- Berkeley UNIX *sockets*
- AT&T System V *message queues*

Mach-IPC unterscheidet sich von allen diesen Ansätzen durch einige Eigenarten:

Kommunikation zwischen Prozessen erfolgt in Mach vollkommen netzwerktransparent. Nachrichten können zwischen Prozessen auf derselben Maschine oder zwischen verteilten Prozessen auf diesselbe Weise ausgetauscht werden. Keiner der Kommunikationspartner nimmt Notiz davon, wenn eine Nachricht über das Netz

transportiert wird. Mach-Nachrichten sind getypt, die Konversion eines Typs in unterschiedliche Repräsentationen erfolgt in heterogenen Umgebungen für den Programmierer transparent.

In Mach sind das IPC-System und die virtuelle Speicherverwaltung miteinander integriert. Große Nachrichten müssen im lokalen Fall nicht kopiert werden, sie können durch die Speicherverwaltung direkt in den Adreßraum des empfangenden Prozesses abgebildet werden. Über das Netzwerk werden solche Nachrichten erst dann transportiert, wenn der empfangende Prozeß tatsächlich auf die Daten zugreift.

Die effiziente Implementation von IPC im Mach-System beseitigt damit Einschränkungen, die sowohl UNIX *pipes* (Kommunikation nur zwischen verwandten Prozessen, ungetypt, unidirektional, nur lokal nutzbar), System III *named pipes* (ungetypt, unidirektional, nur lokal nutzbar), System V *message queues* (ungetypt, nur lokal nutzbar) wie auch Berkeley *sockets* (ungetypt, häufiges Kopieren der Daten) anhaften.

2. Grundlegende IPC Konzepte

Mach definiert Kommunikationskanäle zwischen *Task*s[1] als *Port*s. Ports werden durch Warteschlangen für Nachrichten implementiert und vom Kern verwaltet. Nachrichten enthalten getypte Datenobjekte, der Programmierer muß Wert und Typ der Daten angeben, die in einer Nachricht transportiert werden sollen.

Zugriffsrechte beschreiben die IPC-Operationen, die eine Task mit einem Port ausführen darf. Sie werden als Kombination der Rechte *send*, *receive* und *ownership* ausgedrückt. Nur Tasks mit Senderecht können erfolgreich Nachrichten an einen Port schicken, analog muß eine Task Empfangsrechte besitzen, um Nachrichten von einem Port entgegennehmen zu können. Während mehrere Tasks Senderechte zu einem Port besitzen können, kann nur eine Task Empfangsrechte besitzen. Existieren in einer Task mehrere *thread*s, so gelten für alle threads identische Zugriffsrechte bezüglich eines Ports.

Eigentümerrechte spielen dann eine Rolle, wenn eine Task ihre Empfangsrechte zu einem Port aufgibt oder gar terminiert. In diesem Fall gehen die Empfangsrechte auf die Task mit Eigentümerrechten an dem Port über. Existiert keine Task mit Empfangs- oder Eigentümerrechten an einem Port, so gibt der Mach-Kern den Port frei und informiert alle Tasks mit Senderechten zu diesem Port.

Besitzt eine Task Empfangs- oder Eigentümerrechte an einem Port, so erhält sie automatisch auch Senderechte zu diesem Port. Eine Task erhält alle drei Rechte, wenn sie einen Port erzeugt. Zugriffsrechte auf einen Port können jederzeit von einer Task auf eine andere übertragen werden. Dazu muß lediglich eine Mach-Nachricht mit dem Namen des entsprechenden Ports an die Zieltask gesandt werden. Die Typangabe für den Port-Namen in der Nachricht spezifiziert dann die Zugriffsrechte, die die Zieltask erhält.

---

[1] Mach Tasks unterscheiden sich von herkömmlichen UNIX-Prozessen in einer Reihe von Details (Speicherverwaltung, Signale, etc.). Im Rahmen diese Artikels können jedoch die Begriffe Prozeß und Task als Synonyma angesehen werden.

Jeder Port wird durch einen eindeutigen Namen bezeichnet. Diese Namen sind durch ganzzahlige Werte implementiert. Verschiedene Tasks können ihre eigenen, womöglich verschiedenen Namen für denselben Port besitzen. Werden Port-Namen in Nachrichten versandt, so transformiert der Mach-Kern den für den Absender gültigen Namen in einen in der Empfänger-Task gültigen Namen. Dabei wird der zugehörige Port für die Empfänger-Task womöglich erst angelegt.

Interessant bleibt jedoch die Frage, wie eine Task jemals von einer anderen Task erfahren kann. Erst dann kann sie ihr ja eine Nachricht senden und mitteilen, unter welchem Port sie selbst erreichbar ist.

3. Verteiltes Rechnen

Modelle für verteiltes Rechnen sollen die physische Verteilung der Komponenten einer Anwendung verbergen und die Existenz des Netzwerks verdecken. Kommunizierende Tasks können dann genauso leicht auf verschiedenen Maschinen ausgeführt werden, wie auf einer einzigen. Machs-IPC stellt ein solches Modell dar und unterstützt sogar transparente Kommunikation in heterogenen Umgebungen.

Anhand eines Klient/Server-Beispiels wollen wir die Möglichkeiten und Werkzeuge die Mach bietet, erläutern. Die beiden Komponenten unserer verteilten Anwendung heißen `packets_client` und `packets_server`. Der Klient kann Nachrichten mit einem Integer-Wert an den Server schicken. Er ruft dazu eine Routine `send_packet()` auf. Der Server druckt für jeden empfangenen Wert eine Notiz aus. Sendet man sehr viele Nachrichten hintereinander oder senden viele Klienten auf einmal an einen Server, so kann eine Überflutung von Puffern oder des Netzwerkes autreten. Zum Aufspüren derartiger Fälle ist unsere Anwendung geeignet — wenn man Klient und Server noch ein wenig erweitert.

Bevor Klient und Server kommunizieren können, stellt sich nun die Frage, wie beide voneinander erfahren. Sie kommunizieren dazu mit einem Namensdienst. Bild 1 zeigt die Initialisierungsschritte, die vor der ersten Klient/Server-Interaktion nötig sind.

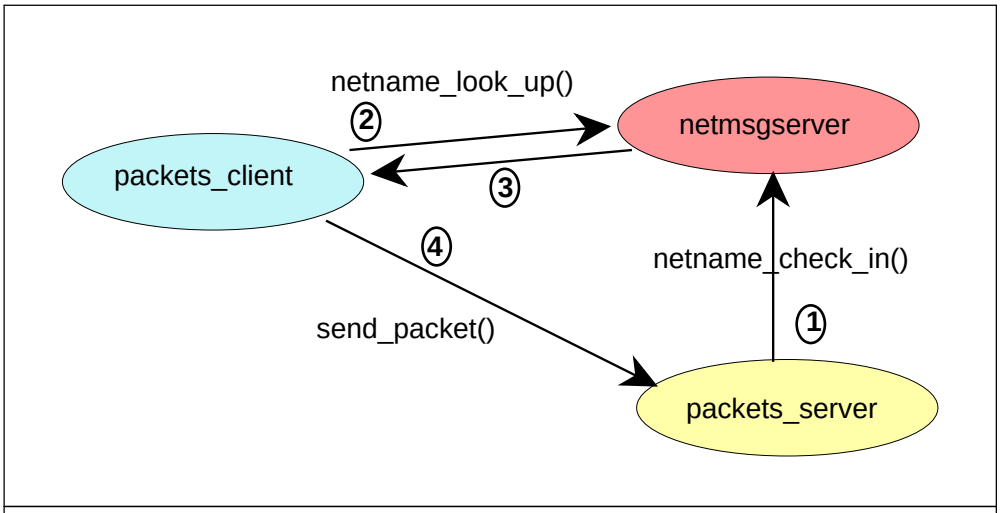

Bild 1*: Der Mach-Namensdienst*

Neben einem *Microkernel* besitzt das Mach-Betriebssystem eine Reihe von *user-level*-Servern. Einer davon heißt *network message server* (`netmsgserver`). Jede Task besitzt Senderechte zu einem Port, der mit dem `netmsgserver` verbunden ist. Ein weiterer Port, zu dem jede Task Senderechte besitzt, ist der die Task beschreibende Port `task_self()`.

Ein Programm, das Senderechte zu einem Port an andere Tasks vergeben will, kann sich nun beim `netmsgserver` registrieren. Das geschieht mit der Funktion `netname_check_in()`. Es gibt dazu neben dem Port-Namen einen ASCII-Bezeichner an. Dieser Bezeichner kann bis zu 80 Zeichen lang sein, inklusive eines abschließenden NULL-Zeichens. Ein anderes Programm erhält nun Senderechte zu einem Port wenn es mit einem Aufruf der Funktion `netname_look_up()` an den Namensdienst eine Anfrage stellt und den korrespondierenden Bezeichner angibt. In unserem Szenario müssen sich daher Klient und Server im voraus auf einen Namen geeinigt haben.

```
# include <servers/netname.h>
# include <mach/port.h>

int allocate_and_check_in_port( char * name, port_name_t * port ) {
/* allocate a new port and check the name in with netmsgserver*/
          kern_return_t ret;
          netname_name_t n_name;

          ret = port_allocate( task_self(), port );
          if (ret != KERN_SUCCESS) {
               mach_error("port_allocate:", ret ); return -1;
          }
          strncpy(n_name, name, sizeof(n_name));
          ret = netname_check_in( name_server_port, n_name,
                         task_self(), *port );
          if (ret != NETNAME_SUCCESS) {
               mach_error("netname_check_in:", ret); return -2;
          }
          return 0;
}
```

Bild 2*: Registrieren eines Ports*

In Bild 2 zeigen wir eine Funktion, die zunächst mit `port_allocate()` einen neuen Port anfordert. Dieser Port wird dann beim Namensdienst mit `netname_check_in()` registriert. Wir benutzen dabei die globale Variable `name_server_port`, die vom Mach-Laufzeitsystem initialisiert wird. Das zweite Argument bei diesem Funktionsaufruf ist der Name, den wir bekannt machen wollen. Das dritte Argument ist ein weiterer Port — die ‘‘Unterschrift’’. Zu diesem Port muß unser Server Senderechte haben. Der Namensdienst notiert sich die ‘‘Unterschrift’’ und erlaubt das Löschen eines registrierten Namens nur dann, wenn derselbe Port als ‘‘Unterschrift’’ angegeben wird. Gibt man die Konstante `PORT_NULL` als ‘‘Unterschrift’’ an, so wird dieser Sicherheitsapparat außer Kraft gesetzt.

In unserem Beispiel haben wir `task_self()` als Unterschrift angegeben. Dieser Port ist eindeutig, die Server-Task hat dafür Senderechte und seine Verwendung ist

bequem. Generell sollte man jedoch den Port `task_self()` nicht an andere Tasks aushändigen, denn dieser Port kann benutzt werden, um eine Task zu manipulieren. Beispielsweise sind bei Senderechten zu diesem Port Manipulation im Adreßraum einer fremden Task möglich. Wir vertrauen jedoch dem Namensdienst und verzichten hier auf eine weitergehende Diskussion.

Wie wir bereits gesehen haben, muß nun noch der Klient den Port des Servers vom Namensdienst erfragen, bevor beide das erstemal miteinander kommunizieren können. Bild 3 zeigt die Funktion `lookup_port()`, die dies bewerkstelligt.

```
# include <mach/mach.h>

# define ASK_LOCAL_NETMSGSRV ""
# define ASK_ANY_NETMSGSRV   "*"

int lookup_port( char * name, port_name_t * port ) {
          /* lookup a port registered with netmsgserver*/
          kern_return_t ret;
          netname_name_t n_name;

          strncpy(n_name, name, sizeof(n_name));
          if ((ret = netname_look_up( name_server_port,
               ASK_ANY_NETMSGSRV, n_name, port)) != NETNAME_SUCCESS) {
               mach_error("netname_lookup", ret); return -1;
          }
          return 0;
}
```

Bild 3*: Anfrage an den Namensdienst*

Bei der Anfrage an den Namensdienst benutzen wir wiederum die globale Variable `name_server_port` und geben ein Argument vom Typ `netname_name_t` zur Beschreibung des gewünschten Dienstes an. Die Funktion `netname_look_up()` alloziert implizit einen neuen Port wenn die Anfrage an den Namensdienst erfolgreich war. Ein Verweis auf diesen Port wird als viertes Argument der Funktion zurückgegeben. Für diesen Port hat unsere Klient-Task nun Senderechte, über ihn kann sie mit dem Server kommunizieren.

Eine besondere Rolle spielt das zweite Argument der Funktion `net-name_look_up()`. Hier kann der Name des Rechners angegeben werden, auf dem nach einem registrierten Namen gesucht werden soll. Der Name `""` beschreibt den lokalen Rechner, während der Name `"*"` einen *broadcast* im lokalen Netz auslöst und auf allen angeschlossenen Rechnern nach einem registrierten Namen gesucht wird. Ist der Name mehrfach registriert, so wählt der lokale `netmsgserver` einen Port aus, um ihn an die Klienten-Task weiterzureichen. Dieses Verfahren erlaubt es, nach einem Dienst zu suchen, ohne den Rechner zu kennen, auf dem er implementiert ist.

Nachdem nun unser `packets_server` und das `packets_client`-Programm die besprochenen Initialisierungsschritte vollzogen haben, ist der Klient in der Lage, Nachrichten an den Server zu senden. Diese Nachrichten müssen eine wohldefi nierte Struktur haben, eine Vielzahl von Einträgen sind vom Programmierer ‘‘von Hand’’ auszufüllen, ehe der Klient das erste Mal eine gültige Mach-Nachricht an den Server

senden kann. Mit dem *Mach Interface Generator* (MIG) existiert jedoch ein leistungsfähiges Werkzeug, das uns viel von diesem Programmieraufwand abnimmt.

4. Der *Mach Interface Generator*

Ausgehend von einer Spezifikation, `packets.defs` in unserem Beispiel (der Suffix `.defs` ist üblich), erzeugt der *Mach Interface Generator* C-Code für Klient und Server. Auf der Klienten-Seite wird dabei ein *procedure call interface* zu Mach-Nachrichten generiert, während auf der Server-Seite ein *message passing interface* zu den C-Funktionen entsteht, die die Server-Funktionalität implementieren. Der Programmierer hat mit dem Erzeugen von Mach-Nachrichten-Strukturen, dem Eintragen von Typ- und Längeninformation und der Zuordnung verschiedener Typen von Nachrichten zu verschiedenen Funktionen auf der Server-Seite nichts zu tun. In Bild 4 zeigen wir die von MIG generierten Dateien in ihrer Beziehung zu ‘‘von Hand’’ codierten Programmstücken.

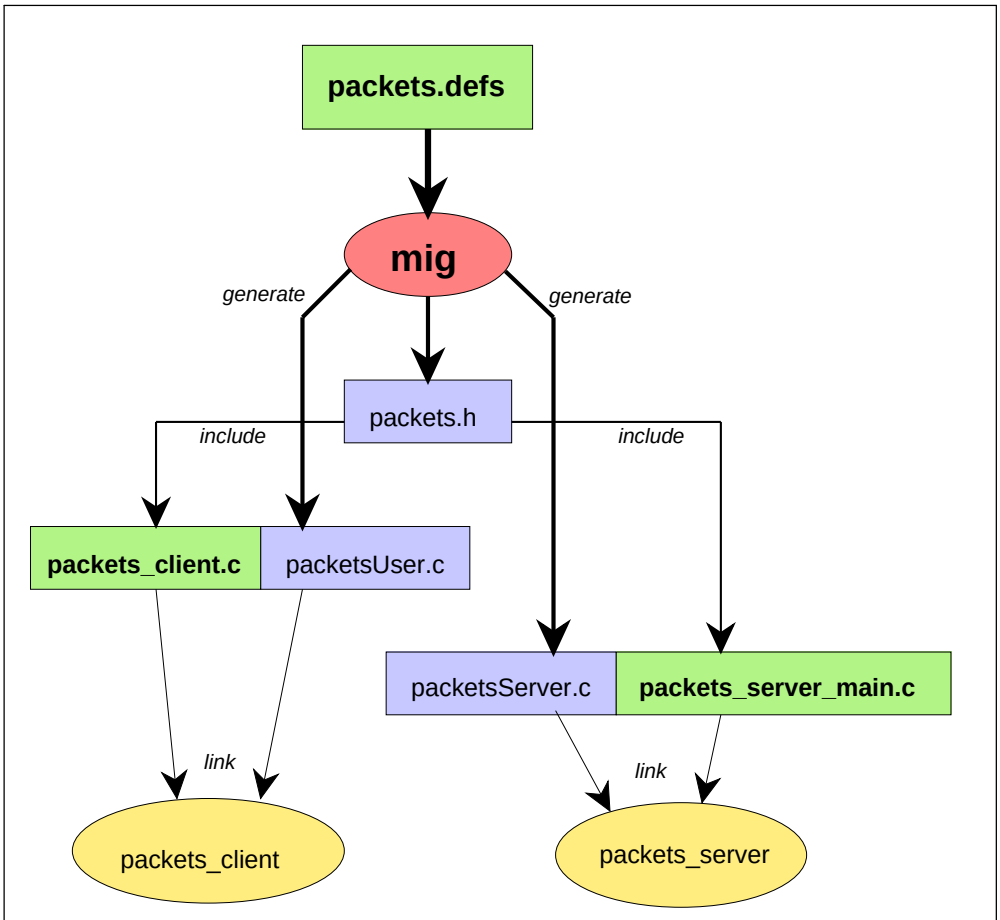

Bild 4*: Generieren von Klient und Server aus einer MIG Spezifikation*

In Bild 4 bezeichnen Ellipsen jeweils ausführbare Programme. Zentral ist hier `/usr/bin/mig`, der *Mach Interface Generator* (MIG). Namen in Kästchen bezeichnen Dateien mit C-Quelltext oder MIG-Spezifikationen. Die in Bild 4 fett gedruckten Dateinamen deuten auf Programmstücke, die der Programmierer zu erstellen hat, der Rest

wird von MIG generiert. Wir wollen uns zunächst mit der Spezifikation der Kommunikationsschnittstelle zwischen Klient und Server in der Datei `packets.defs`[2] beschäftigen.

```
subsystem packets 0;

# include <mach/mach_types.defs>

simpleroutine send_packet(
     RequestPort   server:   port_t;
     in            count:    int
);
```

Bild 5*: Datei packets.defs*

In der Mach-Terminologie wird jedes Klient/Server-Paar mit den zugehörigen Operationen als *Subsystem* bezeichnet. Jedes Subsystem muß seinen eigenen Bezeichner haben, in unserem Beispiel wird in Bild 5 der Bezeicher `packets` festgelegt. MIG generiert für jedes Subsystem die Dateien `<subsystem>User.c` und `<subsystem>Server.c`. Nach dem Subsystem-Namen muß eine Konstante (0 in unserem Fall) angegeben werden. Sie legt den initialen Wert fest, den MIG im `msg_id`-Feld der IPC-Nachrichten zur Bezeichnung der ersten Operation benutzt. Weiteren Operationen werden fortlaufend erhöhte `msg_id`s zugeordnet.

Nach dem Subsystem-Bezeichner folgen in einer MIG-Spezifikationsdatei Typspezifikationen und Import-Deklarationen. In dieser Sektion werden MIG-Typen mit korrespondierenden C-Typen assoziiert. Für alle ‘‘Standard’’-C-Datentypen und viele im Umfeld von Mach verwendete Datentypen existieren Deklarationen in den Dateien `mach/std_types.defs` und `mach/mach_types.defs`, die beide im Verzeichnisbaum unterhalb von `/usr/include` residieren. Für die in unserem Beispiel verwandten Datentypen `int` und `port_t` existieren dort bereits Deklarationen. Neben einfachen Typen unterstützt MIG die Deklaration von Strukturen, Felder und Zeiger.

Import-Deklarationen eröffnen die Möglichkeit, in den von MIG generierten Dateien `#include`-Direktiven zu erzeugen. Mit `{us}import <filename>` können solche Direktiven entweder im ***u**ser*-Module, im ***s**erver*-Module oder in beiden Moduln generiert werden.

Eine weitere Sektion in einer MIG-Spezifikationsdatei nimmt die Beschreibungen von Operationen auf. Für jede solche Operation generiert MIG für den Klienten eine Funktion mit den korrespondierenden C-Argumenten, deren Aufruf das Senden einer Mach-Nachricht veranlaßt. Auf der Server-Seite wird Code generiert, der die Nachricht entpackt und eine Funktion gleichen Namens wie die ursprünglich im Klienten gerufene Funktion aktiviert. Der Programmierer muß im Server die Implementation der Funktion hinterlegen.

---

[2] Wir werden für unser Beispiel wesentliche Konstrukte aus dem MIG-Umfeld besprechen. Für eine vollständige Referenz sei der Leser auf die Literaturliste oder die Dokumentation eines Mach-Systems (z.B. NeXTSTEP) verwiesen.

Es existieren fünf verschiedene Arten von Operationen, darunter synchrone und asynchrone. Bild 6 zeigt die Unterschiede, die sich bei der Behandlung von Rückgabewerten und Fehlern ergeben, die beim Senden der Mach-Nachrichten auftreten können.

	Nachricht senden	Antwort empfangen	Fehler-meldung zurück	Rück-gabe-wert
`simpleroutine`	✓		✓	
`routine`	✓	✓	✓	
`simpleprocedure`	✓			
`procedure`	✓	✓		
`function`	✓	✓		✓

Bild 6*: MIG Operationen*

Die in unserem Beispiel definierte Operation `send_packet()` ist als `simpleroutine` deklariert. Wie in Bild 6 ersichtlich, wird für diese Routine nur eine Nachricht mit der Operationsanforderung vom Klienten zum Server gesandt, dieser sendet keine Antwort (asynchron). Der Aufrufer der Funktion `send_packet()` erfährt jedoch, ob das Aussenden der Nachricht an den Server ohne Probleme vonstatten gegangen ist.

Klassische RPC's werden in MIG-Notation als `routine` oder `procedure` vereinbart. Für diese Fälle generiert MIG Code, der die Mach-Funktion `msg_rpc()` benutzt. Dies funktioniert effizienter als Paare aus `msg_send()` und `msg_receive()`.

Die Spezifikation der Operationen mit ihren Parametern muß folgender Syntax genügen:

operation	::=	operation-type op-name `(` parameter-list `);` | `function` op-name `(` parameter-list `) :` func-type`;`.
operation-type	::=	`simpleroutine` | `routine` | `simpleprocedure` | `procedure` .
parameter-list	::=	parameter { `;` parameter }.
parameter	::=	[specifikation] var-name `:` type [`,` dealloc-flag].
specifikation	::=	`in` | `out` | `inout` | `RequestPort` | `ReplyPort` | `WaitTime` | `MsgType` .

Bild 7: *MIG Operations-Deklarationen*

Auch zu der in Bild 7 angegebenen Syntax für MIG-Operationen sollen nur einige ausgesuchte Erklärungen genügen.

Die Spezifikationen `in`, `out` und `inout` geben an, ob ein Parameter nur vom Klient zum Server, nur zurück oder in beiden Richtungen transportiert werden soll.

`RequestPort` bezeichnet den Server, fehlt diese Spezifikation, so wird angenommen, daß der erste unspezifizierte Parameter diesen Port beschreibt. Fehlt die Angabe eines `ReplyPorts`, so generiert MIG Code zur Allokation eines internen Ports für Antwortnachrichten. Generell werden nicht-spezifizierte Parameter als `in`-Parameter interpretiert.

Nach den Operationsdeklarationen kann eine MIG-Spezifikationsdatei noch einige Optionen enthalten. So kann man mit `ServerPrefix` und `UserPrefix` dafür sorgen, daß Klienten- und Server-Version einer Operation verschiedene Namen erhalten. Damit wird es möglich, Server zu implementieren, die als Klienten von sich selbst auftreten.

Wir wollen unsere Exkursion durch MIGs Gefilde hier abschließen und uns im nächsten Abschnitt mit den noch fehlenden Code-Fragmenten unseres Klient/Server-Beispiels beschäftigen.

5. Vervollständigung von Server und Klient

```
# include "packets.h"

boolean_t packets_server(msg_header_t *InHeadP,msg_header_t *OutHeadP);

void server_loop(port_t port) {
          struct dummy_msg {
               msg_header_t head; msg_type_t type; int data;
          } msg, reply;

          while (TRUE) {
              msg.head.msg_local_port = port;
              msg.head.msg_size = sizeof(dummy_msg);
              msg_receive(&msg.head, MSG_OPTION_NONE, 0);

              (void)packets_server((msg_header_t *)&msg,
                              (msg_header_t *)&reply);

              reply.head.msg_local_port = port;
              msg_send(&reply.head, MSG_OPTION_NONE, 0);
          }
}

kern_return_t send_packet ( port_t server, int count) {
          fprintf(stderr,"got packet %d from %d\n", count, server);
          return KERN_SUCCESS;
}
```

Bild 8: *Server-Hauptschleife und Implementation von send_packet()*

Während MIG auf der Klient-Seite Code zum Versenden von Mach-Nachrichten generiert, liegt das Empfangen der Nachrichten auf der Server-Seite in der Verantwortung des Programmierers. Immerhin generiert MIG eine Dispatcher-Funktion, die empfangene Nachrichten analysiert und die ‘‘richtige’’ Operation im Server aufruft. Bestandteil eines jeden Servers ist daher eine Hauptschleife, die fortwährend Nachrichten entgegennimmt, die Dispatcher-Funktion ruft und eine Antwort an den Klienten sendet. Für unseren `packets_server` ist diese Hauptschleife in Bild 8 dargestellt.

Ein Problem bei dieser Art, Server zu implementieren, versteckt sich hinter der Typvereinbarung von `dummy_msg`. Der Programmierer muß hier eine Struktur angeben, die so groß ist, daß sie alle Argumente jeder in einem Subsystem deklarierten Operation aufnehmen kann. In unserem Beispiel ist das kein Problem — wir müssen nur die Operation `send_packet()` betrachten und jene besitzt lediglich einen ganzahligen Parameter, der in der Nachricht ‘‘unterkommen’’ muß. Für die MIG-generierte Dispatcher-Funktion `packets_server()` ist nur der Kopf der empfangenen Nachrichten interessant. Ein Aufruf dieser Funktion mit der passenden Nachricht als Argument hat dann einen Aufruf der Server-Version von `send_packet()` zur Folge.

Auf der Klienten-Seite geht alles viel einfacher. Wir zeigen in Bild 9 das Hauptprogramm mit einem Aufruf der Klienten-Version von `send_packet()`.

```
# include <packets.h>

main ( int argc, char ** argv ) {
          port_t server;
          int i, count = 100;

          if (argc > 1) count = atoi( argv[1] );


          if (lookup_port( PACKETS_SERVER_PORT, & server) < 0) {
                fprintf(stderr,"packets server not found, exiting\n");
                exit(1);
          }

          for (i = 0; i < count; i++)
               send_packet( server, i );
          exit(0);
}
```

Bild 9*: Vollständiges Klienten-Programm*

Hier wird die Sache elegant. Der Anwendungsprogrammierer, der einfach einen Mach-Server benutzen möchte, bekommt von Nachrichten, Sende- und Empfangsoperationen überhaupt nichts mit. Er muß lediglich die Anfrage an den Namensdienst codieren und kann von diesem Punkt an vollkommen transparent mit einer anderen, womöglich entfernten Mach-Task kommunizieren.

6. Zusammenfassung

Das Betriebssystem Mach ist ein existierendes *Microkernel*-Betriebssystem mit einer UNIX-artigen Oberfläche. Neben einer Reihe kommerzieller Mach-basierender Systeme, existieren etliche frei zugängliche Mach Implementationen. Darunter sind Carnegie Mellons Mach 3 (http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/project/mach/public/www/-mach.html), das an der University of Utah bearbeitete *flexmach* (http://www.cs.utah.edu/-projects/flexmach/index.html), die OSF Mach Distribution (ftp riftp.osf.org; pub/-snapshots/osfmach3) und GNUs HURD-Projekt (http://www.cs.pdx.edu/~trent/gnu/-hurd.html). Den freien Mach-Systemen fehlt üblicherweise ein UNIX-Server, jedoch sind Arbeiten im Gange, Linux- und BSDLite-basierende Server verfügbar zu machen. So ist in Zukunft mit weiterer Verbreitung Mach-basierender Systeme zu rechnen.

Wir haben anhand einer kleinen verteilten Anwendung die grundsätzlichen Prinzipien von Mach-IPC und des Werkzeugs MIG erörtert. Die Möglichkeiten, die MIG zur Spezifikation von Protokollen zur Kommunikation in verteilten Anwendungen bietet, sind für viele Fälle angemessen und leicht zu handhaben. MIG unterstützt die Generierung von Code für Server, die als Klienten von sich selbst auftreten. Damit lassen sich beliebig komplizierte Interaktionen in verteilten Anwendungen als ‘‘ineinander geschachtelte’’ Klient/Server-Strukturen spezifizieren.

Die Verwendung von Daten variabler Größe und von Zeigern in MIG-Operationen ist etwas trickreich zu handhaben. Beim Umgang mit solchen Datentypen sind Relationen zwischen Mach-IPC und der virtuellen Speicherverwaltung zu berücksichtigen. Dieses Thema soll in einem späteren Artikel zur Sprache kommen.

Literatur

J.Boykin, D.Kirschen, A.Langerman, S.LoVerso; *Programming under Mach*; Addison-Wesley UNIX and Open System Series, ISBN 0-201-52739-1, Addison-Wesley 1993.

K.Loepere, Editor; *The Mach 3 Server Writer’s Guide*; Open Software Foundation and Carnegie Mellon University 1991, (http://www.cs.cmu.edu:8001/afs/cs/project/mach/-public/www/doc/osf.html).

R.P.Draves, M.B.Jones,et al.; *MIG — The Mach Interface Generator*; Unpublished Manuscript, Carnegie Mellon University, Pittsburgh PA, July 1989.