Entwicklung und Implementierung eines verteilten File-Assigners und File-Servers für das verteilte Realzeit-Betriebssystem MELODY unter Linux

Horst F. Wedde, Sebastian Lehnhoff, Nils Kemper
und Mohammed Benchelih

Technical Report #817

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung: Unter einem Realzeitbetriebssystem versteht man ein Betriebssystem mit deterministischem Zeitverhalten. Typischerweise werden von außen (durch Sensoren, Benutzereingaben, usw.) ein oder mehrere Tasks erzeugt, auf die das System innerhalb einer festen Zeit reagiert haben muss. Hinzu kommt ein sicherheitskritischer Aspekt: bei Nichteinhaltung der Frist von sogenannten essentiell kritischen Tasks kann das Überleben des Gesamtsystems in Frage gestellt sein, d.h. es könnte schwer beschädigt oder gar zerstört werden. Man spricht hier von funktionskritischen Systemen (*mission critical systems*). Beispiele für solche Systeme sind Kontrollsysteme für Flugzeuge, Kraftwerke oder Industrieroboter. MELODY ist ein verteiltes Realzeitbetriebssystem für solche funktionskritischen Systeme, in dem lokal verteilte Tasks mit unterschiedlichen Prioritäten und Deadlines um verteilte Ressourcen konkurrieren. Die einzelnen, periodischen Tasks sind jeweils an einzelne Knoten gebunden, d.h. es findet keine Prozessmigration statt. Die Ressourcen hingegen können von einem Knoten auf den anderen kopiert oder verlagert werden, um ihre Verfügbarkeit zu erhöhen. In diesem Kontext spricht man von Replikation bzw. Relokation.

Die vorliegende Arbeit beginnt mit einer kurzen Vorstellung von MELODY und dessen einzelnen Systemkomponenten. Danach folgt eine Darstellung des Systemdesigns und der Entwicklungsschritte des Systems. Im Anschluss daran folgen umfangreiche verteilte Experimente und eine Diskussion der Ergebnisse.

1 MELODY

MELODY ist ein verteiltes Realzeitbetriebssystem für zeitkritische Anwendungen, in dem lokal verteilte Tasks mit unterschiedlichen Prioritäten und Fristen um verteilte Ressourcen konkurrieren (siehe Abb. 1.1). Die einzelnen, aperiodischen Tasks sind jeweils an einzelne Knoten gebunden, d.h. es findet keine Prozessmigration statt. Die Ressourcen hingegen können von einem Knoten auf den anderen kopiert oder verlagert werden, um ihre Verfügbarkeit zu erhöhen. Hierbei spricht man von Replikation bzw. Relokation.

Die Entwicklung des MELODY Projektes begann bereits Ende der achtziger Jahre als For-

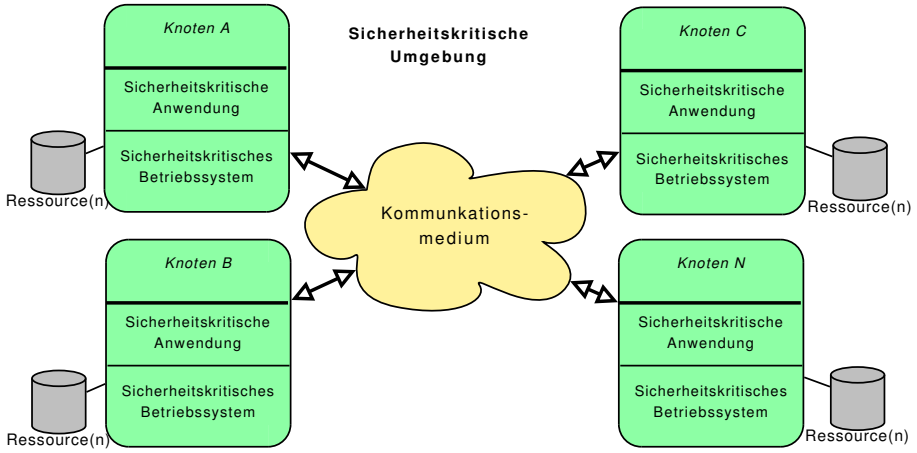

Abbildung 1.1: Ein verteiltes Realzeitsystem

schungsprojekt an der Wayne State University in Detroit, Michigan (USA), [ALIJANI 1988] und wurde erstmals als Prototyp auf einem Token-Ring aus 6 IBM RS/6000 Maschinen am Lehrstuhl 3 des Fachbereichs Informatik der Universität Dortmund getestet. Ergebnisse aus verschiedenen Phasen der Entwicklung findet man u.a. in [WEDDE et al. 1990], [WEDDE und XU 1992], [WEDDE et al. 1993], [WEDDE et al. 1994] und [WEDDE und LIND 1997] dokumentiert.
MELODY lässt sich am besten durch die Beschreibung seiner drei hervorstechendsten Konzepte darstellen: das Ressourcen-Modell, das Task-Modell und die Systemarchitektur.

1.1 Ressourcen-Modell in MELODY

Ein wichtiger Gesichtspunkt bei der Verwendung eines verteilten Betriebssystems ist die Frage, wie der gleichzeitige Zugriff auf die verteilten Ressourcen geregelt wird (*concurrency control*). Ein anderer ist die Frage, wie die Ressourcen am besten über das betrachtete System zu verteilen sind, also z.B. wie groß die Anzahl der Kopien sein soll und ob diese Verteilung während der Laufzeit noch verändert wird (*dynamisch*) oder nicht (*statisch*).
Bei der Konsistenz unterscheidet man zwischen starker bzw. wechselseitiger (*strong consistency*) und schwacher Konsistenz (*weak consistency*). Die erste bietet fehlerfreien Datenzugriff, d.h. man kann sich darauf verlassen, die aktuellste Version der Ressource zu erhalten, erkauft sich das allerdings mit längeren Antwortzeiten, da immer wieder Synchronisationen durchgeführt werden müssen. Dagegen kommt das schwache Konsistenzmodell mit einem sehr viel geringeren Kommunikations-Overhead aus, allerdings wird nicht für jeden Fall garantiert, die aktuellste Ressourcenversion zu erhalten.
MELODY verwendet beide Konsistenzmodelle parallel: es gibt sogenannte öffentliche Kopien (***public copies***), die die wechselseitige Konsistenz bieten und private Kopien (***private copies***), die, da lokal verfügbar, schnell zugänglich sind, aber nur schwache Konsistenz bieten. Darüberhinaus gibt es noch zu jeder *public copy* eine sogenannte Schattenkopie (***shadow copy***), die eine *Read-only*-Kopie der letztgültigen Information ist.
Public copies stellen eine gemeinsam genutzte (*shared*) Ressource in MELDOY dar. Alle Schreiboperationen werden stets auf den *public copies* durchgeführt. Dazu werden alle im Netz vorhandenen Kopien dieser Ressource gesperrt. Diese Aufgabe übernimmt der *File-Server* (siehe Abschnitt 1.3.6) durch Verwendung des sogenannten *Delayed Insertion Protocols*, einem Protokoll, welches sicherstellt, dass vor jeder Aktualisierung simultan alle Ressourcen-Kopien geblockt werden (siehe Abschnitt 1.3.9). Zusätzlich ist die Erneuerung der Schattenkopie, auf die stets bei Leseoperationen zugegriffen wird, lokal-atomar mit diesem Update verbunden.
Private copies verkörpern dagegen nicht-gemeinsam nutzbare Ressourcen, die nur von sogenannten lokal-robusten oder lebenswichtig-kritischen (*essentially critical*) Tasks für lokale Leseoperationen verwendet werden dürfen. Bei diesen ist nicht gewährleistet, dass die Kopie mit der aktuellsten im Netz verfügbaren Ressourcen-Version übereinstimmt, und darf deshalb nur in den erwähnten Ausnahmefällen genutzt werden. Der File-Server garantiert, die Kopie nach jedem Update der Ressource ebenfalls zu erneuern. Diese gelockerte Eigenschaft bietet dafür den Vorteil schnellerer Zugriffszeiten, da das Erstellen und die Pflege einer lokalen Privatkopie selbstverständlich weniger aufwendig ist als das einer öffentlichen Kopie oder dem Zugriff auf eine entfernte Kopie.
Es ist leicht einzusehen, dass, je mehr *public copies* in dem Netz vorhanden sind, es umso aufwendiger ist, die gleichzeitige Aktualisierung aller durchzuführen, weshalb man versucht ist, die Anzahl der öffentlichen Kopien so klein wie möglich zu halten. Dem steht gegenüber, dass ein Zugriff auf eine entfernte Kopie um einige Faktoren langsamer ist, als der Zugriff auf eine lokal vorhandene, weshalb jeder Knoten darum bemüht ist, möglichst viele der benötigten Ressour-

cen vor Ort zu erhalten. Dieses Balanceproblem begegnet man in MELODY dynamisch durch den Einsatz eines *File-Assigners*, der die Verteilung der Kopien im Netz adaptiv steuert (siehe Abschnitt 1.3.3).

1.2 Task-Modell in MELODY

Aufgrund der unvorhersehbaren Umweltbedingungen, die für die meisten funktionskritischen (*safety-critical*) Anwendungen typisch sind, geht man von sich aperiodisch wiederholenden Tasks aus. Jeder Task enthält einen kritischen Abschnitt (*critical section*), während dessen er auf eine oder mehrere Ressourcen gleichzeitig lesend oder schreibend zugreifen muss. Darüberhinaus geht man davon aus, dass die Ausführungszeit (*computation time*) in engen Grenzen abgeschätzt werden kann.
Traditionell wird ein Betriebssystem so gestaltet, dass die Ressourcen-Akquise vor dem eigentlichen Task-Scheduling durchgeführt wird. Erst wenn alle benötigten Ressourcen bereitgestellt sind, wird der Task vom Scheduler betrachtet und in den Ablaufplan eingefügt. Währenddessen sind diese Ressourcen für konkurrierende Tasks geblockt.
Übertragen auf MELODY bedeutet dies, dass ein Großteil der tatsächlichen Taskausführungszeit mit dem Warten auf verteilte (und damit oft entfernte) Ressourcen verbracht wird. Das dabei auftretende sogenannte *remote blocking* ist für einen lokalen Scheduler unkontrollierbar, so dass es durchaus dazu kommen kann, dass trotz letztendlich erhaltener Ressourcen nur noch das Verpassen der Deadline festgestellt werden kann.
Um diesem Problem aus dem Weg zu gehen, wird in MELODY die Reihenfolge der Ressourcen-Allokation und des Task-Scheduling vertauscht. Dadurch sollen die Tasks so früh wie möglich abgebrochen und die Ressourcen so spät wie möglich geblockt werden können. Ein unmittelbarer Nachteil dieser Entscheidung ist, dass dem TS nur noch ungefähre Ausführungszeiten als Planungsgrundlage bleiben, da die Zeit für die Ressourcen-Akquise nur abgeschätzt werden kann. Dieses Problem wird in 1.3 ausführlich besprochen.
Die *k*-te Inkarnation einer Task j T_{jk} ist erfolgreich, wenn sie folgende vier Phasen vor Erreichen ihrer Deadline durchlaufen hat (siehe auch Abb. 1.2):

Scheduling-Phase Task-Inkarnation T_{jk} wird in vom Task-Scheduler in den Ausführungsplan (*execution schedule*) eingefügt

Resource-Location-Phase Alle von T_{jk} benötigten Ressourcen werden vom File-Server lokalisiert

Resource-Allocation-Phase Die benötigten Ressourcen werden vom File-Server exklusiv reserviert

Computation-Phase Die Task-Inkarnation T_{jk} führt ihre Schreib-/Lesezugriffe unter Zuhilfenahme des File-Servers durch

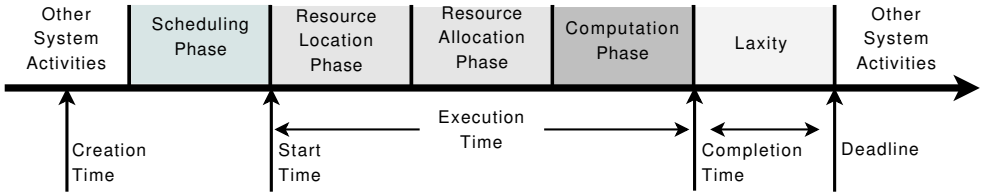

Abbildung 1.2: Ausführungs-Phasen einer Task-Inkarnation

Die eigentliche Ausführungszeit umfasst den Zeitraum zwischen dem Beginn der Ressourcen-Lokalisierung und dem Ende der Ressourcen-Zugriffe. Der Spielraum (*laxity*) einer Task-Inkarnation ist der Zeitraum zwischen dem Ende der Ressourcen-Zugriffe und der eigentlichen Deadline, bis wann die Inkarnation spätestens beendet sein müsste.
Darüber hinaus werden die Task-Inkarnationen zusätzlich mit zwei Eigenschaften charakterisiert: *Kritikalität* (*criticality*) und *Sensitivität* (*sensitivity*).

1.2.1 Criticality

Die Wichtigkeit von Aufgaben, die an ein sicherheitskritisches System gestellt werden, variiert in der Regel. Die sogenannte *criticality* beschreibt eben diese Wichtigkeit einer Task-Inkarnation für das Überleben des Systems. Jeder Task erhält bei seiner Initialisierung hierfür einen Wert, der sich, wie in der folgenden Definition beschrieben, in Abhängigkeit von Erfolg oder Misserfolg der vorherigen Inkarnation erhöht oder erniedrigt.

Definition 1.1 *Für jeden Knoten I bilden zwei Schwellenwerte a_i' und a_i'' ein Kritikalitäts-Fenster für alle an I auftretenden Tasks. Jedem Task T_j wird ein Kritikalitäts-Wert C_j wie folgt zugewiesen:*

wenn	$C_j \geq a_i''$	*gilt, ist*	T_j	*nicht kritisch (**non-critical**),*
wenn	$a_i' < C_j < a_i''$	*gilt, ist*	T_j	*kritisch (**critical**),*
wenn	$C_j \leq a_i'$	*gilt, ist*	T_j	*hochkritisch (**essential critical**)*

Dieser Wert C_j wird als Kritikalität der initialen Task-Inkarnation vom Task T_j verstanden, und wird typischerweise experimentell im Vorfeld festgelegt. Aus diesem wird nun anhand der nachfolgenden Definition der relative Kritikalitätsgrad (*relative degree of criticality*) für jede weitere Task-Inkarnation bestimmt:

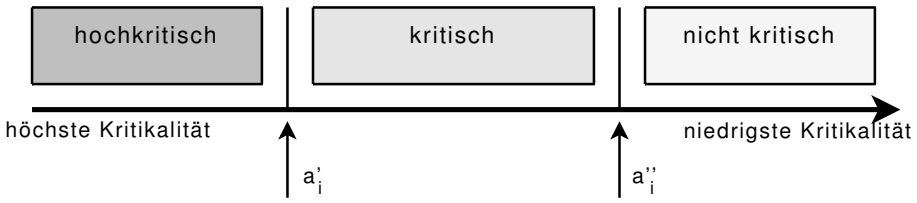

Abbildung 1.3: Kritikalitäts-Fenster

Definition 1.2 *Sei C_j die Kritikalität von Task T_j und T_{jk} die k-te Inkarnation von T_j. Dann ist die relative Kritikalität C_{jk} wie folgt als Ganzzahl-Wert definiert:*

$C_{jk} := C_j$, *wenn* $C_j \leq a'_i$ *oder* $C_j \geq a''_i$ *oder* $T_{j(k-1)}$ *war erfolgreich, ansonsten*

$C_{jk} := max\{C_j - (\textit{Anzahl erfolgloser Inkarnationen seit letzter erfolgreichen}), a'_i\}$

1.2.2 Sensitivity

Neben der Kritikalität wird in MELODY ein zweiter Wert zur Charakterisierung eines Tasks bzw. einer Task-Inkarnation, hier allerdings auf Leseoperationen beschränkt, eingeführt. Umgangssprachlich versucht man mit dieser Eigenschaft zu erfassen, dass mit zunehmender Verfehlung der Deadline die Aktualität der Ressource für den Task immer weniger wichtig wird. Formal wird der Wert, Sensitivität (*sensitivity*) genannt, wie folgt definiert:

Definition 1.3 *Für jeden Knoten I bilden zwei Schwellenwerte b'_i und b''_i ein Sensitivitäts-Fenster für alle an I auftretenden Tasks. Jedem Task T_j wird ein Sensitivitäts-Wert R_j wie folgt zugewiesen:*

wenn	$R_j \geq b''_i$	*gilt, ist*	T_j	***robust***,
wenn	$b'_i < R_j < b''_i$	*gilt, ist*	T_j	*sensitiv (**sensitive**),*
wenn	$R_j \leq b'_i$	*gilt, ist*	T_j	*höchst sensitiv (**essential sensitive**)*

Wie schon bei der Kritikalität C_j wird R_j als die Sensitivität für die initialen Task T_j begriffen, und muss passend zur Anwendung experimentell bestimmt werden. Für die nachfolgenden Task-Inkarnationen T_{jk} lässt sich dann die relative Sensivitität durch folgende Definition bestimmen:

Definition 1.4 *Sei R_j die Sensitivität von Task T_j und T_{jk} die k-te Inkarnation von T_j. Dann ist die relative Sensitivität R_{jk} wie folgt als Ganzzahl-Wert definiert:*

$R_{jk} := R_j$, *wenn* $R_j \leq b'_i$ *oder* $R_j \geq b''_i$ *oder* $T_{j(k-1)}$ *war erfolgreich,*

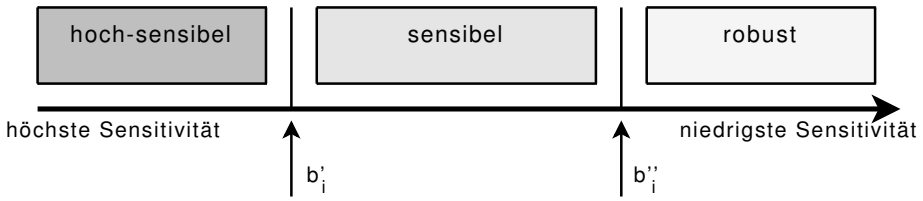

Abbildung 1.4: Sensitivitäts-Fenster

ansonsten

$$R_{jk} := min\{R_j + (\textit{Anzahl erfolgloser Inkarnationen seit letzter erfolgreichen}), b_i''\}$$

1.3 Systemarchitektur von MELODY

MELODY wird in [LIND 1999] in vier verschiedene Module pro Knoten unterteilt, die jeweils eine Einheit bilden und für spezielle Funktionalitäten verantwortlich sind.

- Task-Scheduler (TS)
- Runtime-Monitor (RTM)
- File-Server (FS)
- File-Assigner (FA)

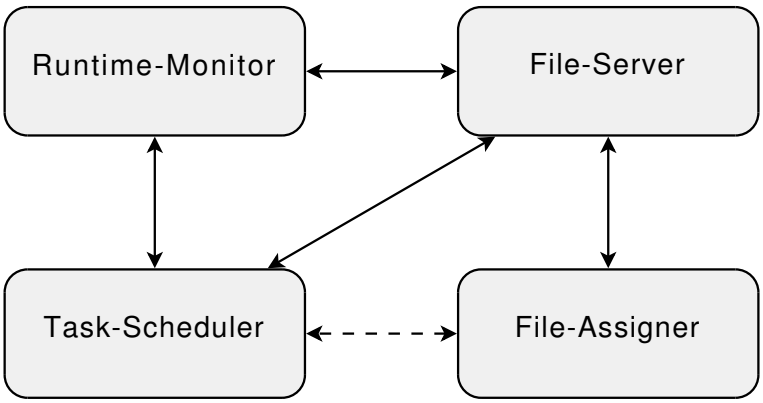

Abbildung 1.5: Module von MELODY auf jedem Knoten

Dabei kooperieren die ersten beiden lediglich mit den auf dem lokalen Knoten vorhandenen Modulen, während die zweiten auch mit entfernten Knoten interagieren. Im folgenden werden die Begriffe Ressourcen, Files und Dateien äquivalent verwendet.

1.3.1 Task-Scheduler (TS)

Der Task-Scheduler (*TS*) eines jeden Knotens verwaltet eine Menge von auf einem Knoten *I* ankommenden, periodischen Tasks T_j. Jede dieser Tasks will dabei auf eine bestimmte Zahl von sich im MELODY-System befindlichen Ressourcen schreibend oder lesend zugreifen. Dabei versucht der Task-Scheduler deren Ausführungsreihenfolge mit Hilfe der folgenden Kriterien so zu planen, dass eine größtmögliche Anzahl von Task-Inkarnationen ihre Deadline einhalten:

Estimated Execution Time (EET_{jk}) die geschätzte Ausführungszeit der k-ten Inkarnation T_{jk} von Task T_j; wird von dem File-Server aus den, gewöhnlich letzten 5, zurückliegenden Inkarnationen des Tasks T_j in der *Task-History* berechnet und dem Task-Scheduler auf Nachfrage zur Verfügung gestellt.

List of Required Files (LRF_{jk}) eine Liste mit den vom Task benötigten Ressourcen

Deadline (DT_{jk}) die Deadline der Task-Inkarnation T_{jk}

Static Priority (PrT_j) die (beim Start festgelegte) Priorität der Task T_j

Bei der Ankunft einer Task-Inkarnation T_{jk} kommuniziert der Task-Scheduler mit dem lokalen File-Server, der überprüft, welche Ressourcen aus der LRF_{jk} lokal vorhanden sind und berechnet die EET_{jk}. Daraufhin wird T_{jk}, entsprechend nach der Deadline DT_{jk} und der statischen Priorität PrT_j sortiert, in die lokale Task Queue (*LTQ*) eingefügt bzw. zur Ausführung gebracht. Weiterhin werden die Task-Inkarnationen in die drei folgenden Kategorien eingeteilt:

1. **strongly schedulable** Voraussichtlich kann T_{jk} seine Deadline einhalten und wird in die *LTQ* einsortiert und zur Ausführung gebracht.
2. **weakly schedulable** T_{jk} kann seine Deadline nicht einhalten, *könnte* es aber, wenn alle benötigten Ressourcen aus LRF_{jk} lokal vorhanden wären.
3. **non-schedulable** T_{jk} kann seine Deadline in keinem Fall einhalten.

Die Einteilung in diese Kategorien ist vor allem dann von Interesse, wenn MELODY mit der sogenannten *Task-Scheduler-Integration* arbeitet, auf die im Abschnitt zum File-Assigner auf Seite 13 eingegangen wird.
Aufgrund der Unvorhersehbarkeit von Umwelteinflüssen und daraus resultierenden aperiodischen Tasks ist das Scheduling in MELODY nicht-preemptiv.

1.3.2 Runtime-Monitor (RTM)

Wie bereits angesprochen, findet in MELODY das Task-Scheduling vor der Ressourcen-Allokation statt. Hierdurch sollen die Tasks so früh wie möglich abgebrochen und die Ressourcen so spät

wie möglich geblockt werden können. Ein unmittelbarer Nachteil dieser Entscheidung ist jedoch, dass dem TS nur noch ungefähre Ausführungszeiten als Planungsgrundlage bleiben, da die Zeit für die Ressourcen-Akquise nur abgeschätzt werden kann. Dazu werden die Ausführungszeiten von vorangegangenen Task-Inkarnationen berücksichtigt, z.B. die letzten fünf.
Da der Task-Scheduler nun nicht mehr die Einhaltung der Deadline überwacht, wird dazu der sogenannte Runtime-Monitor in MELODY bereitgestellt. Dieses Modul übernimmt nun die Aufgabe, während der Ressourcen-Lokalisation wie auch Ressourcen-Allokation die Deadline der Task-Inkarnation zu beachten und bei Verpassen dieser die Inkarnation abzubrechen. Dabei kommt dem RTM zu gute, dass man nach der Lokalisation (siehe Abb. 1.2) schon eine sehr gute Idee von der restlichen Laufzeit der Task-Inkarnation hat, da nun Anzahl und Verteilung der Ressourcen bekannt und auch die eigentliche Computation-Time aus früheren Durchläufen abgeschätzt werden kann.
Schon sowohl in frühen Simulationen [WEDDE et al. 1993] als auch in ersten Experimenten [LIND 1999] hat sich gezeigt, dass die umgekehrte Task- und Ressourcen-Schedule Strategie einen großen positiven Einfluss auf die Deadline-Performance, aber auch auf die Überlebensfähigkeit des Systems hat.
In der hier vorgestellten Entwicklungsphase der Implementation des MELODY-Systems gibt es allerdings keine seperate Runtime-Monitor-Komponente. Stattdessen wird die Überwachung der Ressourcen-Location und -Allocation-Phasen von dem File-Server übernommen. Nach jeder Phase wird verglichen, ob die bis zur Deadline verbleibene Zeit noch für die Computation-Phase ausreicht und bei einem negativen Ergebnis die Task-Inkarnation vorzeitig abgebrochen.

1.3.3 File-Assigner (FA)

In MELODY werden die Ressourcen vereinfacht als Dateien dargestellt. Wie schon vorher beschrieben befinden sie sich in meist mehrfacher Ausführung unterschiedlicher Qualität verteilt auf den einzelnen Knoten des Systems. Da es in einem sicherheitskritischen Realzeitsystem besonders auf möglichst schnelle Operationen auf den Ressourcen ankommt, besteht ein Konflikt bezüglich der Anforderungen von Lese- und Schreibzugriffen. Für Leseoperationen wäre es am günstigsten, wenn jede der auf dem Knoten benötigte Datei lokal verfügbar ist, wodurch ein schnellstmöglicher Zugriff möglich wäre. Dagegen bedeutet eine große Anzahl von Kopien für die Schreiboperation einen erhöhten zeitlichen Aufwand, da die Ressourcen konsistent gehalten werden müssen.
Die Aufgabe des File-Assigners besteht nun darin, adaptiv auf den typischerweise nicht vorhersagbaren, aktuellen Systemzustand zu reagieren und eine möglichst optimale Verteilung der Kopien zu erreichen. Zu diesem Zweck überwacht der FA eines jeden Knotens die Nutzung der verschiedenen Ressourcen und entscheidet, ob eine der drei möglichen Aktionen durchgeführt werden kann:

- Replikation (*replication*): Lokale Erzeugung einer neuen öffentlichen Kopie an dem Kno-

ten

- Relokation (*relocation*): Die Kopie einer Ressource wird von einem Knoten zu einem anderen Knoten verschoben
- Löschen (*deletion*): Entfernen einer nicht mehr benötigten lokalen Kopie vom Knoten

Zur Auswahl der besten Alternative arbeitet der File-Assigner eng mit dem lokalen File-Server, den entfernten FAs und u.U. dem Task-Scheduler zusammen. So greift der lokale FA auf die Historie der erfolgreichen und nicht-erfolgreichen Zugriffe zurück, die vom File-Server verwaltete wird. Speziell beim Löschen von Kopien muss per Rückfrage auf alle entfernten FAs sichergestellt werden, dass eine minimale Anzahl von Kopien einer Ressource erhalten bleibt, damit eine ausreichende Zuverlässigkeit des Systems garantiert bleibt.
Bei MELODY gibt es zwei Möglichkeiten den File-Assigner in den Ablauf des Systems zu integrieren: entweder verwendet man die sogenannte Task-Scheduler-Integration, bei der der Task-Scheduler den File-Assigner am Ende der Scheduling-Phase aufruft, oder die File-Server-Integration, bei der der File-Server den FA nach Beendigung der Computation-Phase anstößt. Beide Integrationsmodelle werden in Abbildung 1.6 dargestellt und in [WEDDE et al. 1996] beschrieben. Es wird erwartet, dass ein früheres Anstoßen des File-Assigners bessere Resultate für nachfolgende Task-Inkarnationen liefert, da die Anpassung der Ressourcenverteilung schneller abgeschlossen ist. Auf der anderen Seite ermöglicht ein späterer Aufruf des File-Assigners eine präzisere Entscheidung, welche Maßnahmen zur Verbesserung der Überlebensfähigkeit des Systems (*Survivability*) ergriffen werden müssen.

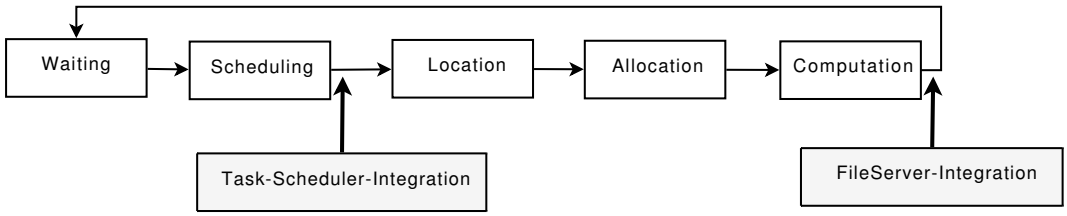

Abbildung 1.6: Mögliche File-Assigner-Integrations Modelle

1.3.4 Task-Scheduler-Integration

In diesem Modell verwaltet der Task-Scheduler drei Warteschlangen, auf welche die Task-Inkarnationen verteilt werden, die, wie auf Seite 12 schon erwähnt, in die Kategorie der *weakly schedulable Inkarnationen* fallen: eine für die *weakly schedulable sensitive read tasks* (*WSRQ*), eine für *weakly schedulable sensitive write tasks* (*WSWQ*) und eine weitere für die sogenannten *weakly schedulable robust read tasks* (*WRRQ*). Da jede Task für einen festgelegten Zeitraum in der jeweiligen Queue gehalten wird, hat man somit immer einen Satz von (in einem Zeitfenster

gescheiterten) Task-Inkarnationen vorliegen.
Zu jeder Queue hat man vorher einen Grenzwert (*threshold*) bestimmt, der festlegt, ab welcher Anzahl an Tasks der TS am Ende des Task-Scheduling den File-Assigner mit einer Änderung der Kopien-Verteilung zu beauftragen, und zwar nach folgenden Regeln:

- zu viele Elemente in der WSRQ ⇒ erzeuge lokale öffentliche Kopien der von den Task-Inkarnationen aus der WSRQ benötigten Ressourcen ⇒ der Lesezugriff kann zukünftig lokal durchgeführt werden
- zu viele Elemente in der WSWQ ⇒ verringere die Zahl öffentlicher Kopien der von den Task-Inkarnationen aus der WSWQ benötigten Ressourcen ⇒ es müssen weniger Kopien konsistent gehalten werden
- zu viele Elemente in der WRRQ ⇒ erzeuge lokale private Kopien der von den Task-Inkarnationen aus der WRRQ benötigten Ressourcen ⇒ der Lesezugriff kann zukünftig lokal durchgeführt werden

Die Anfrage für den letzten Fall geht direkt an den File-Server, da für die Erzeugung einer privaten Kopie kein anderer Knoten seine Zustimmung geben muss.

1.3.5 File-Server-Integration

Wie der Name bereits vermuten lässt wird in diesem Modell der File-Assigner von dem File-Server mit Aufträgen versorgt, und zwar erst nach Beendigung einer Task-Inkarnation und der Aktualisierung der (noch zu beschreibenden) *File-History*. Auch hier gibt es drei verschiedene Warteschlangen, in diesem Fall für die Task-Inkarnationen, die ihre Deadline verpasst haben: eine mit den *sensitive read tasks* (*FSR*), eine für *write tasks* (*FSW*) und eine für *robust read tasks* (*FRR*). Auch hier bleiben die Task-Inkarnationen nur in einem festen Zeitrahmen in der jeweiligen Schlange und so erhält man über die Anzahl der Elemente einen Grenzwert, ab wann der File-Server den File-Assigner zum Handeln auffordern muss:

- zu viele Elemente in der FSR ⇒ erzeuge lokale öffentliche Kopien der von den Task-Inkarnationen aus der FSR benötigten Ressourcen ⇒ der Lesezugriff kann zukünftig lokal durchgeführt werden
- zu viele Elemente in der FSW ⇒ verringere die Zahl öffentlicher Kopien der von den Task-Inkarnationen aus der FSW benötigten Ressourcen ⇒ es müssen weniger Kopien synchron gehalten werden
- zu viele Elemente in der FRR ⇒ erzeuge lokale private Kopien der von den Task-Inkarnationen aus der FRR benötigten Ressourcen ⇒ der Lesezugriff kann zukünftig lokal durchgeführt werden

Für die im Rahmen dieser Arbeit durchgeführte Implementierung von MELODY wurde der File-Server-orientierte Ansatz gewählt. Nachfolgende Kapitel beschreiben diese File-Server-orientierte Integration der MELODY Systemdienste.

1.3.6 File-Server (FS)

Der File-Server ist das zentrale Modul zur Verwaltung des physikalischen Zugriffs auf die Ressourcen in MELODY. Einerseits verwaltet er die konkurrierenden Zugriffe des lokalen und der entfernten Knoten auf die lokalen Ressourcen, andererseits stellt er den lokalen Task-Inkarnationen die zur Ausführung benötigten Ressourcen bereit. Dazu wird der FS gemäß eben dieser zwei Aufgabenbereiche in zwei Komponenten aufgeteilt:

- Der sogenannte *FileServer-Client* (FSC) kommuniziert mit dem lokalen Task-Scheduler um die nächste Task-Inkarnation entgegenzunehmen und dem TS den Erfolg oder Misserfolg der Ausführung mitzuteilen, sowie mit dem lokalen Runtime-Monitor, um gegebenenfalls eine Task-Inkarnation frühzeitig abbrechen zu können. Durch Anfragen an den lokalen und die entfernten FileServer-Server findet er die benötigten Ressourcen, reserviert sie und lässt anschließend die beauftragten Arbeiten an ihnen durchführen. Parallel dazu verwaltet der FSC die *Task-History*, in der einige Eigenschaften, wie die Einhaltung der Deadline, für eine gewisse Anzahl von zurückliegenden Task-Inkarnationen gespeichert sind.
- Der sogenannte *FileServer-Server* (FSS) verwaltet die lokalen Ressourcen des Knotens und regelt den Zugriff des lokalen und der entfernten FSC auf diese. Zusätzlich verwaltet er die physikalischen Zugriffe, indem er sich z.B. um deren Aktualisierung kümmert. Parallel dazu verwaltet der FS-Server die sogenannte *File-History*, in der einige Eigenschaften, wie die Anzahl der Schreib- oder Lesevorgänge, für eine gewisse Anzahl von zurückliegenden Zugriffe auf lokale Ressourcen oder von lokalen Task-Inkarnationen gespeichert sind.

Der Ablauf für eine Task-Inkarnation geht dann folgendermaßen vonstatten: nachdem der Task-Scheduler die nächste ausführbare Task-Inkarnation ausgewählt hat, übergibt er diese dem FileServer-Client. Nun beginnt dieses Modul damit, die von der Inkarnation benötigten Ressourcen in dem verteilten Netz zu lokalisieren und die bestmögliche Einteilung von lokalen und entfernten Zugriffen zu finden. Diese Phase wird in Abbildung 1.2 mit *Resource Location Phase* bezeichnet. Der File-Server als Ganzes ist in der Darstellung von dem Zeitpunkt *Start Time* bis zur *Completion Time* für den Ablauf zuständig.
Ist dann für jede Ressource der Task-Inkarnation eine geeignete Version bzw. Kopie gefunden worden, folgt die *Ressource Allocation Phase*, in der der FS-Client den FileServer-Server, an dem die Ressource liegt (lokal oder entfernt), beauftragt, diese für ihn zu reservieren (*to allocate*). Sobald alle Reservierungen bestätigt sind, wird die von der Task-Inkarnation in Auftrag

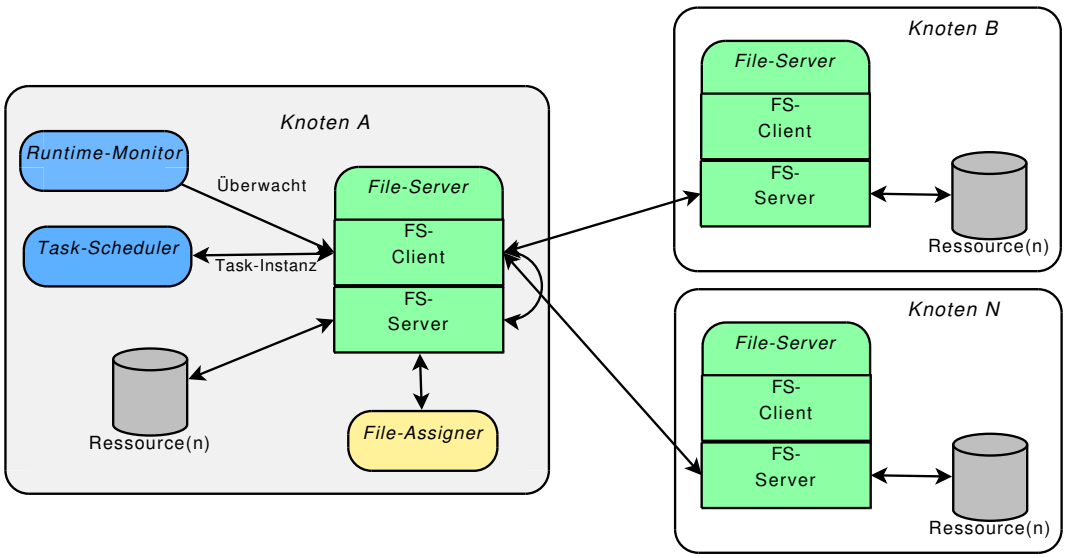

Abbildung 1.7: Das Client/Server-Modell für den File-Server

gegebenen Arbeiten an den Ressourcen dem FSS übergeben (die sogenannten *Remote Procedure Calls* (RPCs)), der diese dann durchführt.
Für den FileServer-Client endet die Task-Inkarnation, nachdem entweder alle entsprechenden FileServer-Server ihr Ergebnis zurückgeliefert haben, oder der FSC zwischendurch festgestellt hat, dass die Deadline für die Task-Inkarnation verpasst wurde. In jedem Fall wird von dem FSC die Task-History aktualisiert und dort je nachdem eingetragen, ob die Inkarnation erfolgreich oder nicht-erfolgreich war, und wie lange die einzelnen Phasen dauerten. Der FileServer-Server aktualisiert dafür seine lokale File-History, je nachdem, ob die Inkarnation abgebrochen/durchgefallen ist oder erfolgreich abgeschlossen wurde, und ob es eine Lese- oder Schreibzugriff war. Im folgenden werden einige Merkmale des File-Servers erläutert.

1.3.7 Task-History

Von dem FileServer-Client werden für eine gewisse Anzahl an zurückliegenden Task-Inkarnationen, die an dem Knoten lokal ausgeführt wurden, in der sogenannten Task-History Daten protokolliert, deren Umfang in der Tabelle 1.1 aufgelistet sind. In Experimenten hat sich zunächst ein Rückgriff auf die Werte der letzten fünf Inkarnationen als ausreichend bewährt. Aus diesen Werten versucht man nun Rückschlüsse auf die nächste Inkarnation der Task zu ziehen. Der relative Kritikalitätsgrad lässt sich dann wie in Definition 1.2 beschrieben berechnen, so wie sich auch die relative Sensitivität gemäß Definition 1.4 ergibt. Aus den aufgezeichneten Laufzeiten lässt sich die geschätzte Ausführungszeit, mit der der Task-Scheduler die Ausführungsreihenfolge der

Wert	Bedeutung
Status	war die Inkarnation erfolgreich oder nicht
Criticality	relative Kritikalität der Inkarnation
Sensitivity	relative Sensitivität der Inkarnation
Location Time	Zeit zum Auffinden aller benötigten Ressourcen
Acquisition Time	Zeit zur Reservierung aller benötigten Ressourcen
Computation Time	Zeit zur Durchführung der eigentlichen Zugriffe (RPCs)

Tabelle 1.1: Task-History-Inhalt

kommenden Task-Inkarnationen planen kann, anhand der folgenden Formel berechnen:

$$\begin{aligned} EET_{jk} &= \text{Estimated\_Location\_Time}(T_j) + \text{Estimated\_Acquisition\_Time}(T_j) \\ &\quad + \sum_{r \in LRF_{jk}} \text{Computation\_Time}(r) \end{aligned}$$

Dabei steht T_{jk} für die Task-Inkarnation, deren Ausführungszeit ermittelt werden soll, und LRF_{jk} (*list of required files*) für die von ihr benötigte Menge an Ressourcen-Kopien. *Estimated_Location_Time* und *Estimated_Acquisition_Time* sind die durchschnittlich benötigten Zeiten für die letzten entsprechenden Phasen von Task T_j, während *Computation_Time(r)* die Zeit für die gewünschte Operation auf der Datei r beschreibt.

1.3.8 File-History

Die von dem FileServer-Server verwaltete File-History protokolliert die erfolgreichen und nicht-erfolgreichen Zugriffe sowohl auf die lokal vorhandenen Ressourcen, als auch auf die von den lokalen Tasks benötigten Ressourcen. Für jede lokal vorhandene *public copy* wird bei jedem Schreibzugriff mitgezählt, ob er erfolgreich war oder nicht. Dabei wird unterschieden, ob diese Operation von einem lokalen Task ausging (der entsprechende Wert für *write-local* wird erhöht) oder von einem entfernten (dann *write-global*). Bei den Lesezugriffen beachtet man nur die lokalen Operationen, differenziert aber, ob die ausführende Task-Inkarnation robust oder sensitiv war. Dies wird nochmal kompakt in Tabelle 1.2 zusammengefasst. Aus der File-History kann nun von dem dafür zuständigen Modul ausgelesen werden, ob und wie der File-Assigner angestoßen werden muss, um möglicherweise eine bessere Verteilung der Ressourcen zu erreichen, so dass weniger Task-Inkarnationen scheitern. Verwendet man die auf Seite 14 beschriebene Task-Scheduler-Integration nutzt der Task-Scheduler die Informationen, wenn man dagegen die File-Server-Integration von Seite 15 einsetzt, greift der File-Server direkt auf die File-History zu. Bei ihrer Implementation ist zu beachten, dass sie auf jeden Fall auch für den File-Assigner direkt zugänglich sein muss, da dieser sie zur Entscheidung nutzt, ob z.B. eine lokale Datei überhaupt gelöscht werden kann, weil sie nicht so häufig genutzt wird.

Operation	Eigenschaft	Ergebnis	Variable	Entsprechung
Schreibzugriff	lokal	erfolgreich	Write-Local-Success	
		gescheitert	Write-Local-Failure	FSW
	entfernt	erfolgreich	Write-Global-Success	
		gescheitert	Write-Global-Failure	FSW
Lesezugriff	robust	erfolgreich	Read-Robust-Success	
		gescheitert	Read-Robust-Failure	FRR
	sensitiv	erfolgreich	Read-Sensitiv-Success	
		gescheitert	Read-Sensitiv-Failure	FSR

Tabelle 1.2: File-History-Inhalt

Wie schon bei der Task-History bietet auch hier der Umfang des beobachteten Zeitraums eine Möglichkeit zur Verbesserung des Systemablaufs. Hier haben Experimente gezeigt, dass der Rückblick auf die letzten fünf Zugriffe einen recht guter Wert für die Größe des Beobachtungsfensters darstellt [LIND 1999, Seite 40].

1.3.9 Delayed Insertion Protocol (DIP)

Das sogenannte *Delayed Insertion Protocol* ist ein Scheduling-Algorithmus, der speziell auf den Umgang mit verteilten Ressourcen ausgerichtet ist. Durch eine große Anzahl von vergleichenden Simulationen (siehe hierzu [DANIELS 1992, WEDDE et al. 1991, WEDDE und DEKKER 1994]) wurde erreicht, dass das die Auswirkungen durch das Problem des *cascaded blocking* verringert werden. Hierbei behindern sich konkurrierende Tasks bei dem exklusiven Zugriff auf gemeinsame Ressourcen, so dass die durchschnittliche Ausführungszeit sehr lang wird. Das DIP kann das Blockieren zwar nicht vollständig verhindern, verringert aber die Häufigkeit seines Auftretens durch das im folgendenden beschriebene Vorgehen.
Der Verwalter einer Ressource (*file-manager*) M_i benutzt Prioritätsschlangen zur Regelung des Zugriffs auf die Ressource. In diesem Fall ist es die *waiting-queue* W_i, die *candidate-queue* C_i und zu guter Letzt die *execution-queue* E_i. Neu ankommende Anfragen landen zunächst in W_i. Erst wenn in E_i keine Anfrage mehr vorhanden sind und diese Schlange somit leer ist, wird der komplette Inhalt von W_i nach C_i kopiert und an jeden der anfragenden Task-Inkarnationen eine Bestätigung geschickt.
Hat die Task-Inkarnation T_{jk} für alle Ressourcen r aus LRF_j eine solche Bestätigung erhalten, kann sie nun beantragen, in die execution-queue E_i aufgenommen zu werden. Dies wird M_i auch durchführen, aber erst wenn die Anfrage von T_{jk} an erster Stelle in E_i angekommen ist, bekommt die Task-Inkarnation die Reservierungbestätigung für die Ressource und somit den exklusiven Zugriff darauf (*lock*).
Zur Vermeidung von Deadlocks gibt es im Delayed Insertion Protocol einen *Call Back Mecha-*

nismus. Sollte eine Task in die Execution-Queue gelangen, die eine höhere Priorität besitzt als diejenige am Anfang von E_i, so wird eine Call-Back-Nachricht an diese verschickt. Sollte sie noch nicht für alle ihre Ressourcen einen "lock" besitzen (die Task-Inkarnation ist also noch nicht ausführend), so wird sie auf die Nachricht mit einem *Acknowledgement* reagieren (sie gibt sozusagen das lock wieder zurück), so dass die Reihenfolge in E_i gemäß der Prioritäten wieder richtig gestellt werden kann. Ist die Task-Inkarnation allerdings schon ausführend, so wird die Call-Back-Nachricht schlicht ignoriert und der lock wird erst am Ende der Ausführung zurückgegeben. Eine ausführliche Beschreibung hierzu findet sich in [LIND 1999, Seite 19ff] und bei [DANIELS 1992].

2 Realzeitfähige Linuxumgebung

2.1 Überblick

Ziel dieser Arbeit ist es, das MELODY-System in ein weit verbreitetes und leicht anzupassendes Betriebssystem zu integrieren und implementieren. Dazu bietet sich der Linux-Kernel an, der als Open-Source unter der GNU General Public License (*GPL*) stehend eine sehr weite Verbreitung besitzt. Für ihn gibt es eine große Auswahl an Entwicklungswerkzeugen und Dokumentation, darüberhinaus wird er seit vielen Jahren beständig weiterentwickelt und auf eine große Anzahl und ein weites Spektrum an unterschiedlicher Hardware angepasst. Allerdings ist der Linux-Kernel standardmäßig nicht realzeitfähig, kann aber durch die im folgenden beschriebende Erweiterung Namens Xenomai für weiche oder sogar harte Realzeitumgebungen einsetzbar gemacht werden.
Das Konzept, das hinter Erweiterungen wie Xenomai, RTAI und RTLinux steckt, ist, dass man die Interrupts virtualisiert und das Linux-System (Kernel- und Nutzerprozesse) zu einem RT-Task mit niedrigster Priorität verwandelt.

2.2 Xenomai

Die eigentliche Idee, die hinter der Entwicklung von Xenomai steckt, ist die, dass man für traditionelle RTOS-APIs (z.B. *VxWorks*, *pSOS+*) ein Linux-basiertes Echtzeit-Framework bietet, so dass bestehende (Industrie-)Anwendungen möglichst einfach in eine Linux-Umgebung migriert werden können, ohne deren Echtzeitfähigkeiten zu verlieren. Zur Realisierung wurde 2001 bei der Gründung des Projekts das grundlegende Konzept von RTAI übernommen, allerdings mit einer weiteren Abstraktionsschicht namens *Xenomai-Nucleus* erweitert. Dieser sogenannte Echtzeit-Nanokernel stellt alle generischen Dienste zur Verfügung, die von den spezialisierten API anderer Echtzeitbetriebssysteme (RTOSs) benötigt werden. Dies ist in Abbildung 2.1 dargestellt. Auf diesen Nucleus greift man nicht direkt zu, sondern eben über einen sogenannten RT-Skin. Zu den traditionellen RTOS-APIs gesellt sich noch eine für das Xenomai-Projekt selber speziell entworfene API namens *Native API*, die vorallem dann genutzt werden soll, wenn eine Anwendung ganz neu entwickelt werden soll. Mit ihr kann man am einfachsten auf den kompletten Funktionsumfang zurückgreifen, den der Nanokernel an Diensten anbietet. Da in der vorliegenden Arbeit nicht auf vorhandenen Quellcode zurückgegriffen wurde, bot sich die Implementation von MELODY in eben jener nativen Xenomai-API an.

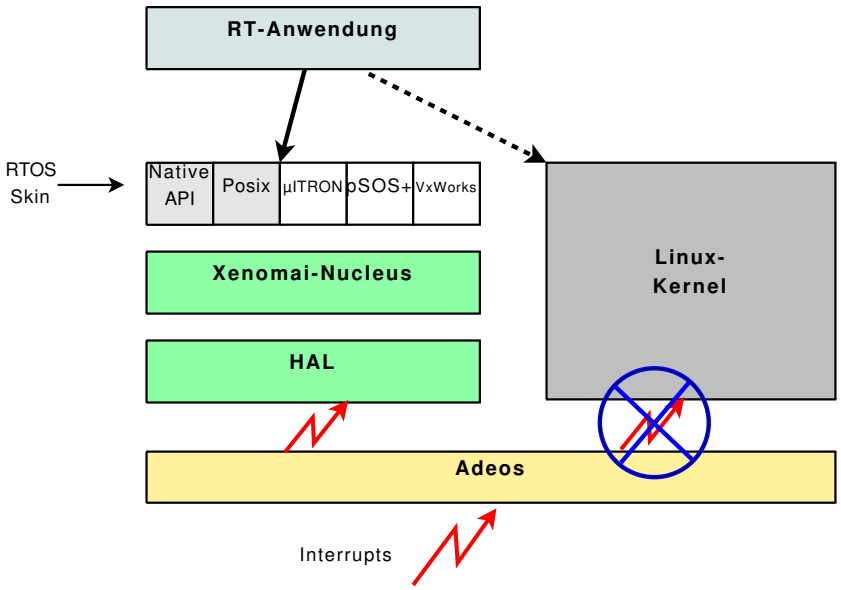

Abbildung 2.1: Der Xenomai-Stack

Die zweite wichtige Idee hinter Xenomai ist eine bessere Unterstützung von Echtzeit-Anwendungen im User-Space. Gerade während der Implementierungs- und Test-Phase für eine neue Anwendung ist es sehr hilfreich, wenn bei einem Fehler nicht das restliche System in Mitleidenschaft gezogen wird, oder man Debugging-Tools verwenden kann. Beides ermöglicht Xenomai, speziell durch die Native API, an, da hier der User-Space als normales Anwendungsziel vorgesehen ist. So ist GDB (oder KGDB für den Kernel-Space) ohne weiteres einsetzbar, und es gibt einen *I-Pipe tracer*, mit dem man Bereiche mit hohen Latenzzeiten auffinden kann (oder die Kernelausführung auf Mikroebene verfolgbar macht).
Von Xenomai werden aktuell sieben System-Architekturen unterstützt (im Einzelnen x86, x86_64, ia64, ppc, ppc64, ARM und blackfin) und die Portierung weiterer ist nur von der Anpassung des ADEOSs abhängig. Auch werden sowohl CPUs mit MMU *Memory Management Unit* wie auch MMU-lose durch die mögliche Verwendung der Kernel-Versionen 2.4 und 2.6 von Linux unterstützt.
All diese Vorzüge, der erfolgversprechende Ausblick, die beständige aktive Weiterentwicklung, wie auch die engagierte Community um das Projekt sind die Ursache, sich für Xenomai als Implementationsziel zu entscheiden. Hierzu kommt noch, dass einer der Hauptentwickler von Xenomai, in der Person von Jan Kiszka (z.Zt. am “Institut für Systems Engineering - Fachgebiet Echtzeitsysteme” Universität Hannover [KISZKA 2007], ab Ende 2007 bei SIEMENS in München), gleichzeitig der derzeitige Hauptentwickler des Projekts *RTnet* ist. Dieses Projekt bietet genau die für MELODY benötigten Eigenschaften einer Echtzeitkommunikation über Ethernet. Da beide Projekte so eng miteinander verbunden sind ist, ein problemloses Zusam-

menarbeiten beider zu erwarten.

2.3 RTnet

Der heutige Ethernetstandard kann Aufgrund der möglichen Datenkollisionen auf dem Netzwerk von Hause aus die Anforderungen der Echtzeitfähigkeit nicht erfüllen. Deshalb wurde von der Universität Hannover das *RTnet*-Projekt ins Leben gerufen. Das Projekt ist eine Open-Source-Lösung unter der GNU General Public License.
Primäre Aufgabe von RTnet ist es, die vollständige Kontrolle über den Zeitpunkt von Sendevorgängen zu ermöglichen und eine maximale Übertragungsdauer zu gewährleisten. Dabei wird ausschließlich auf kostengünstig verfügbare Standard-Hardware zurückgegriffen und die Steuerung des Medienzugriffs in Software realisiert. Ziel des Projekts ist es darüber hinaus, an RTnet angeschlossene Teilnehmer per TCP/IP erreichbar zu erhalten, um zeitunkritische Aufgaben, wie z.B. zur Fernwartung, weiterhin über das Internet durchführen zu lassen.

2.3.1 Systemaufbau

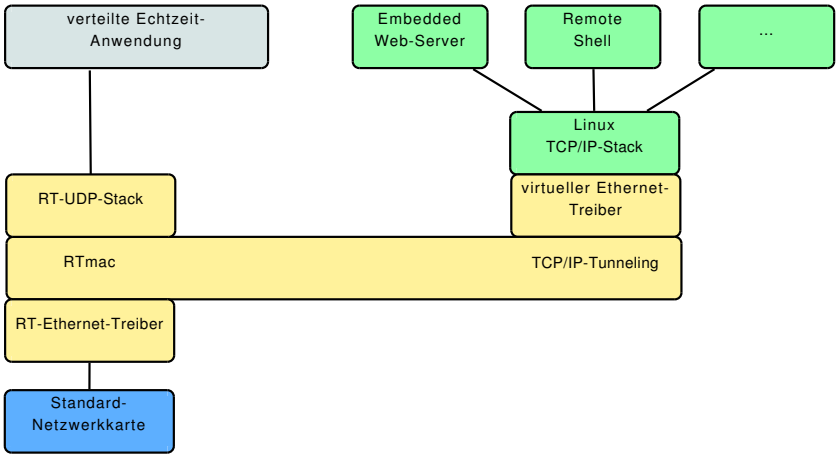

Abbildung 2.2: Aufbau von RTnet

RTnet implementiert einen echtzeitfähigen IP-Protokoll-Stack. Dazu wurde der sehr robuste TCP/IP-Stack des Linux-Kernels zunächst auf RTAI, später auf Xenomai portiert. Unterstützt wird allerdings nur das UDP-Protokoll, da TCP durch die automatische Paketwiederholung im Fehlerfall nicht mehr deterministisch arbeitet. Der Aufbau des RTnet-Stacks ist in Abbildung 2.2 dargestellt. Neben dem UDP-Protokoll besteht RTnet aus der modularen Medien-Zugriffsschicht

RTmac (*RealTime media access control*), einem Treiber für einen virtuellen Netzadapter zur Anbindung von Standard-Linux sowie echtzeitfähigen Treibern der physikalischen Netzadapter. Die letztgenannten Treiber wurden ebenfalls aus dem Linux-Kernel entnommen und durch wenige Änderungen an die Anforderungen von RTnet angepasst, so dass es aktuell bereits eine große Basis an unterstützen Chipsätzen gibt, wie z.B. RTL8139 und RTL8169 der Firma RealTek oder PRO/100 (82559) und PRO/1000 von Intel. Die Anwendungsprogrammierung erfolgt über die bekannte BSD-Socket-Schnittstelle, für die es umfangreiche Dokumentation gibt [STEVENS et al. 1998].
Der Medienzugriff wird über das TDMA-Verfahren (*Time Division Multiple Access*) geregelt. Dazu weist ein Master-Knoten den übrigen Knoten im Netz, den sogenannten Slave-Knoten, Zeitschlitze zu, zu denen sie exklusiv das Medium zur Kommunikation nutzen dürfen.

3 Design

Der in dieser Arbeit gewählte Ansatz orientiert sich, im Gegensatz zu vorangegangenen Implementierungen, möglichst nah an den von Jon Arthur Lind in [LIND 1999] vorgestellten Design-Ideen. Eine über das Folgende herausgehende Beschreibung des MELODY-Systems findet sich bei [BENCHELIH 2007] und [KEMPER 2007].

Das MELODY-System besteht, wie in Abschnitt 1 beschrieben, aus vier Bestandteilen:

- einem Task-Scheduler (TS)
- einem File-Server (FS)
- dem File-Assigner (FA)
- dem Runtime-Monitor (RTM)

Hierbei sind sowohl der File-Server als auch der File-Assigner in einen Server- als auch einen Client-Teil aufgeteilt. Die einzelnen Module werden nachfolgend auch als Services bezeichnet. Die übrigen Teile von MELODY, vor allem bestehend aus Task-Scheduler und Runtime-Monitor, wurden in dieser Arbeit auf ihre notwendige Funktionalität reduziert, implementiert. Statt zweier getrennter Module gibt es ein einziges kombiniertes Modul, welches bei einer späteren Vervollständigung des MELODY-Systems jederzeit wieder aufgetrennt werden kann. Die Reduzierung äußert sich darin, dass bereits vor dem Start des Systems eine statische Reihenfolge der abzuarbeitenden Tasks vorgegeben und nicht erst vom Task-Scheduler dynamisch erzeugt wird. Ansonsten kann das kombinierte Runtime-Monitor-/Task-Scheduler-Modul Werte aus der Task-History vom File-Server erfragen und gegebenfalls, wenn der Aufruf so früh erfolgt, dass noch keine Ressourcen reserviert sind, eine Task-Inkarnation auch abbrechen.
Dafür ist noch der sogenannte *Debug-Service* hinzugekommen, der allerdings keine direkte Arbeit im MELODY-System verrichtet, sondern allein zum Sammeln von Statusmeldungen und zur äußeren Kontrolle des Systems zur Laufzeit dient.

Wie bereits mehrfach beschrieben, ist in MELODY-System die Reihenfolge von Ressourcen- und Task-Scheduling vertauscht, vergleicht man es mit dem klassischen Entwurf von Betriebssystemen. Die Aufgabe des File-Servers ist es also, für eine vorgegebenen Task-Inkarnation die notwendigen Ressourcen zunächst im verteilten Netz zu finden, zu reservieren und schließlich die gewünschte Operation durchzuführen. Da es sich zudem um eine sicherheitskritische Umgebung handelt, muss darauf geachtet werden, dass die Deadline der Task nicht überschritten wird. Vereinfacht sieht der Ablauf so aus, dass zunächst eine Liste der Ressourcen und deren Operation auf ihnen an den File-Server vom Task-Scheduler übermittelt wird. Nun muss der

FS entscheiden, ob etwaige lokale Versionen der Task genügen, oder er muss sich gegebenenfalls mit entfernten File-Servern über den Zugriff auf entfernte Kopien einigen. Nach Ablauf des Zugriffs wird dieser ausgewertet, und bei einer Häufung von Fehlerfällen über das Modul File-Assigner eine Änderung der Ressourcenverteilung ausgelöst.

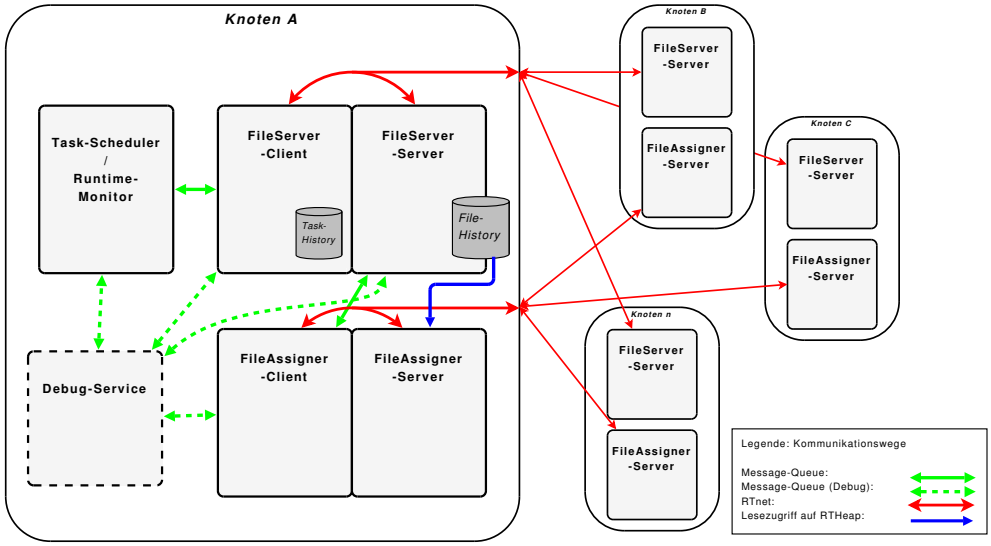

Abbildung 3.1: Designkonzeption für MELODY

3.1 Laufzeitumgebung und API

Da diese Arbeit auf Xenomai aufbaut, stellt sich die Frage nach der Wahl einer geeigneten API. Will man eine neue Anwendung erstellen, die nicht auf vorhandenem Sourcecode aufbaut, bietet Xenomai eine um Realzeitfähigkeiten erweiterte *POSIX API*, sowie eine so genannte *Native API*, die mit Xenomai entwickelt wird. Da die *Native API* mit einer sehr guten Online- [KISZKA et al. 2007] wie Offline-Hilfe [KISZKA et al. 2006] ausgestattet ist, wird letztere bevorzugt. Um eine möglichst stabile und Entwickler-freundliche Realzeit-Umgebung zu bieten, wird von den Xenomai-Entwicklern versucht, bei der *Native API* folgende Regeln einzuhalten:

Kompakte API - die nicht mehr als hundert Services zur Verfügung stellt.

Klarheit - d.h. für jeden Anwendungsfall gibt es einen eindeutigen Service, den man dazu aufrufen muss.

Kontext-Unabhängigkeit - egal ob für den Kernel- oder User-Space programmiert wird, die API bleibt gleich.

Nahtlose Topologie - mithilfe einer so genannten *Registry* lässt sich aus allen Umgebungen auf alle Objekte per eindeutigem Deskriptor auf immer die selbe Art und Weise zugreifen.

Wie auch bei der Laufzeitumgebung bietet sich die Möglichkeit, später, z.B. bei einer Erweiterung des MELODY-Systems, zusätzlich auch auf die POSIX API zuzugreifen.

3.1.1 Netzkommunikation

Ein weiterer Grund für die Entscheidung zu Gunsten von Xenomai als Realzeit-Erweiterung für Linux ist die sehr gute Integration von RTnet in Xenomai. Mithilfe von standardisierten PC-Komponenten kann man über ganz gewöhnliche (exklusive) Ethernet-Netze Realtime-Kommunikation zwischen verteilten Knoten herstellen, indem man Nachrichten per UDP/IP austauscht. Unterstützt werde einige populäre Netzchips, wie z.B. die RealTek RTL8139 und RTL8169 (Gigabit Ethernet), oder die Intel 8255x EtherExpress Pro100 und PRO/1000 (Gigabit Ethernet). Die Beschränkung auf UDP-Nachrichtenpakete stellt für MELODY kein Problem dar, da aufgrund der Abgeschlossenheit des verwendeten Netzsegmentes (es werden nur RTnet-Rechner verbunden) und der Verwendung des TDMA-Verfahrens nur sehr wenige Nachrichten verloren gehen. In eigenen Tests mit fünf Knoten konnte nie ausreichend Last im Netz erzeugen werden, als das auch nur ein Paket verloren gegangen wäre.

4 Implementierung

4.1 Programmablauf

Der von uns gewählte Entwurfsansatz lässt sich am besten anhand der von uns in drei Phasen aufgeteilten Laufzeit des MELODY-Systems erklären:

Initialisierung Zunächst werden von jedem Service die entsprechenden Konfigurations-Dateien mit den, bei jedem Start einstellbaren, Optionen geladen. Im zweiten Schritt werden alle Kommunikationswege initialisiert, die ein Service benutzen muss. Nachdem dann die Zeit auf den einzelnen Knoten der Zeit eines (vorbestimmten) Master-Knotens angeglichen wurde, wird zum Schluss die Hauptphase durch das Versenden eines Startpakets gestartet.

Hauptphase Dies ist die eigentliche Laufzeit des MELODY-Systems, in der die Task-Inkarnationen auf die Ressourcen lesend und schreibend zugreifen. Zur späteren Auswertung werden, in Abhängigkeit des eingestellten *Debug-Level*, für alle Ereignisse und Änderungen Log-Einträge an den so genannten Debug-Service übermittelt.

Shutdown Ist die vorgegebene Laufzeit abgelaufen, werden die einzelnen Services *heruntergefahren*. Dazu werden die veränderten Ressourcen wieder auf die Festplatte gespeichert, die aufgezeichneten Logs des *Debug-Services* werden ebenfalls gesichert.

Im Nachfolgenden werden die drei einzelnen Phasen genauer beschrieben.

4.2 Initialisierung

Die einzelnen Services sind jeweils in einen eigenen Realtime-Task ausgegliedert, der im User-Space ausgeführt wird. Diese werden aus der *main*-Funktion heraus aufgerufen. Um überhaupt echtzeitfähig zu sein, muss zunächst der gesamte benötigte Speicher alloziert werden, damit dies nicht mehr dynamisch zur Laufzeit vom Linuxkernel gemacht werden muss.
Durch `rt_task_create(RT_TASK *task,*name,stksize,prio,mode)` wird ein RT-Task erzeugt, aber erstmal suspendiert, bis er mit dem Aufruf von `rt_task_start(RT_TASK *task, ...)` zum ersten Mal gescheduled wird. Wie man sieht, gibt man an dieser Stelle die Priorität des Tasks an, und man kann dabei zwischen 0 als niedrigster und 99 als höchster Priorität wählen. In Experimenten hat es sich bewährt, den Task-Scheduler mit der niedrigsten aller drei Prioritäten auszustatten, da er die meiste Zeit auf eine Antwort des FileServer-Clients

wartet. Der FileServer-Client bekommt die mittlere und der FileServer-Server die höchste Priorität, da er oftmals auf eine hohe Anzahl von Anfragen von entfernten Knoten antworten muss. Der eigentliche Wert für die Prioritäten ist dabei natürlich unerheblich, da der Schedule allein von der Verhältnis zueinander abhängig ist.
Nach dem Start der Tasks befinden sich diese nun zunächst in der Startphase. In dieser Phase werden zunächst alle diejenigen Arbeiten durchgeführt, für die auf Linux-Kernel-Services zurückgegriffen werden muss. Xenomai kennt für den User-Space zwei verschiedene Ausführungsmodi, den so genannten *primary execution mode*, in dem der Xenomai-Task echtzeitfähig läuft, und den *secondary execution mode*, in der der Task in der so genannten *Linux-Domain* läuft, in der Linux-Systemaufrufe benutzt werden können, die Echtzeitfähigkeit aber verloren geht.
Die Migration zwischen diesen beiden Modi ist möglich, kostet aber jedes mal Zeit. Deshalb werden in dieser Phase alle benötigten Systemaufrufe zusammengefasst, sodass in der Hauptphase die Tasks alle möglichst konstant in der *Xenomai-Domain* verbleiben können. Aus diesem Grund wird auch ein Debug-Service verwendet, der die Debug-Ausgabe aufnimmt bzw. verwaltet. Dazu wird für jeden Service eine Message-Queue zum Debug-Service benutzt, die dieser reihum abfragt. Da diese für jeden Service gleich ist, wird nachfolgend nicht mehr explizit hierauf eingegangen.
Der erste Service, der gestartet wird, ist der Task-Scheduler, welcher eine Kombination aus dem eigentlichen Task-Scheduler-Modul und dem Runtime-Monitor ist, und der Einfachheit halber nachfolgend nur noch als Task-Scheduler bezeichnet wird. Als erstes wird direkt nach dem Start des Systems die zugehörige Konfigurationsdatei namens *ts.conf* geöffnet. In dieser steht, ob der aktuelle Knoten der sogenannte *Master* ist, oder nicht. Im gesamten Xenomai-System gibt es nur einen Master-Knoten, und der ist für den gemeinsamen Wechsel aller Knoten in die Hauptphase und aus ihr wieder heraus verantwortlich. Wer der Master ist, wird noch vor dem Start des MELODY-Systems festgelegt durch einfaches editieren der Konfigurationsoption `Master` auf den Wert `1`. Wie dieser Wechsel abläuft wird im Detail im nächsten Abschnitt beschrieben.
Ansonsten ist in `ts.conf` noch die Liste der abzuarbeitenden Tasks vermerkt, zusammen jeweils mit der Deadline, ob lesend oder schreibend operiert wird, welche Ressourcen benötigt werden, die initiale Kritikalität und Sensitivität, und wie oft die Liste der Tasks durchlaufen werden soll.
Nach dem Auslesen der Konfigurationsdatei werden noch zwei Message-Queues angelegt, um darüber mit dem lokalen FileServer-Client zu kommunizieren. Man kann zwar eine Message-Queue in beide Richtungen benutzen (bzw. können auch mehr als zwei Tasks eine Message-Queue gemeinsam nutzen), allerdings ist nur bei exklusiver Nutzung in jeweils einer Richtung gewährleistet, dass keine Nachrichten unbeabsichtigt von der "falschen" Seite wieder aufgenommen werden.
Neben einem Socket zur Kommunikation mit den entfernten Task-Schedulern zum Starten und Stoppen der Hauptphase liest der TS dann nur noch den Offset zur netzweiten-Zeit aus, wie im folgenden beschrieben, ist ansonsten aber bereit für die Hauptphase.
Von allen Services, die zur korrekten Funktion eine gemeinsame Zeitbasis benötigen, wie z.B. der FileServer-Client für Anfragen an entfernten Knoten, wird zur Zeitsynchronität zwischen

allen Knoten des Netzes die Zeit aus der TDMA-Schicht des RTnet-Stacks gelesen. Um die Echtzeitfähigkeit über Ethernet zu erhalten, wird bei RTnet ein TDMA-Protokoll verwendet, das von einem Master gesteuert wird. Dieser Rechner initiiert das Netz, indem er beim Start auf alle in einer Konfigurations-Datei aufgeführten *Slaves* wartet.
In der Konfigurations-Datei bei den Slaves ist wiederum jeweils deren eigene IP-Adresse aufgeführt, so dass jeder Slave weiß, wie er auf eine Start-Nachricht reagieren soll. Bei dem Start von RTnet schickt der Master mittels Broadcast diese Start-Nachricht in das, an die ausgewählte Netzwerkkarte angeschlossene Netz, und wartet solange, bis sich zu jeder in der Konfigurationsdatei vermerkten IP-Adresse ein Teilnehmer (bzw. eben Slave) gemeldet hat. Nach einer kurzen *Einschwingzeit* kann man dann Pakete realzeitfähig über das so aufgebaute Netz versenden. Da man für diese TDMA-Schicht schon eine gemeinsame Zeitbasis hat, kann man diese einfach abfragen und sich als Bezugspunkt zur eigenen Systemzeit merken, ohne wirklich die Systemzeit langwierig anpassen zu müssen.
Dazu stellt RTnet das Device `TDMA0` zur Verfügung, welches man mit dem Befehl `rt_dev_open` öffnen, und dann mit `rt_dev_ioctl(fd,RTMAC_RTIOC_WAITONCYCLE_EX,&waitinfo)` Informationen über die Zeiten des eigenen TDMA-Slots auslesen kann. Der Zeitabstand zwischen dem Master und dem auslesenden Slave ist dann der Wert `waitinfo.clock_offset` vom Typ `int64`, der die Zeitdifferenz in Nanosekunden liefert. Diesen Wert kann man sich nun abspeichern, da während der überschaubaren Laufzeit des MELODY-Systems die Zeiten der Rechner nicht weiter auseinander laufen sollten. Wann immer man nun die vom Master vorgegebene *netzweit gültige Zeit* wissen möchte, z.B. um zu testen, ob eine Deadline schon abgelaufen ist, berechnet man einfach die Summe aus eigener Systemzeit (realzeitfähig auslesbar durch die Xenomai-Funktion `rt_timer_read()`, ebenfalls in Nanosekunden) und dem gespeicherten Offset.
Das Auslesen von `TDMA0` stellt in der Startphase in sofern eine Besonderheit dar, als dass diese nur im *primary mode* möglich ist.
Für solche Fälle stellt die Native API den Befehl `rt_task_set_mode()` zur Verfügung:

- `rt_task_set_mode(0, T_PRIMARY, NULL)` wechselt Zwangsweise in den *primary mode*
- `rt_task_set_mode(0, T_PRIMARY | T_WARNSW, NULL)` wechselt Zwangsweise in den *primary mode* mit Signal für `SIGXCPU` (zum Debuggen)
- `rt_task_set_mode(T_PRIMARY, 0, NULL)` wechselt Zwangsweise in den *secondary mode*
- `rt_task_set_mode(T_PRIMARY | T_WARNSW, 0, NULL)` wechselt Zwangsweise in den *secondary mode* mit Signal für `SIGXCPU` (zum Debuggen)

Der FileServer-Client (**FSC**) initialisiert bzw. bindet sich im ersten Schritt an seine benötigten Kommunikationswege. Da sind zum einen die Message-Queues vom Task-Scheduler, über den die Liste der bereitzustellenden Ressourcen für den jeweils aktuellen Task kommt, als auch die

Anfragen nach den aktuellen Werten für *Criticality*, *Sensitivity* und *Estimated Execution Time* eines Tasks, die der **FSC** in der *Task-History* aus vorangegangenen Task-Inkarnationen berechnet und verwaltet. Über die andere Message-Queue wird dann die Antwort an den Task-Scheduler zurückgeliefert. Ein weiterer Kommunikationsweg besteht aus der Verbindung zu dem lokalen wie auch den entfernten FileServer-Servern (**FSS**) über das RTnet. Dazu wird in dieser Phase ein Socket für eingehende Antworten der FSS an einen Port gebunden.

Aus der Konfigurationsdatei *fsc.conf* ermittelt der FSC im Übrigen dann nur noch, wieweit zurück in die Vergangenheit gegriffen werden soll, um die Werte für die Task-History zu berechnen. Diese Information liefert der Wert *MAX_OBSERVED_TI*, der festlegt, nach wievielen Task-Inkarnationen der erste Wert wieder überschrieben wird. Daneben stehen in *fsc.conf* die Thresholds *MIN_CRITI*$\equiv a'_i$ und *MAX_CRITI*$\equiv a''_i$ für die Berechnung des relativen Kritikalitätsgrads und entsprechend *MIN_SENSI*$\equiv b'_i$ und *MAX_SENSI*$\equiv b''_i$ für die relative Sensivitität (siehe Abschnitt 1.2. Zu guter Letzt findet sich auch noch die Zeit, gemessen in Nanosekunden, wie lange auf eine einzelne Nachricht in RTnet gewartet werden soll.
Der FileServer-Server dagegen liest aus der Konfigurations-Datei zunächst einmal, welche Ressourcen im Startzustand überhaupt auf dem Knoten vorhanden sind, und lädt diese dann in dieser Phase von der Festplatte in seinen Speicherheap. Da der Festplattenzugriff ein Systemcall ist, wird erst wieder nach Abschluss der Hauptphase wieder auf die Festplatte zugegriffen, und so bleiben Änderungen an den Ressourcen zunächst immer nur auf den Speicherheap beschränkt, bis in der Shutdown-Phase der Inhalt des Heaps auf der Festplatte gesichert wird. Für jede Ressourcenklasse *private*, *public* und *shadow* wird ein eigener Heap in ausreichender Größe alloziert, dass jederzeit alle Ressourcen aufgenommen werden könnten.
Ein weiterer Heap wird für die so genannte *File-History* benutzt. Die Verwendung des Heaps ermöglicht es dem lokalen File-Assigner, lesend direkt auf die File-History zuzugreifen, ohne den File-Server fragen zu müssen. Diese Anforderung sind direkt der Arbeit von John Lind [LIND 1999] entnommen. Erstellt wird ein solcher RT-Speicherheap mit dem Befehl `rt_heap_create(RT_HEAP *heap,*name,heapsize,mode)` und es kann aus diesem Bereich dynamisch mit der Funktion `rt_heap_alloc(RT_HEAP *heap,size,timeout, void **blockp)` Speicher alloziert werden. Das besondere an dem RT-Heap von Xenomai ist, dass man über Namen als Symbol systemweit per `rt_heap_bind(RT_HEAP *heap, *name, timeout)` von anderen Tasks auf den Speicherbereich zugreifen kann.
Neben den erwähnten einzulesenden Dateien ermittelt eine Reihe von Thresholds, die für die Bearbeitung der File-History notwendig sind. So liest auch der FSS, wieweit zurück auf die Vergangenheit gegriffen werden soll, in diesem Fall, um die Werte für die File-History zu berechnen *FHIST_CONTENT*. Dazu kommen noch die Thresholds, die festlegen, wann der File-Assigner aufgerufen werden muss, um die Verteilung der Ressourcen zu verbessern: *FRRTh*, *FSRTh* und *FSWTh*.
An Kommunikationswegen benutzt der FileServer-Server immer RTnet für den Nachrichtenaustausch mit anderen FileServer-Teilen, ob nun lokale oder entfernte FSC oder entfernte FSS.

So gibt es nur diese eine Stelle, an der der FSS Aufforderungen erhalten kann, was die Implementierung der FSS erleichtert, da nur immer diese Stelle regelmäßig auf Nachrichten überprüft werden muss.
Ansonsten kommuniziert der FSS noch mit dem lokalen FileAssigner-Client über zwei Message-Queues, um unter anderem die Aufgabe der *FileServer-Oriented Integration* zu erfüllen. Im Gegensatz zur *Task-Scheduler-Oriented Integration* ist hier der FSS dafür verantwortlich, anhand der File-History und aus eingelesenden Thresholds zu erkennen, ob, nach verpassen einer Deadline, die Anzahl eines bestimmten Public-Copy im Netz verringert, eine lokale Public-Copy angelegt, oder eine lokale Private-Copy angelegt werden muss. Für die Implementierung wurde dieses Modell gewählt, da zum einen der FSS sowieso schon alle erforderlichen Kennzahlen kennt, zum anderen der Task-Scheduler in bisherigen Arbeiten nicht vollständig implementiert wurde, im Gegensatz zum File-Server. Eine wichtige Datenstruktur für den FSS ist die des sogenannten *File-Managers*, mit deren Hilfe die Ressourcen und die Zugriffe auf diese verwaltet werden. Die Struktur hat den Namen `ResourceList_T` und enthält folgende Elemente:

- der Name der verwalteten Ressource zur Identifikation,
- die Größe der Ressource im Heap (zum späteren Speichern auf der Festplatte)
- Identifikation der Ressource (zum Wiederfinden in der File-History)
- die Art der Kopie, wobei eine Unterscheidung zwischen *public* und *private* genügt, da es zu jeder *public*- immer eine *shadow-copy* gibt,
- der aktuelle Status der Ressource: *active* (zugreifbar), *inactive* (hier nicht vorhanden), *partial* (wird geschrieben, ist schon zugreifbar) oder *deleted* (wird hier entfernt, wenn alle derzeitigen Zugriffe beendet sind),
- der Zeiger auf die public-copy im RT-Heap,
- der Zeiger auf die shadow-copy im RT-Heap,
- der Zeiger auf die private-copy im RT-Heap,
- der Zeiger auf das erste Element in der candidate-queue (für die public-copy), welche durch eine verkettete Liste realisiert ist, wobei jedes Element eine Anfrage an die Ressource darstellt:
  - die IP des anfragenden Knoten
  - die Nummer des Tasks
  - die Kritikalität der Task-Inkarnation (zur Sortierung der Anforderungen)
  - die Sensitivität der Task-Inkarnation (zur Sortierung der Anforderungen)
  - die Deadline des Tasks (zur Sortierung der Anforderungen)

  - eine Identifikationsnummer der Anfrage-Nachricht (damit beim FSC nichts durcheinander kommen kann, wenn dieser inzwischen eine Task-Inkarnation weiter ist)
  - ein Zeiger auf das nächste Element der Liste
- der Zeiger auf das erste Element in der waiting-queue (für die public-copy)
- der Zeiger auf das erste Element in der execution-queue (für die public-copy)
- der Zeiger auf das Element, dass den Schreibzugriff zur Zeit hat (für die public-copy)
- der Zeiger auf das erste Element in der access-queue (für die shadow- und private-copy)

Auf die Listen wird dann mit den beiden Funktionen `queue()` und `dequeue()` zugegriffen, und entsprechend entweder das übergebene Element richtig sortiert in die Liste eingefügt, oder es wird durch Vergleich der übergebenen Task-ID und Auftrags-IP-Adresse mit den Elementen der Liste ein Element gesucht und bei Erfolg entfernt.

4.3 Hauptphase

Der Beginn und das Ende der Hauptphase wird von dem Task-Scheduler ausgelöst. Das Problem, dass sich stellt, ist, dass alle Knoten im Netzwerk zur gleichen Zeit mit ihrer Hauptphase beginnen sollen, um das MELODY-System in einer konsistenten Umgebung ablaufen lassen zu können. Bis dahin konnten die einzelnen Knoten noch voneinander unabhängig agieren, doch nun kommt es zur Kommunikation untereinander. Das MELODY-System muss auf jedem Knoten einzeln gestartet werden, die Hauptphase wird jedoch zentral eingeleitet.
Auslöser für den Wechsel zwischen der Initialisierungsphase und der Hauptphase ist die Nachricht des Master-Knoten an die Slaves.

In der vorliegenden Implementierung wird die zentrale Steuerung dadurch erreicht, dass die Task-Scheduler die eigentlichen Auslöser für die nächsten Arbeitsschritte sind. Wenn der TS keine Aufträge weitergibt, warten der FileServer-Client und damit nachfolgend der FileServer-Server immer weiter, bis schließlich eine Task-Inkarnation oder eine Anfrage über File-History-Werte angetragen werden. Diesen Umstand macht man sich zu Nutze: man lässt alle Task-Scheduler, bis auf den Master, auf eine Broadcast-Nachricht von eben diesem warten. So wird erreicht, dass quasi Zeitgleich (sieht man einmal von den unterschiedlichen Laufzeiten des RTnet-Pakets ab) alle Knoten mit der Abarbeitung ihrer Tasks beginnen. Nach dem Ablauf der Initialisierung warten alle Module auf die erste Nachricht, die abgearbeitet werden kann. So wartet am Start dieser Phase und immer nach Ende einer Task-Inkarnation der FileServer-Client auf die nächste Nachricht in der Message-Queue, die vom Task-Scheduler mit Nachrichten befüllt wird. Die native Xenomai-API bietet genau für einen solchen Fall den Befehl `rt_queue_read(RT_QUEUE *q, *buf, size, timeout)`, der, wenn man `timeout` statt mit einer Zeitangabe mit der Definition *TM_INFINITE* belegt, den aufrufenden Task genau

solange blockieren lässt, bis eine neue Nachricht in der Message-Queue vorliegt.
Bei der ausgelesenen Nachricht wird als erstes über den vom Task-Scheduler mitgeschickten *Message-Typ* entschieden, was der FSC als nächstes zu tun hat. Im Grunde gibt es drei Gruppen von Anfragen, die der TS an den FSC stellen kann:

1. ***MSG_TASK_COPY_LIST***: die eigentliche Hauptaufgabe des FileServer-Clients wird mit diesem Nachrichtentyp angestoßen, und zwar das Verarbeiten einen Task-Inkarnation bzw. das Reservieren der verlangten Ressourcen und der eigentlichen anschließende Beauftragung der Operation auf diesen (der eigentliche physikalische Zugriff wird natürlich vom FileServer-Server durchgeführt). Die Nachricht, definiert durch die `MessageQ_T` enthält folgende Informationen:
   - die wievielte Nachricht dies ist, damit die Nachrichten untereinander nicht verwechselt werden können,
   - eine Liste der Ressourcen-Kennungen,
   - die gewünschte Operation auf jeder Ressource,
   - der Zeitpunkt, bis wann die Task-Inkarnation abgearbeitet sein muss (Deadline in netzweit gültiger Zeit),
   - die ID des Task, aus der die Task-Inkarnation erzeugt wurde (u.a. zum Finden der zugehörigen Werte in der Task-History) und
   - die ID der Task-Inkarnation selber
2. ***MSG_INQ_XXX***: hierüber kann der TS die Werte Sensitivität, Kritikalität und geschätzte Ausführungszeit (*EET*) aus der Task-History vom FileServer-Client ausrechnen lassen. Der FSC liefert die Werte dann umgehend in einer Nachricht vom entsprechenden Typ *MSG_INQ_REPLY_XXX* an den Task-Scheduler zurück.
3. ***MSG_STOP_WORKING***: diese Nachricht löst beim FSC aus, dass er eine Nachricht vom Typ *MSG_FSS_STOP* an seinen lokalen FileServer-Server über RTnet schickt, dann als Bestätigung an den TS mit *MSG_STOPPED_WORKING* zurückschickt, und danach die Hauptphase beendet.

Der Ablauf zwischen den Modulen wird in Abbildung 4.1 verdeutlicht.

Zunächst einige grundlegende Dinge zur Kommunikation zwischen FSC und FSS über RTnet:

- Die API für RTnet ist ganz ähnlich zu der für BSD-Socket-Programmierung, in der Regel muss vor jeden Funktionsaufruf nur *rt_* oder *rt_dev* vorangestellt werden.
- Es gibt zwei Arten von Zieladressen: entweder ein BROADCAST an alle Knoten im Netz (z.B. 10.255.255.255), oder eine einzige bestimmte Zieladresse, wie LOCALHOST (127.0.0.1) oder die Adresse wurde aus einer Antwort ermittelt (`rt_dev_recvfrom()`

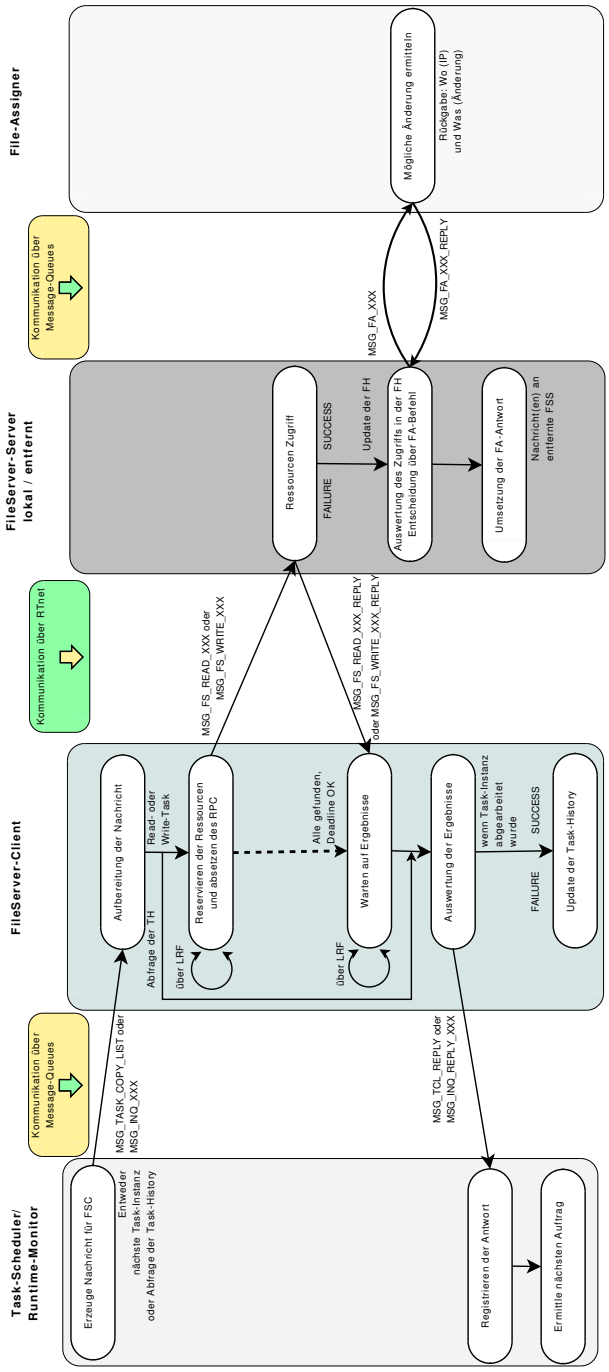

Abbildung 4.1: Ablauf zwischen allen Modulen in der Hauptphase

speichert an fünfter Stelle der Parameter die Ursprungsadresse der Nachricht). Es gibt in RTnet (noch) keinen MULTICAST, also das gleichzeitige Senden an einen Teilbereich des Netzes.

- Da nur UDP-Pakete versendet werden und somit die Kommunikation *verbindungslos* abläuft, wird nicht `rt_de_recv` sondern immer `rt_de_recvfrom` aufgerufen, da man auf diese Weise, wie erwähnt, die Absendeadresse mitgeliefert bekommt.
- Ziel ist bei *jeder* Nachricht vom FSC der Port vom FileServer-Server *FSS_PORT*, genauso wie umgekehrt die Nachrichten des FSS an den *FSC_PORT* adressiert sind (mit Ausnahme der Befehle an entfernte FSS, die sich aus dem Aufruf des File-Assigners ergeben haben, wie das Löschen einer Kopie).
- Wie bei der Message-Queue kann man die Zeit einstellen, wie lange auf eine Nachricht gewartet werden soll. Dazu ruft man vor dem eigentlichen `rt_dev_recvfrom` die Funktion *rt_dev_ioctl(fscsock, RTNET_RTIOC_TIMEOUT, &timeout)* auf, wobei der Parameter *timeout* die Wartezeit an Socket *fscsock* bestimmt. Fehlt dieser Befehl, dann wird der aufrufende Task solange blockiert, bis eine Nachricht am Socket vorhanden ist. Dieses Feature wird bei dem FileServer-Server verwendet, so dass der FSS-Task ununterbrochen auf Nachrichten warten kann, ohne dass man ihn über andere Wege aufwecken muss.

Das Verarbeiten einer Task-Inkarnation kann wiederum in zwei Fälle unterteilt werden. Der etwas einfachere Teil ist der *Read-Task*, also ein Lesezugriff auf die Ressourcen. Das Zusammenspiel zwischen dem FileServer-Client und dem/den FilerServer-Servern ist schematisch in Abbildung 4.2 dargestellt, und lässt sich, wie in der Abbildung angedeutet, in vier Abschnitte einteilen:

1. **Location**: Für jede Ressource wird ein Ort gesucht, an dem man lesend auf diese zugreifen kann. Bevorzugt wählt man in dieser Reihenfolge die Kopien aus:
   - eine *lokale* Schattenkopie (shadow copy), da man auf diese am schnellsten Zugreifen kann. Dazu wird als erstes eine Nachricht an den Localhost geschickt. Erhält der FSC keine Antwort in der, in der Konfigurations-Datei festgelegten, Zeit (weil der FSS bei nicht-vorhandensein der Kopie keine Antwort schickt),
   - eine (lokale) Privatkopie (private copy), wofür der aufrufende Task allerdings entweder das Attribut *robust* oder *hochkritisch* (essential critical) haben muss (ermittelt aus der Task-History), ansonsten
   - eine *entfernte* Schattenkopie, wobei man nach der BROADCAST-Nachricht einfach die Adresse nimmt, von der die erste Meldung wieder eintrifft.

   Die Adressen sind dann in einem Array über alle Ressourcen namens *location* hinterlegt.
2. **Acquisition**: Hat man für jede Ressource eine Kopie gefunden, wird an jede Adressen aus dem erwähnten Array die Aufforderung verschickt, in die Access-Queue für die Ressource

gestellt zu werden. Wird dies gemacht hat der FSC den Lesezugriff auf diese Kopie.

3. **Computation**: Da zu Beginn dieses Abschnitts sichergestellt ist, dass für alle Ressourcen aus der LRF_{jk} ein Lesezugriff vorliegt, wird nun die eigentliche Leseoperation an den der Kopie entsprechenden FSS weitergereicht. Hierauf führt der FSS diese aus und liefert dem FSC das gewünschte Operationsergebnis. Dem FSC wird automatisch danach vom FSS der Lesezugriff entzogen, so dass nach diesem (erfolgreichen) Abschnitt keine Freigabe-Anforderungen an den FSS verschickt werden müssen.

4. **Complete Computation/Read-Release**: Sollten die vier vorherigen Abschnitte *nicht* erfolgreich durchlaufen worden sein, wird als letzte Nachricht an den FSS eine Freigabe-Aufforderung für den Lesezugriff verschickt. Danach wird in jedem Fall die Task-History, abhängig vom Erfolg der Task-Inkarnation (SUCCESS oder FAILURE), aktualisiert und die entsprechende Antwort in die Message-Queue an den Task-Scheduler gegeben.

Nach jedem Abschnitt wird hier, wie auch bei dem Schreibzugriff, die Deadline mit der aktuellen netzweit gültigen Zeit verglichen, und sollte sie abgelaufen sein, wird eine entsprechende Fehlermeldung an den Task-Scheduler geschickt, die Task-History aktualisiert und etwaige gehaltene Reservierungen von Lesekopien durch die Nachricht *MSG_READ_RELEASE* an die entsprechenden FileServer-Server aufgehoben (wobei hierbei keine Antwort vom FSS erwartet wird).
Der zweite Fall einer Task-Inkarnation ist die Aufforderung zum Schreibzugriff auf die Ressourcen (*Read-Task*). Auch hierfür lässt sich der Ablauf, wie in der Abbildung 4.3 zu sehen, in mehrere Abschnitte aufteilen:

1. **Location**: Wieder müssen erstmal alle erforderlichen Ressourcen im Netz gefunden werden. Da ein Schreibzugriff aber auf alle vorhandenen öffentlichen Kopien (*public copies*) gleichzeitig zugreifen muss, werden hier keine konkreten Adressen, sondern nur die bloße Anzahl an Kopien ermittelt. Hierzu wird pro Ressource eine Anfrage per BROADCAST und eine an den LOCALHOST (da BROADCAST den eigenen Knoten nicht erreicht) verschickt, und die Anzahl der empfangenen Antworten gesichert.

2. **Allocation**: Da in MELODY das *Delayed Insertion Protocol* verwendet wird, um *cascading blocking* (siehe Abschnitt 1.3.9) zu verhindern, läuft für das Schreiben die Reservierung des Zugriffs mehrstufig ab. In dieser ersten Stufe wird der FSS aufgefordert, in die Queue für die Zugriffskandidaten (*candidate-queue* aufgenommen zu werden. Dies geschieht erst, wenn die execution-queue kein Element mehr enthält, und es wird auch erst dann eine Antwort vom FSS versendet. In der Zwischenzeit wird die Anfrage in die waiting-queue sortiert.

3. **Locking**: Im nächsten Schritt versucht der FSC von der candidate- an die erste Stelle der execution-queue beim FSS zu gelangen, womit er dann den sogenannten *lock* auf die Ressource erlangt hat. Der FSS trägt die Anfrage aus der candidate-queue aus und in die execution-queue wieder ein, jedoch können noch andere Task-Inkarnationen noch davor

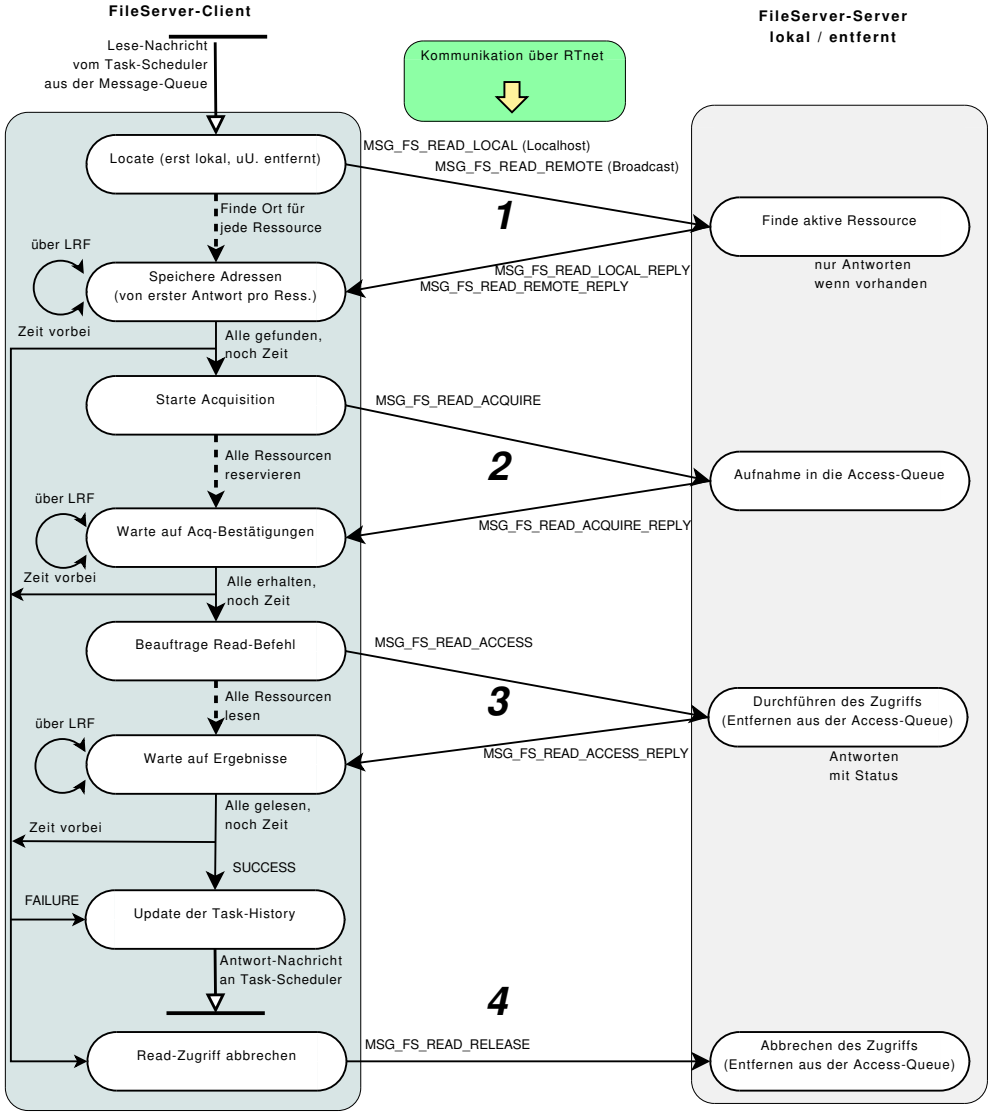

Abbildung 4.2: Ablauf eines Lesezugriffs zwischen FileServer-Client und -Server

in der Liste stehen. Die Bestätigung wird erst dann ausgesendet, wenn die Anfrage an den Anfang der Liste gelangt ist. Ein Sonderfall ist der von MELODY verwendete Call-Back-Mechanismus, der eine Prioritätsumkehr vermeiden soll: es kann passieren, dass ein neu in die execution-queue sortiertes Element direkt an den Anfang der Liste gelangt, noch vor früheren Anfragen. Nun wird vom FSS eine Anfrage an den bisherigen *lock*-Inhaber geschickt, ob er schon in seinem Computation-Abschnitt ist, d.h. ob demjenigen FSC bereits die *lock*-Nachrichten für alle seine Ressourcen vorliegen. Ist dies *nicht* der Fall, gibt der FSC diesen einen *lock* zurück, und der FSS übergibt den *lock* dem ersten Element der execution-queue.

4. **Computation**: Die letzten beiden Abschnitte ähneln wieder den letzten Abschnitten beim Read-Task: in diesem Abschnitt wird die gewünschte Operation an alle FSS geschickt, da zu diesem Zeitpunkt der FSC den *lock* für alle benötigten Ressourcen hat. Auf Seiten des FSS wird die Anfrage aus der execution-queue entfernt, und überprüft, welche Anfrage als nächstes den *lock* erhält: ist die execution-queue nach dem Entfernen nicht leer, kommt das neue erste Element zum Zug; hat sich die execution-queue dagegen geleert, werden alle Elemente der waiting-queue in die candidate-queue umgeschichtet, und die entsprechenden Nachrichten versendet.

5. **Complete Computation/Write-Release**: Sollten die bisherigen Abschnitte *nicht* erfolgreich durchlaufen worden sein, so wird an die FSS eine Freigabe-Aufforderung für den Schreibzugriff verschickt. Danach wird in jedem Fall die Task-History, abhängig vom Erfolg der Task-Inkarnation (SUCCESS oder FAILURE), aktualisiert und die entsprechende Antwort in die Message-Queue an den Task-Scheduler gegeben. Auf Seiten des FSS wird die Anfrage in den Listen gesucht und entfernt, und, je nachdem in welcher sie war, wird wie im vorherigen Abschnitt (nach der Vollendung der Schreiboperation) der nächste mögliche *lock*-Empfänger, durch Umverteilen der jeweiligen Queues, ermittelt.

Der FileServer-Server aktualisiert seine File-History immer entweder nach einer erfolgreichen Lese- oder Schreiboperation (mit SUCCESS), oder nach dem Erhalt der *Read-Release*- bzw. *Write-Release-Nachricht* (mit FAILURE). Als nächstes wird überprüft, ob nun einer der Werte in der File-History einen Grenzwert (*Threshold*) überschreitet, und schickt dann gegebenfalls einen dazu passenden Auftrag über die Message-Queue an den File-Assigner:

- Wurde der **FSWTh** überschritten, wird der FA aufgefordert, eine Möglichkeit zur Verringerung der Anzahl der public-copies im Netz zu suchen. Dadurch soll sich der Schreibzugriff für diese Ressource verbessern, da so auf weniger Kopien gleichzeitig zugegriffen werden muss.Als Antwort erhält man, wenn das Löschen möglich ist, eine IP-Adresse des Knotens, an dem die public-copy am ehesten gelöscht werden kann. Der FSS verschickt daraufhin (wenn es nicht seine eigene ist) eine Nachricht an den FSS dieses Knotens, mit der Aufforderung, die Kopie zu löschen. An diesem Knoten wird der Status der Kopie daraufhin von *active* auf *deleted* gesetzt, womit nur noch die schon empfangenen Zugriffe abgearbeitet werden, aber keine neuen mehr akzeptiert werden.

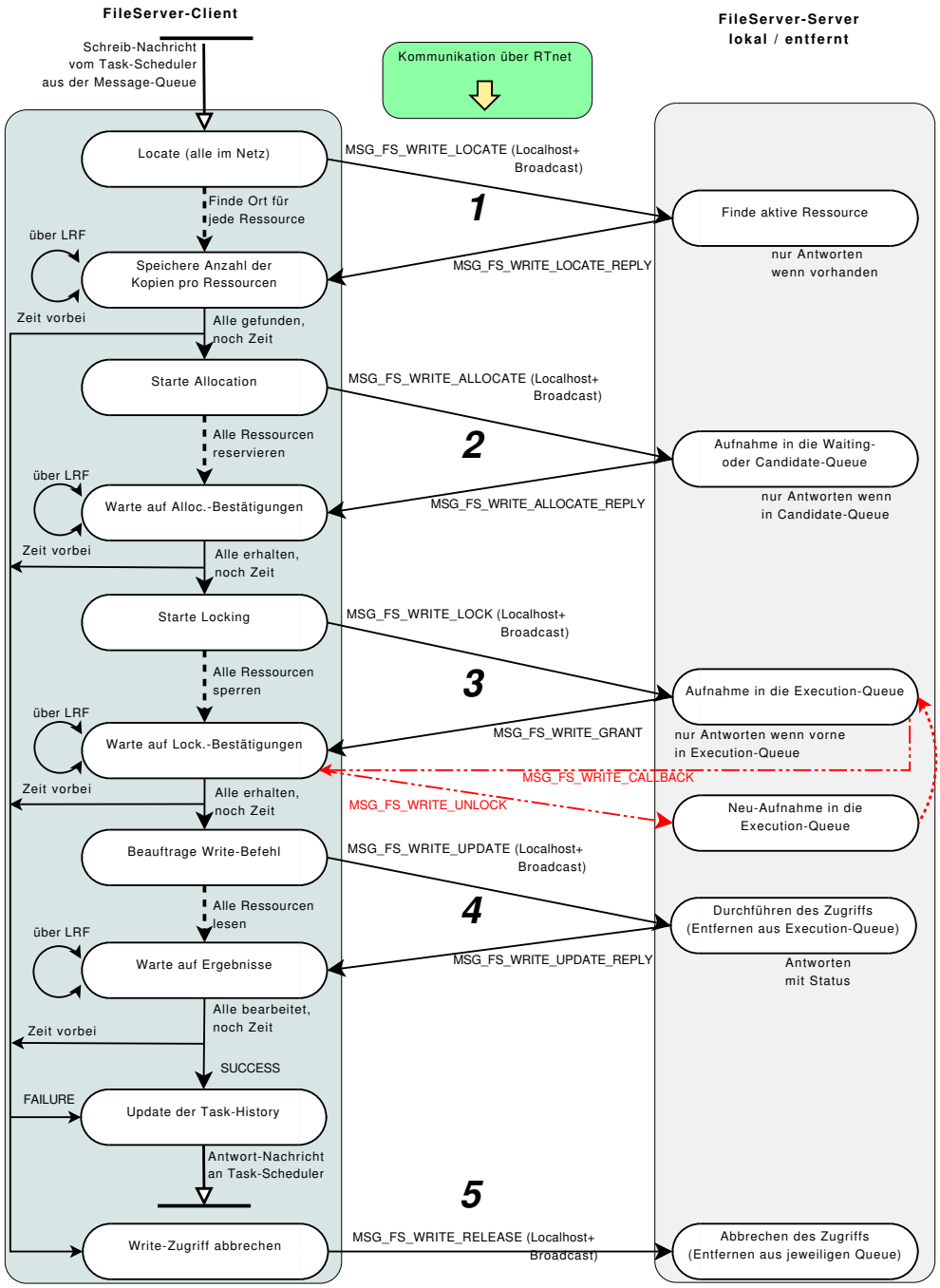

Abbildung 4.3: Ablauf eines Schreibzugriffs zwischen FileServer-Client und -Server

- Wurde der **FRRTh** überschritten, soll der Lesezugriff für lokale Task-Inkarnationen durch eine neue lokale private-copy verbessert werden. Hierzu wird der File-Assigner nicht benötigt, da die übrigen Kopien von einer zusätzlichen privaten Kopie einer Ressource nicht betroffen sind. Der FileServer-Server sendet an die übrigen FSS eine BROADCAST-Nachricht zur Suche der Ressource aus, und von demjenigen FSS, der als erster eine Antwort schickt, wird eine Nachricht mit dem aktuellen Inhalt der Kopie angefordert.
- Wurde der **FSRTh** überschritten, soll der Lesezugriff für sensitive Task-Inkarnationen für diese Ressource durch das Anlegen einer lokalen public-copy verbessert werden. Dazu wird nun wieder der File-Assigner benutzt, der überprüft, ob solch eine öffentlich Kopie noch erzeugt werden darf, und wenn ja, ob eine andere öffentliche Kopie von einem anderen Knoten zum lokalen bewegt werden kann (was vorzuziehen ist, da sich dann die Schreibzugriffe nicht verschlechtern sollten), oder ob eine zusätzlich Kopie erzeugt werden muss. Kann eine neue lokale public-copie erzeugt werden, wird der Status des File-Managers von *inactive* auf *partial* gesetzt, damit bei zukünftigen Zugriffen diese Kopie mit berücksichtigt wird. Ist die Kopie dann vollständig erzeugt, wechselt der Status von *partial* zu *active*.

4.4 Shutdown

In der Phase des Shutdowns werden alle verwendeten Realtime-Ressourcen, wie z.B. RTnet-Sockets, die Message-Queues und die RT-Heaps für die Kopien, wieder geschlossen bzw. freigegeben. Dabei werden die bisher nur auf den Heaps veränderten Ressourcen wieder auf die Festplatte geschrieben. Zur Auswertung speichert der Debug-Service nun die bisher empfangenen Nachrichten in Log-Dateien.
Durch diese Zugriffe verliert MELODY in dieser Phase wieder seine Echtzeitfähigkeit, da durch die Festplattenzugriffe Xenomai vom *primary* in den *secondary execution mode* wechseln muss. Zu beachten ist dabei, dass die Ressourcen in dem selben Ausführungs-Modus geschlossen werden müssen, in dem sie auch geöffnet bzw. erzeugt wurden, ansonsten schlägt der Vorgang fehl, und bei einem erneuten Programmstart können diese Ressourcen nicht mehr genutzt werden.
Auch hier wird der Wechsel zwischen den Phasen durch den Task-Scheduler ausgelöst, der wiederum auf eine Nachricht vom Master-Knoten reagiert. So ist sichergestellt, dass, sollte ein Knoten eine längere Task-Queue haben als alle anderen Knoten, er dennoch weiter die benötigten Ressourcen auf den entfernten Knoten allozieren kann.

5 Verteilte Experimente

In diesem Abschnitt werden einige Experimente mit dem entwickelten System beschrieben und die Auswirkung der Veränderung von Rahmenbedingungen dargelegt. Im Kern untersuchen die Tests die Funktionalität und die Performanz im verteilten system.
Als System werden in allen Tests ein Netz mit vier Knoten genutzt. Jeder Knoten ist dabei ein Debian-System der Version 3.1 mit Xenomai 2.3.1 und RTnet 0.9.9, die per Fast-Ethernet über einen dedizierten Router miteinander verbunden sind. Bei den Ergebnissen ist zu beachten, dass trotz identischen Ausgangskonfigurationen eigentlich niemals das exakt selbe Ergebnis zu erwarten ist, da der implementierte Task-Scheduler eine randomisierte Zeit abwartet, bis er die folgende Task-Inkarnation an den File-Server sendet. Der Zeitraum ist dabei nach oben begrenzt durch das Doppelte der Latency des vorhergehenden Tasks. Durch diese Verwendung von aperiodischen Tasks sollen wirklichkeitsnähere Versuchsbedingungen erzeugt werden. Im Rahmen des Setups für ein Experiment werden Task-Profil und die Verteilung der Ressourcen auf die einzelnen Knoten vor jedem Experiment zufällig gewählt, aber dann für die gesamte Testreihe beibehalten. Auf jedem Knoten existiert dann die selbe Reihenfolge an auszuführenden Tasks. Ein Testlauf ist entweder beendet, wenn die Deadline für einen kritischen Task verpasst wird, oder alle Tasks durchgelaufen sind.
Um überhaupt ein aussagekräftiges Ergebnis zu erhalten, müssen durch eine Vielzahl von Vorabversuchen günstige Werte für das Task-Profil ermittelt werden. Wählt man z.B. die Deadline zu großzügig, so gibt es überhaupt keine fehlgeschlagenen Task-Inkarnationen, ist die Deadline dagegen zu klein bemessen, bricht der Durchlauf schon noch sehr wenigen Inkarnationen ab, da ein kritischer Task abgebrochen wurde.

5.1 Funktionstest

Für einen ersten Test der Zusammenarbeit zwischen File-Server und File-Assigner wird ein Testprofil erstellt, bei dem nur eine Ressource pro Task angefordert wird. Damit soll die grundlegende Funktionalität der einzelnen Komponenten überprüft werden. Tabelle 5.1 fasst die Kenngrößen für diese erste Testreihe zusammen. So gibt es 16 unterschiedliche Tasks, die an jedem Knoten immer wieder in der gleichen Reihenfolge ausgeführt werden. Die Werte für die Deadline der Lese- und Schreibtasks von 25 msec hat sich in Vorversuchen als guter Kompromiss aus nicht zu häufiges oder zu seltenes Auftreten von Fehlschlägen einer Task-Inkarnation bei einem Ressourcenzugriff ergeben. Kritikalität und Sensitivität liegen zu Beginn am oberen Grenzwert,

um zunächst nur unkritische und robuste Tasks zu erzeugen, so dass etwaige Relokationen oder Replikationen von Ressourcen nicht schon gleich zu Beginn der Laufzeit nötig werden.

Für die Versuchsreihe gibt es fünf verschieden Task-Profile, die sich jeweils nur durch einen anderen Prozentsatz an Lese- bzw. Schreibtasks unterscheiden. Insgesamt sollen auf jedem Knoten jeweils 1500 Tasks ausgeführt werden und es werden jeweils zu jeder Sekunde registriert, wieviele der ausgeführten Tasks erfolgreich waren. Aus der Summe der Werte aller Knoten werden dann die nachfolgenden Diagramme generiert. Für jedes Task-Profil wird jedes der vier Modelle (MELODY-Modell, Basismodell, Private-Modell und Public-Modell; s.u.) fünf mal gestartet und dann ein typisches Ergebnis mit möglichst langer Laufzeit ausgewählt.

Als Vergleichsgröße zu dem **MELODY-Modell** werden die selben Task-Profile an drei weiteren Modellen angewendet: das Basismodell, das Private-Modell und das Public-Modell. Bei dem **Basismodell** bleibt die Ressourcenverteilung über die gesamte Laufzeit hinweg wie zu Beginn des Systems bestehen. Dagegen erlaubt das **Private-Modell** das Erzeugen und Löschen ausschließlich von *private copies*, wobei von jeder Ressource im Netz zu jeder Zeit nur genau eine Public-Copy existiert. Diese darf aber im Netz bewegt werden. Dagegen dürfen bei dem **Public-Modell** allein *public copies* erzeugt und gelöscht werden, *private copies* werden hierbei nicht berücksichtigt.
Für die Tests ist zu erwarten, dass das Basismodell im Vergleich am meisten fehlgeschlagende Tasks erzeugt, bei dem MELODY-Modell dagegen der geringste Prozentsatz. Die anderen beiden Modelle sollten dazwischen liegen, und je nach Art des Task-Profils, einmal das Private-Modell bei einer Überzahl von Lesetasks ein besseres Ergebnis erzielen, oder bei einer Mehrzahl von Schreibtasks das Public-Modell. Bei einem ausgeglichenden Profil sollten die beiden Modelle eine ähnliche Ausbeute an fehlgeschlagenden Tasks zeigen.

Anzahl der Knoten	4
Anzahl der Tasks	1500 pro Knoten
Anzahl der Ressourcen	16
Ressourcenzugriff pro Task	1
Deadline	25 msec
Criticality	8
Criticality-Threshold	2-8
Sensitivity	8
Sensitivity-Threshold	2-8
Anzahl *public copies* zu Beginn	1
Min-Public-Copies	1

Tabelle 5.1: Setup für den Funktionstest

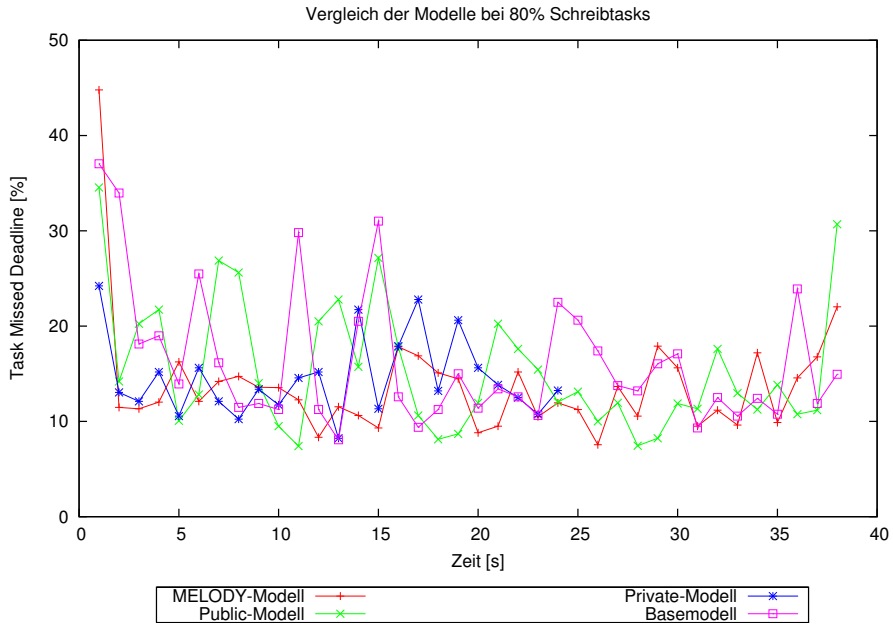

Abbildung 5.1: Testreihe 1: Write-Dominance (80% Schreibtasks)

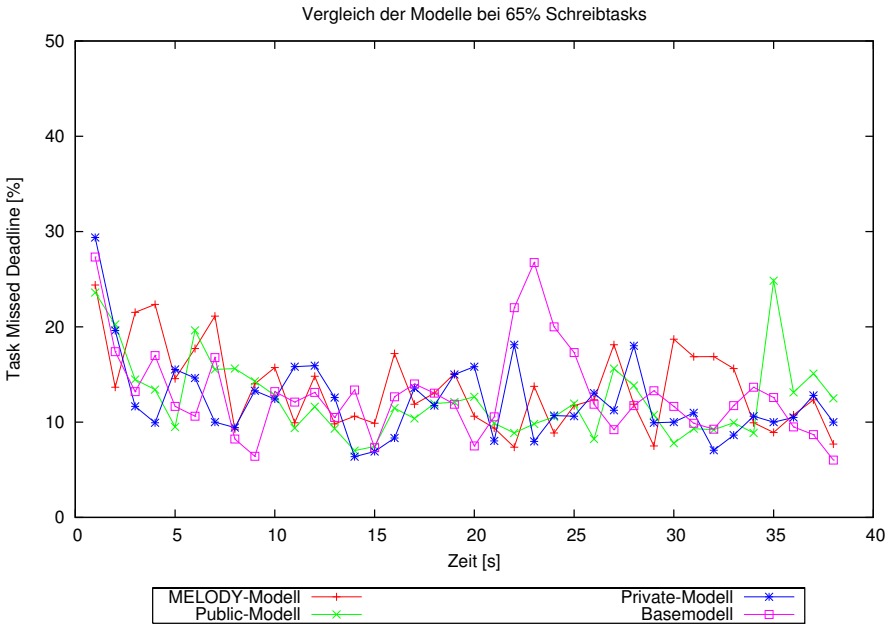

Abbildung 5.2: Testreihe 1: Write-Dominance (65% Schreibtasks)

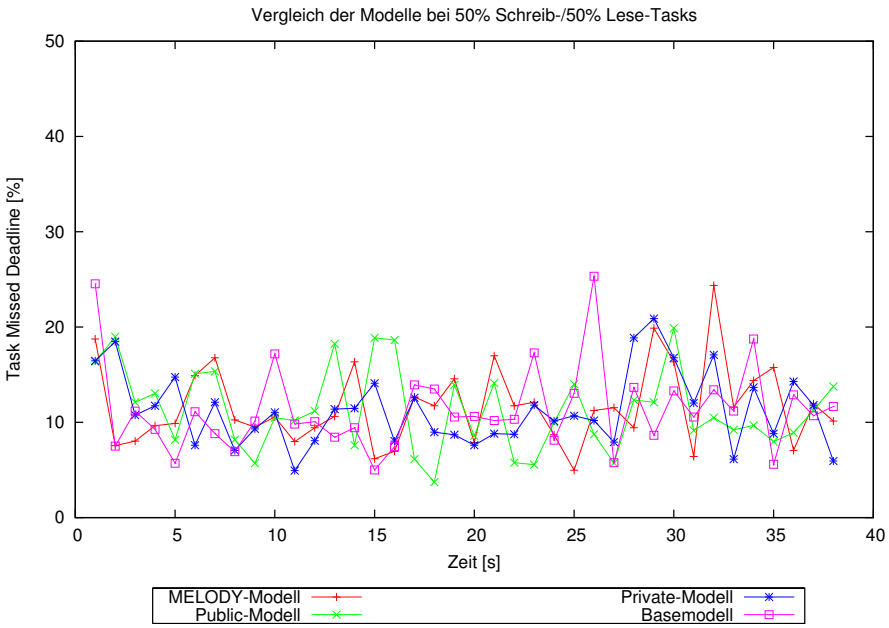

Abbildung 5.3: Testreihe 1: Even (50% Schreibtasks)

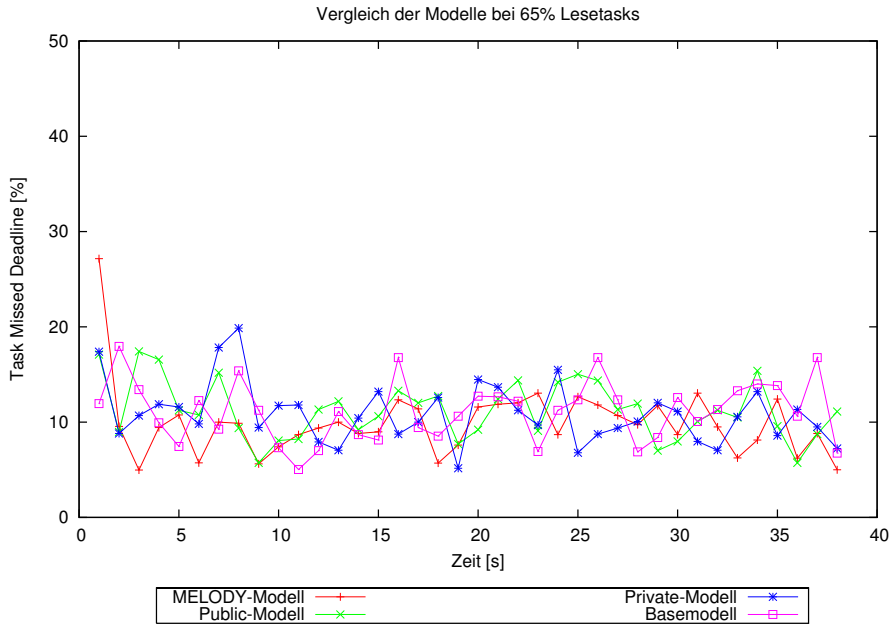

Abbildung 5.4: Testreihe 1: Read-Dominance (65% Lesetasks)

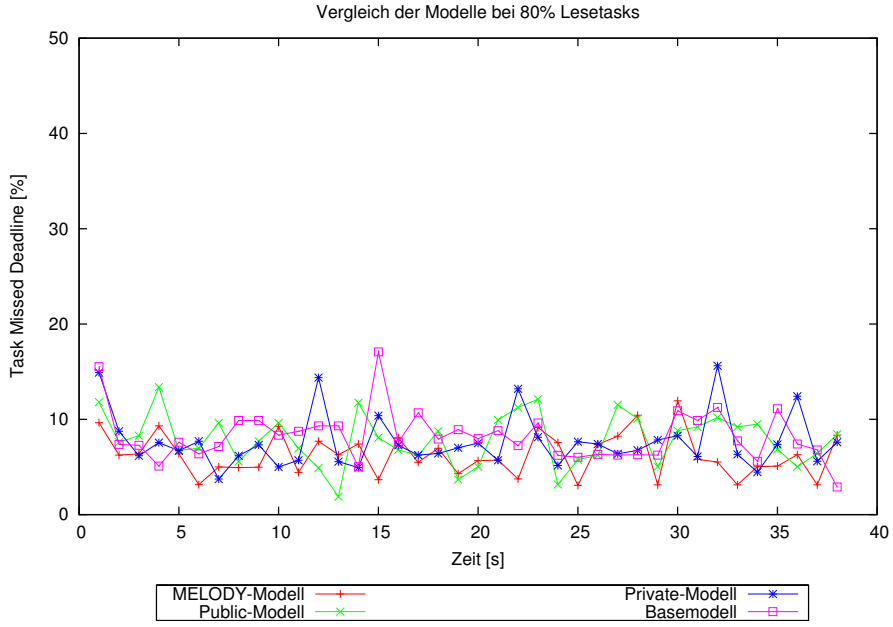

Abbildung 5.5: Testreihe 1: Read-Dominance (80% Lesetasks)

Die Ergebnisse dieser Basis-Funktionalitätstests sind nicht sehr aussagekräftig im Hinblick auf die verschiedenen Modelle, da nur sehr geringe Unterschiede zwischen ihnen feststellbar sind. Allerdings war während der Versuche erkennbar, dass bei dem Basismodell häufiger das System frühzeitig abgebrochen wurde, da ein kritischer Tasks fehlgeschlagen war. Auch gelang es bei dem Private-Modell nicht einmal bei einer Schreibtask-Dominanz von 80% alle Knoten bis zum Ende durchlaufen zu lassen, da immer einer der Knoten aufgrund eines abgebrochenen kritischen Tasks vorzeitig gestoppt ist. Dies lässt sich dadurch erklären, dass bei einer hohen Anzahl von Schreibtasks fast keine Verbesserung gegenüber dem Basismodell erreicht werden kann, zusätzlich aber der Overhead durch den File-Assigner und File-Server das Verhalten des Modells negativ beeinflusst.
Der zusätzliche Administrationsaufwand durch den File-Assigner und File-Server bei dem Public- und Private-Modell, im Besonderen aber beim MELODY-Modell ist für die im Diagramm erkennbaren und sehr ähnlichen Ergebnisse zwischen den einzelnen Modellen verantwortlich. Im Gegensatz zum Basismodell werden bei den anderen Modellen sehr viel mehr Nachrichten zwischen den Knoten versendet, und die Wartezeit auf die entsprechenden Antworten schlägt sich negativ auf den Durchsatz an Task-Inkarnationen nieder. Dennoch lässt sich in den Ergebnissen erkennen, dass die Implementierung des File-Assigners und -Servers grundsätzlich erfolgreich ist und erwartungsgemäß funktioniert.

5.2 Perfomanztest

Nach der erfolgreichen Überprüfung der Funktionalität der verteilten Komponenten wird nun das Testprofil so erweitert, dass statt nur auf einer nun auf eine größere Anzahl von Ressourcen pro Task zugegriffen wird. Dadurch sollten sich die Vorteile des adaptiven MELODY-Modells gegenüber den anderen Modellen deutlicher herausbilden.
Wie in der ersten Versuchsreihe gibt es 16 Ressourcen im Netz verteilt, die Deadlines werden, in Abhängigkeit, ob es ein Lese- oder Schreibtask ist, auf 38 msec bzw. 68 msec verlängert. Zufällig wird jedem Task für die Kritikalität und Sensitivität ein Wert von 4, 5 oder 6 zugeordnet und jede Ressource in ein- bis vierfacher Ausführung im Netz verteilt. Damit soll besser eine ”durchschnittliche” Startsituation erzeugt werden, die eher der typischen Umgebung für MELODY entspricht. Wieder wird bei dem einmal erzeugten Task-Profil (siehe Tabelle 5.2) in den weiteren Versuchen nur noch das Verhältnis von Lese- zu Schreibtasks variiert, in dem aus dem Schreibzugriff auf die Ressourcen ein Lesezugriff wird, wobei hier die Deadline entsprechend angepasst wird.

[1] Bei dem Private-Modell immer nur 1

Anzahl der Knoten	4
Anzahl der Tasks	1500 pro Knoten
Anzahl der Ressourcen	16
Ressourcenzugriff pro Task	2-4
Deadline	38 msec / 68 msec
Criticality	4-6
Criticality-Threshold	2-8
Sensitivity	4-6
Sensitivity-Threshold	2-8
Anzahl *public copies* zu Beginn	1-4[1]
Min-Public-Copies	1

Tabelle 5.2: Setup für den Performanztest

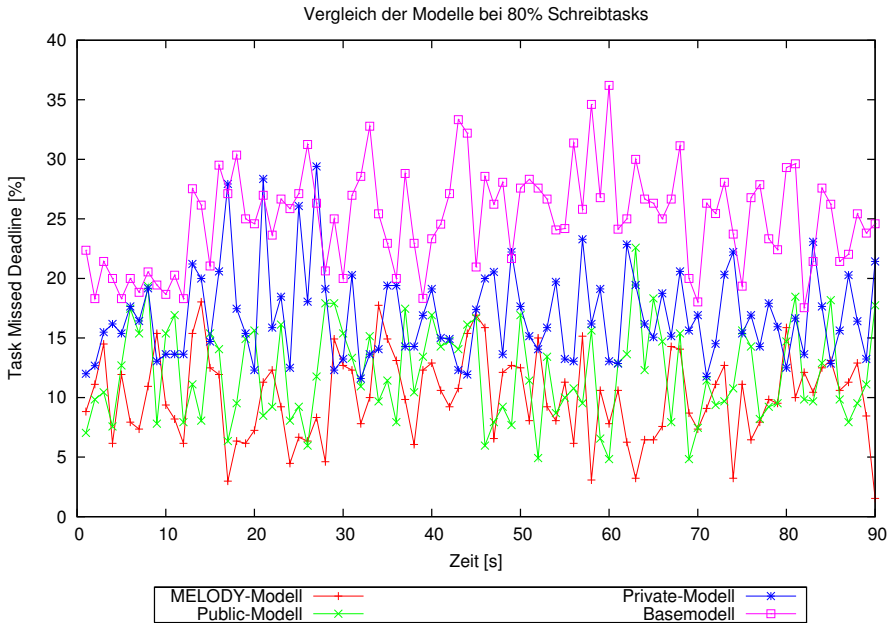

Abbildung 5.6: Testreihe 2: Write-Dominance (80% Schreibtasks)

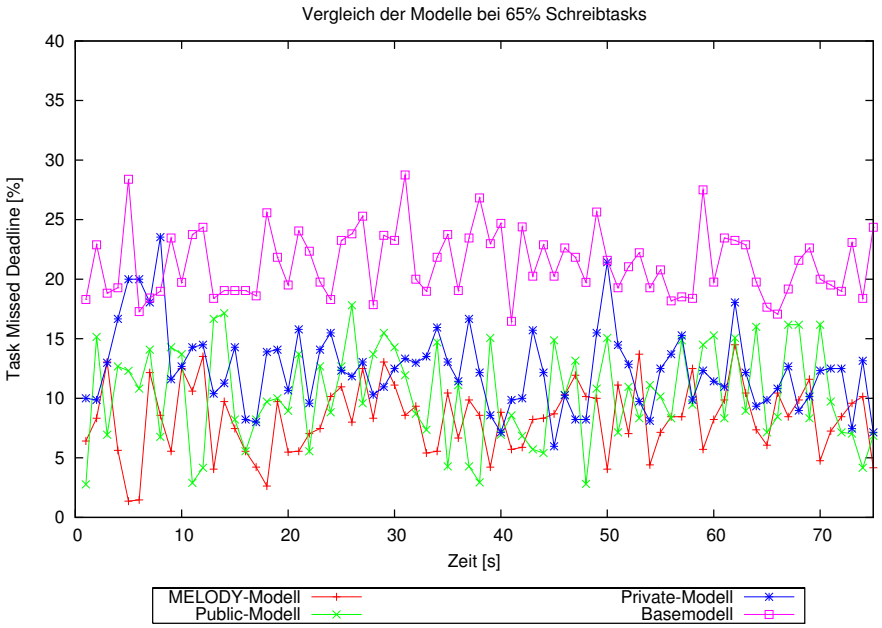

Abbildung 5.7: Testreihe 2: Write-Dominance (65% Schreibtasks)

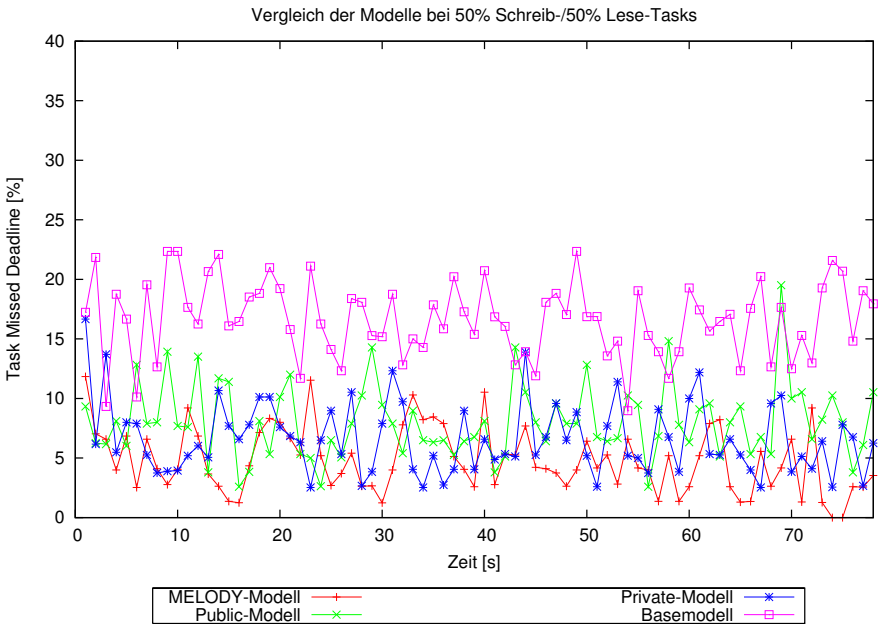

Abbildung 5.8: Testreihe 2: Even (50% Schreibtasks)

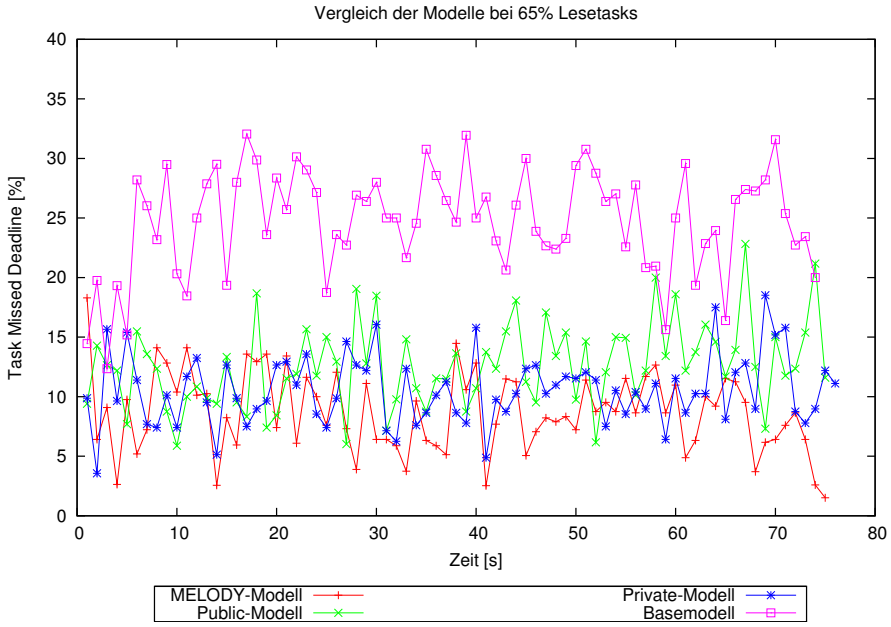

Abbildung 5.9:)
Testreihe 2: Read-Dominance (65% Lesetasks)

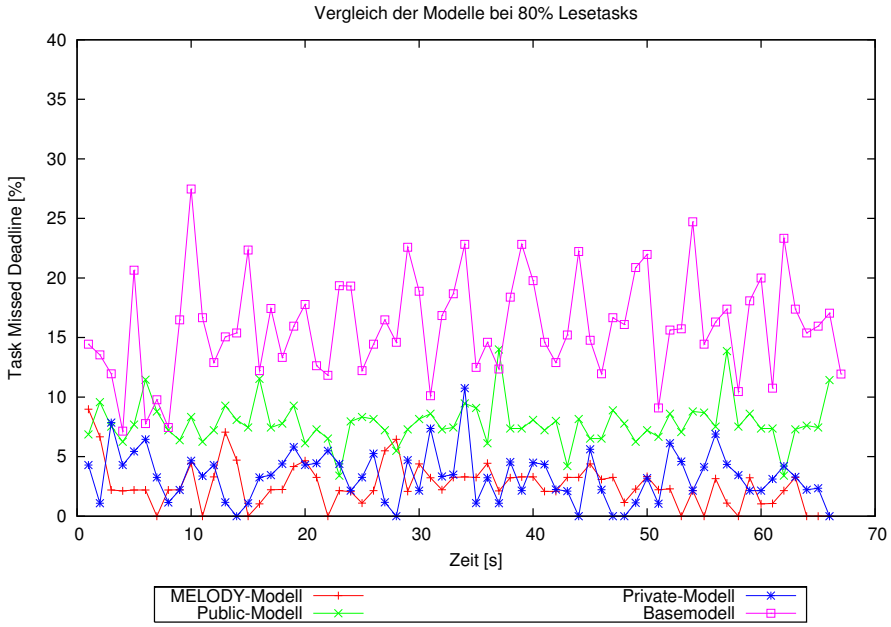

Abbildung 5.10: Testreihe 2: Read-Dominance (80% Lesetasks)

In dieser Versuchsreihe lässt sich nun sehr gut das eingangs erhoffte Verhalten der einzelnen Modelle wiederfinden. Das Basismodell liefert die schlechtesten Ergebnisse, und es erforderte auch eine größere Anzahl an Versuchen, speziell bei den Profilen mit mehr als der Hälfte Schreibtasks, um überhaupt einmal auf allen Knoten die 1500 Task-Inkarnationen abarbeiten zu können.
Das Public-Modell ist bei einer Vielzahl von Schreibtasks besser als das Basismodell, bei dem Private-Modell ist dies bei einer Vielzahl von Lesetasks der Fall. Je weiter sich die Anzahl der Lese- und Schreibtasks angleicht, desto geringer sind aber die Unterschiede zwischen diesen beiden Modellen. Die adaptiven Eigenschaften ermöglichen dem MELODY-Modell über alle Task-Profile die besten Ergebnisse zu liefern. Die Unterschiede sind zwar, verglichen mit dem Private-, und dem Public-Modell, oftmals gering, dafür liefert das MELODY-Modell unabhängig von der Art der Ressourcenzugriffe durchgängig gute Ergebnisse. Der Overhead, der durch die Kommunikation von File-Assigner und -Server erzeugt wird, nimmt trotz größerer Anzahl von Ressourcenzugriffe im Vergleich zum Funktionstest nur leicht zu, da er zu einem großen Teil aus dem Auffinden der Ressourcen besteht. Dies geschieht zu Beginn des Tasks mittels einer Broadcast-Nachricht pro Ressource, und es werden dann parallel Antworten bis zu einem Timeout gesammelt. Da sich somit der Overhead nur geringfügig vergrößert, lässt sich so der größere Abstand zwischen den Ergebnissen von Basismodell und den übrigen Modelle erklären.

5.3 Zusammenfassung

Weitere aussagefähige Ergebnisse lassen sich erwarten, sobald umfangreichere verteilte Experimente in einem größeren Netz mit wesentlich mehr Knoten durchgeführt werden. So kann zum Beispiel die Anzahl der *public copies* im System viel stärker variiert werden, ohne dass nach relativ kurzer Laufzeit der Grenzwert für die Mindestzahl an Kopien erreicht wird.

6 Fazit und Ausblick

Diese Arbeit berichtet über die Entwicklung und Implementierung der MELODY-Module File-Server und File-Assigner, unter einer realzeitfähigen Linuxumgebung. Ein beträchtlicher Teil der Arbeitszeit befasst sich mit der Auswahl einer geeigneten Linuxumgebung und dem Austesten ihrer Eigenschaften (siehe Kaitel 2). Die Auswahl erfolgt sehr gewissenhaft, da sich zukünftige Weiterentwicklungen des MELODY-Systems auf diese Entscheidung stützen werden.
Das Zusammenspiel von Xenomai und RTnet hat sich im Laufe unserer Implementierung als sehr positiv erwiesen, so war auch die Kommunikation mit den Entwicklern dieser Projekte sehr hilf- und aufschlussreich.
Zusammenfassend lässt sich feststellen, dass File-Server und File-Assigner zufriedenstellend ihre an sie gestellten Aufgaben wahrnehmen. Die Verwendung der nur rudimentären Funktionalität des Task-Schedulers und des Runtime-Monitors lassen jedoch nicht viel Spielraum für über die dargestellten Experimente hinausgehende Szenarien.
Weitergehende Arbeiten sollen die noch verbleibenden Teile des MELODY-Systems vervollständigen und sich dann mit der interessanten Aufgabe befassen, die in früheren Simulationen und Experimenten festgestellten Eigenschaften des MELODY-Systems in einem konkreten Netzverbund zu bestätigen. Danach steht dem Einsatz in unterschiedlichsten verteilten Anwendungen nichts mehr im Wege. So bietet es sich an, MELODY bei aktuellen Forschungsprojekten aus dem Bereich der verteilten adaptiven Realzeitanwendungen, wie STOP (Stauvermeidung durch on-online Verkehrsplanung auf der Basis naturinspirierter Algorithmen) [WEDDE et al.] oder DEZENT (Dezentrales Management verteilter regenerativer Energieerzeugung durch Realzeit-Multiagentensysteme) [WEDDE und LEHNHOFF] als verteiltes Betriebssystem zu nutzen. Wir werden über weitere Fortschritte in verwandten Arbeiten berichten.

Literaturverzeichnis

[ALIJANI 1988] ALIJANI, G.S. (1988). *Object Mobility in Distributed Computer Systems*. Doktorarbeit, Wayne State University, Detroit Michigan.

[BENCHELIH 2007] BENCHELIH, MOHAMMED (2007). *Entwicklung und Implementierung eines verteilten File-Assigners für das verteilte Realzeit-System MELODY unter Linux*. Diplomarbeit, Universität Dortmund.

[DANIELS 1992] DANIELS, DOUGLAS C. (1992). *The Design and Analysis of Protocol for Distributed Resource Scheduling Under Real-Time-Constraints*. Doktorarbeit, Wayne State University, Detroit Michigan.

[KEMPER 2007] KEMPER, NILS (2007). *Entwicklung und Implementierung eines verteilten File-Servers für das verteilte Realzeit-System* MELODY *unter Linux*. Diplomarbeit, Universität Dortmund.

[KISZKA 2007] KISZKA, JAN (2007). *Dipl.-Ing. Jan Kiszka, Homepage*. Online at `http://www.rts.uni-hannover.de/index.php/Dipl.-Ing._Jan_Kiszka`.

[KISZKA et al. 2006] KISZKA, JAN, P. GERUM und G. CHANTEPERDRIX (2006). *A Tour of the Native API*. Online at `http://www.xenomai.org/documentation/branches/v2.3.x/pdf/Native-API-Tour-rev-C.pdf`.

[KISZKA et al. 2007] KISZKA, JAN, P. GERUM und G. CHANTEPERDRIX (2007). *API documentation*. Online at `http://www.xenomai.org/index.php/API_documentation`.

[LIND 1999] LIND, JON A. (1999). *Realisation of the Highly Integrated Distributed Real-Time Safety-Critical System Melody*. Doktorarbeit, University of Dortmund, Fachbereich Informatik, Universität Dortmund.

[STEVENS et al. 1998] STEVENS, W. RICHARD, B. FENNER und A. M. RUDOFF (1998). *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*. Prentice Hall, Englewood Cliffs, NJ.

[WEDDE et al. 1991] WEDDE, HORST F., D. DANIELS und D. HUIZINGA (1991). *Efficient Distributed Resource Scheduling for Adaptive Real-Time Operating System Support*. In: DEHNE, F., Hrsg.: *Advances in Computing and Information - ICCI'91*, Bd. 497 d. Reihe *Lecture Notes in Computer Science*. Springer-Verlag.

[WEDDE und DEKKER 1994] WEDDE, HORST F. und M. DEKKER (1994). *Real-Time Operating Systems and Software: State of the Art and Future Challenges*. In: KENT, A. und J. WILLIAMS, Hrsg.: *Encyclopedia of Microcomputers*, Bd. 14.

[WEDDE et al. 1990] WEDDE, HORST F., D. HUIZINGA, G. KANG und B.-K. KIM (1990). *Melody: A Complete Decentralized Adaptive File System for Handling Real-Time Tasks in Unpredictable Environments*. Real-Time Systems, 2(4):347–364.

[WEDDE et al. 1993] WEDDE, HORST F., B. KOREL und J. LIND (1993). *Highly Integrated Task and Resource Scheduling for Mission-Critical Systems*. In: *Proc. of the EUROMICRO'93 Workshop on Real-Time Systems*, Oulu, Finland. Euromicro, IEEE Computer Society Press.

[WEDDE und LEHNHOFF] WEDDE, HORST F und S. LEHNHOFF. *DEZENT - Dezentrale vernetzte Energiebewirtschaftung auf Basis eines verteilten adaptiven Realzeit-Multiagentensystems*. Online at `http://ls3-www.cs.uni-dortmund.de/Projects/Dezent/index.xml?lang=de`.

[WEDDE et al.] WEDDE, HORST F., S. LEHNHOFF und B. VAN BONN. *STOP - Stauvermeidung durch on-line Verkehrsplanung*. Online at `http://ls3-www.cs.uni-dortmund.de/Teachings/TaskForces/Stop/index.xml?lang=de`.

[WEDDE und LIND 1997] WEDDE, HORST F. und J. A. LIND (1997). *Building Large, Complex, Distributed Safety-Critical Systems*. Real-Time Systems, 13(3).

[WEDDE et al. 1994] WEDDE, HORST F., J. A. LIND und A. EISS (1994). *Achieving Dependability in Safety-critical Operating Systems Through Adaptability and Large-Scale Functional Integration*. In: *Proc. of the ICPAPDS'94 International Conference on Parallel and Distributed Systems*, Hsinchu, Taiwan.

[WEDDE et al. 1996] WEDDE, HORST F., C. STANGE und J. A. LIND (1996). *Integration of Adaptive File Assignment into Distirbuted Safety Critical Systems*. In: *21st IFAC/IFIP Workshop on Real Time Programming*, Gramado, RS,Brazil.

[WEDDE und XU 1992] WEDDE, HORST F. und M. XU (1992). *Scheduling Critical and Sensitive Tasks with Remote Requests in Mission-Critical Systems*. In: *Proc. of the EUROMICRO '92 Workshop on Real-Time Systems*, Athens, Greece. Euromicro, IEEE Computer Society Press.