Shared Objects Memory
Kommunikation unter Mach

Dr. Andreas Polze

Humboldt-Universität zu Berlin
Institut für Informatik

apolze@informatik.hu-berlin.de

ABSTRACT

In diesem Artikel präsentieren wir einen kurzen Überblick über die Architektur unseres Mach-basierenden *Shared Objects Memory*-Systems. Das System implementiert eine Speicherverwaltung (“*Distributed Shared Memory*”) auf der Basis von replizierten C++-Objekten. Replizierte Objekte können von verschiedenen Mach-Tasks gemeinsam benutzt werden, das Ausführen paralleler Tasks in Workstation-Umgebungen wird damit möglich. Das System unterstützt mehrere (schwache) Konsistenzprotokolle zur Aktualisierung der Objekte. Wir konzentrieren uns hier auf die Kommunikationsstruktur: mehrere ineinander-geschachtelte Klienten und Server.

In der vergangenen Ausgabe der UNIX/mail hatten wir uns mit grundlegenden Prinzipien der Interprozeß-Kommunikation (IPC) über Ports im Betriebssystem Mach auseinandergesetzt. Wichtigstes Werkzeug bei der Mach-Programmierung ist der *Mach Interface Generator* (MIG), der die Generierung von Rümpfen für Klienten- und Server-Programme ermöglicht, die über Ports kommunizieren. Nachdem wir bereits die grundlegenden Fähigkeiten von MIG berschrieben haben, nehmen wir das *Shared Objects Memory*-System zum Anlaß, um einige der weitergehenden Fähigkeiten von MIG zu besprechen.

1. Shared Objects Memory

Das *Shared Objects Memory (SOM)*-System erlaubt die Abarbeitung paralleler Programme in verteilten, Mach-basierenden Umgebungen. SOM implementiert ein *Distributed Shared Memory*-System auf der Basis von Objekten. Im Gegensatz zu üblichen, seitenbasierten Ansätzen [Li/Hudak 89][Lo/Nitzberg 91] stehen Objekte als gemeinsam benutzte Einheiten unter voller Kontrolle durch den Programmierer. Das Problem des *false sharing* von unabhängigen Variablen, die nur zufällig auf derselben Speicherseite liegen, kann völlig vermieden werden. Im SOM-System wird die Aktualisierung der Replikate eines Objekts durch schwache Konsistenzprotokolle gesichert. Auf diese Weise kann die zur Konsistenzerhaltung nötige Kommunikation minimiert werden.

Das *Shared Objects Memory*-System besteht aus zwei Komponenten: dem *object repository* und einer C++-Klassenbibliothek, mit der parallele Programmthreads Zugriff zu *shared objects* erhalten. Bild 1 zeigt zwei Mach-Tasks mit parallelen Programmthreads,

die oberhalb des *Shared Objects Memory* angesiedelt sind. Beide Tasks kommunizieren mit einer dritten, der *object repository*-Task. Neben dem eigentlichen Programmthread enthält jede parallele Task einen Konsistenzmanagerthread. Dieser Thread wird implizit von der SOM-Klassenbibliothek erzeugt. Im Sinne von MIG ist der Konsistenzmanager ein Server, der es erlaubt, auf entfernte Replikate eines Objekts zuzugreifen um diese zu aktualisieren.

Betrachten wir nun die Erzeugung und Verwaltung von zwei *shared objects* etwas detaillierter: In Bild 1 erzeugen beide Tasks ein C++-Objekt `o1`. Die zweite Task erzeugt zusätzlich ein Objekt `o2`. Beise Objekte sind Instanzen einer von der `SOM`-Basisklasse abgeleiteten Klasse. Objekte im SOM-System werden durch eindeutige ganze Zahlen bezeichnet. Diese Bezeichner müssen durch den Programmierer oder einen Präprozessor zum C++-Compiler vergeben werden.

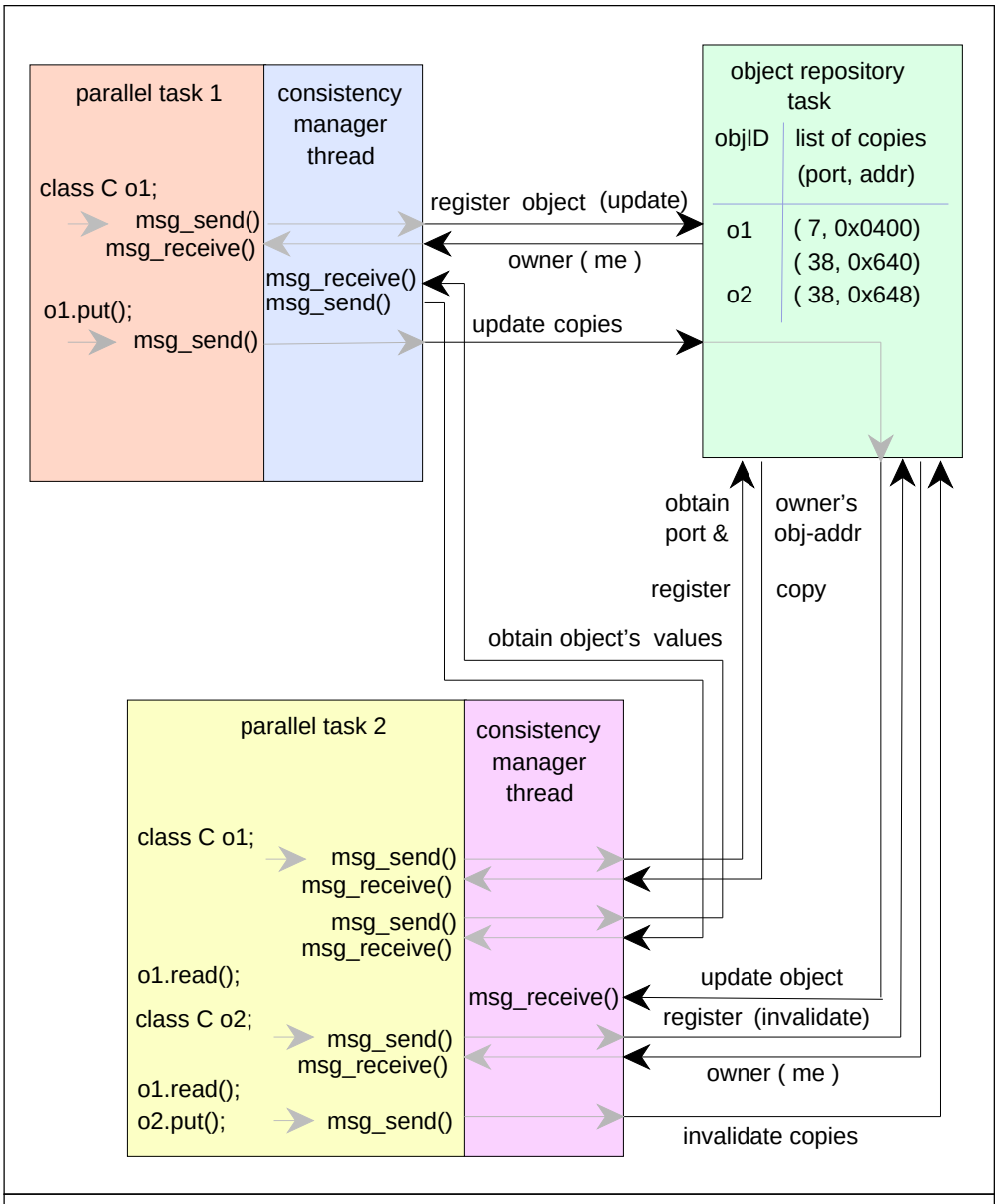

Bild 1*: Shared Objects Memory — Architektur*

Nach der Erzeugung eines Objekts registriert jede Task (d.h. der Konstruktor des Objekts) das Objekt beim *object repository*. Diese Task ist über den Mach `netmsgserver` unter einem wohlbekannten Namen zugänglich. Im *repository* wird für jeden Objektbezeichner eine Liste von Referenzen zu den Replikaten des Objekts gespeichert. Jeder Eintrag in der Liste besteht aus einem Mach-Port als Bezeichner für eine Task und der Adresse unter der das Objekt in der betreffenden Task zugänglich ist. Eine Task, die ein neues Objekt beim *repository* registriert, erhält eine Antwort: den Port und die Objektadresse des Eigentümers des betreffenden Objekts. In unserem Beispiel in Bild 1 stellt Task 2 fest, daß bereits ein Eigentümer für das Objekt `o1` registriert ist. Also wird eine zusätzliche Nachricht an Task 1 ausgesandt, um die Werte von Objekt `o1` von dieser Task zu erlangen.

Ist ein Objekt einmal erzeugt und beim *repository* registriert, so bietet die C++-Klasse `SOM` zwei Komponentenfunktionen, um alle Replikate eines Objekts entweder zu aktualisieren oder für ungültig zu erklären. Beide Funktionen senden Nachrichten an andere Tasks, die von den dort existierenden Konsistenzmanagerthreads behandelt werden. Damit keine Inkonsistenzen durch wechselseitige Zugriffe des Konsistenzmanagers und des eigentlichen parallelen Programms (-threads) auf ein Objekt entstehen, werden alle Zugriffe auf `SOM`-Objekte über *mutex*-Strukturen koordiniert.

Die Klasse `SOM` assoziiert keine bestimmte Politik mit ihren *update*- und *invalidate*-Operationen. Die entsprechenden Komponentenfunktionen der Klasse `SOM` können jedoch benutzt werden, um spezielle (schwache) Konsistenzprotokolle zu implementieren. Schwache Konsistenzprotokolle erlauben es, das Ausmaß an Kommunikation zur Erhaltung von Konsistenz unter replizierten Objekten zu minimieren — ein wichtiges Ziel beim Rechnen in verteilten Umgebungen. Auch ein weiteres Ziel, die Überlagerung von Kommunikation und Berechnung kann mit dem *Shared Objects Memory*-System erreicht werden. In unserem System existieren die von der Klasse `SOM` abgeleiteten Klassen `ENTRY_CONS` und `RELEASE_CONS`. Instanzen dieser Klassen sind replizierte Objekte, die nach dem *entry consistency*-Modell [Bershad et al. 91] oder dem *release consistency*-Modell [Gharachorloo et al. 90] aktualisiert werden. Beide Klassen implementieren die Komponentenfunktionen `acquire_write_lock`, `acquire_read_lock` und `release_lock`, mit denen ein Programm (-thread) entweder exklusive oder non-exklusive Zugriffsrechte auf ein *shared object* erlangen kann.

Subklassen von `ENTRY_CONS` oder `RELEASE_CONS` können weitere Datenkomponenten enthalten. Hinter den Komponentenfunktionen einer solchen Subklasse lassen sich alle mit der Replikation von Objekten zusammenhängenden Details verbergen. Der Benutzer von *shared objects* sieht dann keinen Unterschied zwischen solchen Objekten und herkömmlichen, sequentiell benutzbaren C++-Objekten.

2. Kommunikationsstruktur

In unserem Beispieszenario kooperieren drei Mach-Tasks. Die *object repository*-Task tritt dabei als Server gegenüber den parallelen Programmthreads auf. Gleichzeitig ist diese Task jedoch Klient gegenüber den Konsistenzmanagerthreads.

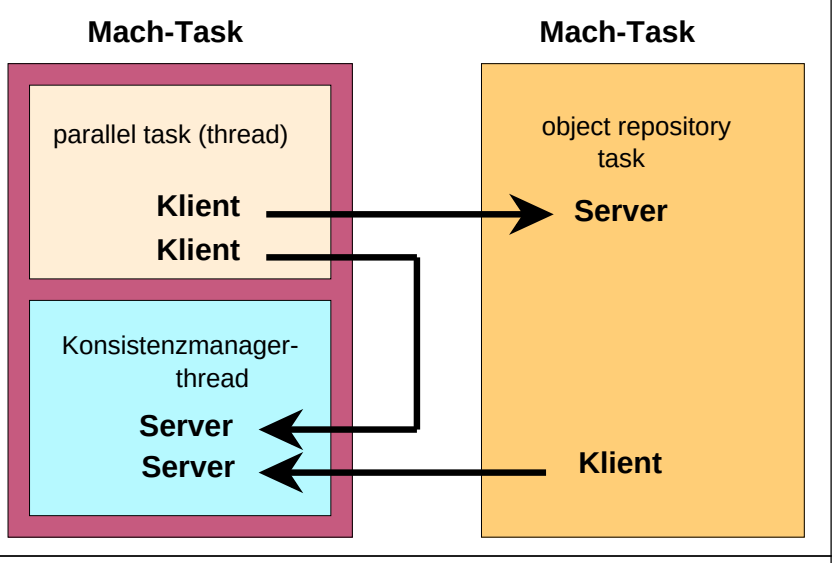

Bild 2*: verschachtelte Klient/Server-Strukturen*

Zwischen den Mach-Tasks, die parallelen Programmthread und den Konsistenzmanager enthalten, gibt es weitere Kommunikationsbeziehungen: Der parallele Programmthread in einer Task tritt womöglich als Klient des Konsistenzmanagers einer anderen Mach-Task auf. In einer Mach-Task müssen sich also sowohl Server- als auch Klientenanteil des Konsistenzmanagers befinden. Mit der MIG-Direktive `ServerPrefix = Serv_;` läßt sich erreichen, daß die in Server und Klient gerufenen Funktionen des Konsistenzmanagers unterschiedliche Namen tragen.

Bild 2 zeigt die Kommunikationsstrukturen zwischen den parallelen Tasks und dem *object repository* im *Shared Objects Memory*-System. Sowohl die Programmierschnittstelle zum *object repository* wie auch die zu den Konsistenzmanagerthreads sind in Form von MIG-Beschreibungsdateien spezifiziert. Wir zeigen in Bild 3 einen Ausschnitt aus der Spezifikation der Schnittstelle zum Konsistenzmanager.

```
subsystem cons_manag 0;

# include "sh_obj_types.defs"

import <mach/cthreads.h>;

ServerPrefix    Serv_;

routine obtain_object_values(
  RequestPort owner:    port_t;        /* object owner's port     */
  in          objID:    int;           /* object's identifier     */
  in          obj_addr: vm_address_t;  /* object's remote address */
  out         data:     object_data_t; /* the object's data       */
  in        deny_write: int            /* remove write rights     */
);
```

Bild 3*: Ausschnitt aus cons_manag.defs*

3. Typspezifikationen für MIG

An vielen Stellen kommt man beim Umgang mit dem *Mach Interface Generator (MIG)* mit den Standardtypdefinitionen aus, die in den Dateien `/mach/std_types.defs` und `/mach/mach_types.defs` oberhalb von `/usr/include` zu finden sind. Im *Shared Objects Memory*-System entstand die Notwendigkeit, Objekte, Datenstrukturen beliebiger Größe, als Argumente von MIG-Routinen verwenden zu können. Wir haben dafür den auch in Bild 3 verwendeten MIG-Typ `object_data_t` definiert.

MIG unterstützt verschiedene Arten von Typdeklarationen, darunter:

- Einfache Typen: MIG Typprimitiven.
- Strukturierte Typen: Felder, Strukturen, Zeiger.
- Polymorphe Typen: C-*union*s.

Jede MIG-Deklaration ordnet einem C-Typ eine Repräsentation zu, die MIG beim Versenden von Nachrichten benutzt. Von allen komplexen C-Typen, die in einer Spezifikation auftauchen, nimmt MIG an, daß sie im Klient- und im Serverprogramm gleichlautend deklariert sind. Diese Annahme wird durch MIG jedoch nicht überprüft.

Deklarationen von MIG-Typen müssen folgender Syntax genügen:

type	::=	**type** typename **=** typedesc [translation]**;**. | **type** typename **= polymorphic ;**.
typedesc	::=	simple-type | structured-type.
simple-type	::=	IPC-primitive | typename | compound-exp .
compound-exp	::=	**(** IPC-primitive **,** size [**, dealloc**] **)**.
structured-type	::=	array-type | struct-type | pointer-type.
array-type	::=	**array [** num-elem **] of** typedesc | **array [* :** size **] of** typedesc.
struct-type	::=	**struct [** num-elem **] of** typedesc.
pointer-type	::=	**^** typedesc.
size	::=	C-integer expression.
num-elem	::=	C-integer expression | *****.

Bild 4: *MIG Typedeklarationen*

Der einfachste Fall einer Typdeklaration (simple-type) ordnet einem C-Typnamen (`typedef`) einen primitiven IPC-Typ zu. `MSG_TYPE_INTEGER_32`, `MSG_TYPE_CHAR` oder `MSG_TYPE_PORT` sind solche primitiven IPC-Typen. Eine vollständige Liste aller primitiven IPC-Typen stellen wir im Anhang vor. Auch ein bereits deklarierter Typname kann nachfolgend anstelle eines primitiven Typs benutzt werden.

Alle primitiven Typen bis auf `MSG_TYPE_STRING` und `MSG_TYPE_REAL` haben eine wohldefinierte Größe. Für Zeichenketten und reelle Zahlen ist dagegen eine explizite Größenangabe (in Bits) mit einem *compound expression* nötig. Neben der Typ- und Größenangabe können *compound expression*s einen zusätzlichen Parameter enthalten, der

angibt, daß der von einem Datum belgte Speicher nach erfolgter Kommunikation freigegeben werden soll. Beispiele für einfache Typen sind:

```
type int = MSG_TYPE_INTEGER_32;
type status_t = int;
type time_string = ( MSG_TYPE_STRING, 8*26 );
type tmp_string = (MSG_TYPE_STRING, 8*80, dealloc );
```

Komplexe Typdeklarationen können Felder, Strukturen und Zeiger beschreiben. Die Beschreibungen von Feldern und Strukturen sehen sehr ähnlich aus. Unterschiede bestehen beim Umgang mit Parametern dieser Typen. MIG generiert speziellen Code für Zuweisungen an als Feld deklarierte Parameter — solche Parameter werden per Referenz übergeben. Strukturen werden dagegen als Wertparameter übergeben.

MIG bietet keine Möglichkeit, Komponenten unterschiedlicher Größe in einer Struktur zu spezifi zieren. Der einzige Ausweg besteht darin, eine einzige große Struktur anstelle einer Struktur mit unterschiedlich großen Komponenten anzugeben. Wie groß diese Struktur sein muß, muß der Programmierer selbst herausfi nden.

Eine spezielle Variante sind Felder variabler Größe. Anstelle der Zahl der Feldelemente muß hier eine obere Grenze für die Größe des Feldes angegeben werden. MIG generiert dann Code für die Übertragung eines Feldes mit der angegebenen maximalen Größe. C-Funktionen, die Felder variabler Größe benutzen, erhalten zusätzliche Parameter, mit denen der ‘‘Füllstand’’ der Felder beschrieben wird. Beispiele für die Deklaration von Feldern und Strukturen können so aussehen:

```
type int_array = array [ 5 ] of MSG_TYPE_INTEGER_32;
type char_array = array [ * : 4096 ] of char;
type big_info = struct [ 10*5 ] of int;
type str = struct [ 10] of array [ 50 ] of MSG_TYPE_CHAR;
```

Benutzt eine MIG-Routine nur einfache Parameter, Felder oder Strukturen, so werden bei Aufruf der MIG-generierten C-Funktion auf Klientenseite die Werte aller Parameter zu einer einzigen großen Mach-Nachricht komponiert. Solche Nachrichten werden zwischen Klient und Server kopiert, diese Art der Übertragung ist für große Datenmengen ineffi zient.

Ausweg bietet das Konzept der *out-of-line*-Daten. Solche Daten werden durch einen Zeiger aus einer Mach-Nachricht heraus referenziert. Im Fall lokaler Kommunikation bildet Machs Speicherverwaltung den referenzierten Speicherbereich (in Vielfachen von Seiten) direkt in den Adreßraum des Empfängers ab. Dort tauchen die *out-of-line*-Daten als frisch allozierte Speicherseiten auf, es liegt in der Verantwortung des Empfängers der Nachricht, den Speicher zu gegebener Zeit wieder freizugeben (mit `vm_deallocate`). Speicherverwaltung und Interprozeß-Kommunikation sind hier miteinander integriert.

Die Übertragung von *out-of-line*-Daten kann lokal sehr schnell bewerkstelligt werden. Beim Empfänger erhalten die ‘‘übertragenen’’ Speicherseiten das Prädikat *copy-on-write*. Solange die empfangenen Daten nicht verändert werden, fi ndet also überhaupt kein

Kopieren statt. Dagegen wird bei der Übertragung an einen entfernten Emfänger die gesamte Nachricht kopiert, auch dann, wenn der Empfänger nur auf Teile davon zugreift.

Auch MIG läßt die Spezifikation von *out-of-line*-Parametern zu. Hierfür existiert das Konzept der Zeigertypen. Einige Beispiele für Zeigertypen können so aussehen:

```
type int_ptr = ^ MSG_TYPE_INTEGER_32;
type some_ptr = ^ array [ 10000 ] of int;
type large_char_array = ^ array [] of MSG_TYPE_BYTE;
```

Jeder beliebige MIG-Typ kann als Basis für die Deklaration eines Zeigertyps dienen. Da Zeigertypen stets *out-of-line* übermittelt werden und dabei mindestens eine Speicherseite transportiert wird, sollten möglichst ‘‘große’’ Typen Basis für Zeigertypen sein. Benutzt man Felder oder Strukturen in Zusammenhang mit Zeigern, so kann das *size*-Feld mit einem * belegt oder freigelassen werden. In beiden Fällen wird angezeigt, daß es sich um einen Typ variabler Größe handelt, MIG generiert einen zusätzlichen Parameter zur Größenangabe für Funktionen, die diesen Typ benutzen.

Mitunter benutzen Funktionen ein einzelnes Argument, um verschiedene Datentypen zu repräsentieren. MIG bietet für diesen Fall polymorphe Typen an. Der beim Aufruf einer Funktion mit einem polymorphen Parameter tatsächlich vorliegende Typ muß in einem zusätzlichen Argument angegeben werden. Diese Typinformation wird dann auch der auf der Serverseite gerufenen Funktion zur Verfügung gestellt.

Eine weitere, exotische Fähigkeit von MIG besteht darin, die Deklaration von *translations* zu unterstützen. Mit einer *translation* kann man einen Typ vereinbaren, der auf Klientenseite eine andere Repräsentation als auf Serverseite hat. Wie die Transformation der Werte erfolgen muß, kann mit speziellen Konvertierungsfunktionen beschrieben werden. Diese Möglichkeit ist so exotisch und wird so selten benutzt, daß wir hier nicht darauf eingehen.

4. Objekte als dynamische Datenstrukturen

Nachdem die von MIG gebotenen Möglichkeiten zur Typspezifikation klar sind, wollen wir nun unsere Lösung für die Übertragung von Objekten im *Shared Objects Memory* vorstellen.

In vielen Anwendungen, die mit Objekten hantieren, sind Objekte relativ kleine Dateneinheiten — in der Größenordnung von wenigen Bytes bis zu einem KByte. Wir gehen davon aus, daß in parallelen Programmen gemeinsam benutzte (*shared*) Datenstrukturen ebenfalls von geringer Größe sind. Diese Datenstrukturen können ja von mehreren Prozessen nur sequentiell geschrieben werden — ihre Größe beeinflußt das erreichbare Maß an Parallelität in einem Programm.

Wir unterstützen daher im *Shared Objects Memory*-System zunächst nur Objekte bis zu einer Größe von 2 KByte. Daten dieser Größ können ohne große Effizienzeinbußen mit gewöhnlichen (*inline*) Mach-Nachrichten transportiert werden. *Out-of-line*-Daten können zwischen Tasks auf demselben Rechner zwar effizienter (gänzlich ohne Kopieren) übertragen werden, bei Kommunikation mit einem entfernten Partner wird aber zumindest eine Speicherseite (8KByte bei NeXTSTEP auf HP-PA) kopiert. Generell muß man

sagen, daß keine klare Regel existiert, die besagt, wann *out-of-line*-Daten anstelle gewöhnlicher *inline*-Daten in Mach-Nachrichten verwendet werden sollten.

Bild 5 zeigt schließlich die Deklaration des MIG-Types `object_data_t`:

```
# include <mach/std_types.defs>
# include <mach/mach_types.defs>

type object_data_t = array [ *: 2048 ] of char;

import "sh_obj_types.h";
```

Bild 5*: Ausschnitt aus sh_obj_types.defs*

Wir transportieren Objekte zwischen kooperierenden Tasks also als *inline*-Daten. Mit einer `import`-Direktive wird dafür gesorgt, daß alle von MIG generierten C-Quellen, die den MIG-Datentyp `object_data_t` benutzen, auch den korrespondierenden C-Typ einschließen.

Nun könnte man eine Verbesserung unseres Ansatzes vorschlagen und für den Umgang mit großen Objekten (> 2KByte) zusätzlich einen *out-of-line*-Typ definieren. In der Tat unterstützt MIG diese Idee: `object_data_t` müsste als polymorpher Typ definiert werden. Kleine Objekte könnten in einer Array-Repräsentation als *inline*-Daten übertragen werden, große Datenobjekte würden als *out-of-line*-Daten repräsentiert. Einziger Nachteil bei diesem Verfahren ist dann, daß bei jeder Verwendung des polymorphen Typs die Größe der aktuellen Instanz spezifiziert werden muß.

Ein kleiner Trick ist noch nötig, um C++-Objekte von einem Adreßraum zu einem anderen zu befördern. Sicherlich kann man etwa mit `memcpy()` ein generisches Objekt in eine Mach-Nachricht kopieren, um es dann zu verschicken. Auf Empfängerseite funktioniert `memcpy()` jedoch nicht: Objekte können Zeiger zu virtuellen Methodentabellen enthalten, und diese Zeiger sind natürlich nur in einem Adreßraum gültig. Mit der folgenden virtuellen Funktion

```
virtual void* assign( void* p ) { *this = *((<class_name>*) p); }
```

kann der Zuweisungsoperator benutzt werden, um den Inhalt eines Speicherbereiches auf ein Objekt zuzuweisen. Der vom C++-Compiler generierte Zuweisungsoperator kopiert Objekte komponentenweise, Zeiger zu Methodentabellen werden dabei korrekt initialisiert. Für benutzerdefinierte Zuweisungsoperatoren muß gefordert werden, daß sie ohne Seiteneffekte arbeiten. Weitere Ansätze zum Kopieren von Objekten zwischen Adreßräumen werden in [Bilris et al. 93] diskutiert.

5. Port Sets

Werfen wir einen Blick zurück auf die Gesamtstruktur des *Shared Objects Memory*-Systems. Sowohl die *object repository*-Task als auch die verschiedenen Konsistenzmanagerthreads treten als Server etlichen Klienten simultan gegenüber. Eine einfache Möglichkeit, alle diese Klienten zu unterscheiden, besteht darin, jedem Klient einen eigenen Port für die Kommunikation mit dem Server zuzuweisen. Allerdings ist nun zur

Behandlung eines jeden Ports ein eigener Thread nötig. Mach kennt kein dem UNIX-Systemaufruf `select()` vergleichbares Konstrukt zur Behandlung asynchroner Kommunikation in einem einzigen Thread.

Dagegen bietet Mach die Möglichkeit, mehrere Ports zu einem *Port Set* zusammenzufassen. `msg_receive()`-Operationen können auch mit Port Sets hantieren —es wird dann die erste an einem Port verfügbare Nachricht empfangen.

Folgende Regeln gelten für den Umgang mit Port Sets:

- Neu angelegte Ports gehören zu keinem Port Set.
- Neu angelegte Port Sets sind leer; kein einziger Port gehört zu einem solchen Port Set.
- Ein Port kann zu jedem Zeitpunkt höchstens zu einem Port Set gehören.
- Gehört ein Port zu einem Port Set, so kann er keine Nachrichten als individueller Port mehr empfangen.
- Eine Task muß Empfangsrechte für einen Port besitzen, bevor sie diesen Port einem Port Set hinzufügen kann.

Es existieren vier Funktionen für den Umgang mit Port Sets, sie werden im folgenden Codefragment demonstriert:

```
task_t           target_task;
port_set_name_t  port_set;
port_name_t      port;
kern_return_t    ret;

ret = port_set_allocate(target_task, &port_set);

ret = port_set_add(target_task, port_set, port);

ret = port_set_remove(target_task, port);

port_name_array_t       parray;
unsigned int            count;

ret = port_set_status(task_self(), port_set,  &parray, &count);
```

Bild 6*: Umgang mit Port Sets*

Ports können auf zwei verschiedene Weisen aus einem Port Set verschwinden: Einerseits kann der Programmierer explizit einen Port aus einem Port Set löschen (mit `port_set_remove()`), andererseits entfernt der Kern einen Port, wenn eine Task ihre Empfangsrechte an diesem Port abgibt. Diese Eigenart erklärt den Wert der Funktion `port_set_status`: hier kann man erfragen, welche Ports zu einem Port Set gehören. Alle Portnamen (Integer-Werte) werden in einem Feld zurückgegeben, den Speicherplatz dafür alloziert der Kern. Es liegt in der Verantwortung des Programmierers, den Speicher zu gegebener Zeit (mit `vm_deallocate`) wieder freizugeben.

6. Zusammenfassung

Mit *Shared Objects Memory* haben wir ein *Distributed Shared Memory*-System auf der Basis von replizierten C++-Objekten kurz vorgestellt. Das System unterstützt mehrere (schwache) Konsistenzprotokolle. Das Ausmaß an Kommunikation zur Aktualisierung der Replikate eines Objekts kann damit minimiert werden. Weiterhin können sich Kommunikation und Berechnung in einer parallelen Task überlappen. *Shared Objects Memory* unterstützt die Abarbeitung paralleler Programme in einem Netzwerk Mach-basierender Workstations.

Wir haben uns hier auf die Kommunikationsbeziehungen zwischen parallelen Tasks oberhalb des *Shared Objects Memory*-Systems konzentriert. In unserem Ansatz entstehen mehrere, ineinander geschachtelte Klient/Server-Strukturen, die durch Definitionen für den *Mach Interface Generator* (MIG) spezifiziert worden sind. Die Benutzung von Objekten dynamischer Größe in Zusammenhang mit MIG erfordert die Deklaration eines komplexen Datentyps. Wir haben uns mit den Möglichkeiten auseinandergesetzt, die MIG zur Typspezifikation bietet. Schließlich kam ein weiterer Trick beim Umgang mit Mach-IPC zur Sprache: die Benutzung von Port Sets zur Behandlung asynchroner Nachrichten von mehreren Absendern in einem einzigen Empfämgerthread.

Liste der primitiven MIG-Typdefinitionen

MSG_TYPE_UNSTRUCTURED	MSG_TYPE_BIT
MSG_TYPE_BOOLEAN	MSG_TYPE_INTEGER_16
MSG_TYPE_INTEGER_32	MSG_TYPE_PORT_OWNERSHIP
MSG_TYPE_PORT_RECEIVE	MSG_TYPE_PORT_ALL
MSG_TYPE_PORT	MSG_TYPE_CHAR
MSG_TYPE_BYTE	MSG_TYPE_INTEGER_8
MSG_TYPE_REAL	MSG_TYPE_STRING
MSG_TYPE_STRING_C	

Literatur

[Bershad et al. 91] B.N.Bershad and M.J.Zekauskas;
Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors; Technical Report CMU-CS-91-170, School of CS, Carnegie Mellon University, September 1991.

[Bilris at al. 93] A.Bilris, S.Dar, N.H.Gehani;
Making C++ Objects Persistent: the Hidden Pointers; Software-Practice and Experience, Vol. 23(12), pp. 1285-1503, December 1993.

[Boykin et al. 93] 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.

[Gharachorloo et al. 90] K.Gharachorloo, D.Lenoski, J.Laudon, P.Gibbons, A.Gupta, and J.L.Hennessy;
Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocesors; in Proceedings of the 17th Annual Symposium on Computer Architecture, pp. 15-26, May 1990.

[Li/Hudak 89] K.Li, P.Hudak;
Memory Coherence in Shared Virtual Memory Systems; in ACM Trans. Computer Systems, Vol. 7, No. 4, pp. 321-359, Nov. 1989.

[Nitzberg/Lo 91] N.Nitzberg, V.Lo;
Distributed Shared Memory: A Survey of Issues and Algorithms; IEEE Computer, August 1991, pp. 52-60.