修 士 論 文

連続メディアデータ処理とそのアプリケーションの製作を支援するツールキットに関する研究

指導教官　中島達夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

大平浩貴

平成9年 3月 24日

要旨

現在、ビデオプレーヤやビデオ会議システムといったマルチメディアアプリケーションが一般に浸透しつつある。これらのソフトウェアが扱っているビデオデータやオーディオデータは連続メディアデータの一種であり、時間に依存していて、大容量になりやすいという特徴がある。

この時間に依存しているという特性のために連続メディアデータを処理するソフトウェアを作成するのは難しく、また作業も膨大になる。連続メディアアプリケーションを効率的に制作するためには制作を支援するシステムが必要である。

本研究は連続メディアアプリケーションの構築法を考察し、連続メディアアプリケーションの制作を支援するツールキットを作成、さらにそれを利用して実際に連続メディアアプリケーションを製作して評価を行なう。

このツールキットはメディアデータを処理する機能だけでなく、メディアデータの流れを抽象化してユーザからの各種制御命令を処理する。さらに新たな命令の追加が可能な構成にする。

目 次

第 1 章

はじめに

1.1 現状と問題点

近年、情報工学を中心に材料科学や電子工学などの発展とともにパーソナルコンピュータの演算能力、I/O 能力が飛躍的に向上している。

この計算機資源、環境の向上にあわせて、個人が購入できる程度のパーソナルコンピュータでも動画データや音声データなどを扱うことが出来るようになっている。その最たる例がビデオプレーヤや、ビデオコンファレンスのソフトウェアであり、具体例として CU-SeeMe[9] などがある。

このような同画像、音声データをはじめとしてあらゆる連続メディアデータは時間に依存しているという特徴を持っている。これまでのテキストデータやイメージデータと異なり、連続メディアデータの入出力や加工などの処理を行う時は常に時間を考慮しなければならない。

連続メディアデータはその名の通り時間的に連続したデータである。しかし計算機では連続的な時間を扱うことができないために、このデータを一定の短い時間で分割して処理をする。例えば、計算機が連続メディアデータを取得する作業は「一定の周期でデータを観測してその値を離散化する」というものになる。このようにしてデータを取得するので、アプリケーション内部では細分化された単位データが次々と生成される。

このような状態では、単位データの処理も次から次へ高速に行なわなければならない。もしある単位データの処理が滞ったとすると次のデータが取得できなくなって連続情報の一部が欠落してしまう。このようなことが頻発すればデータの信頼性が失われてしまう。

つまり連続メディアデータは時間に依存しているので、連続メディアデータの処理も時間的な管理をしなければいけない。このような時間管理を実装するのは非常に難しいため、連続メディアアプリケーションの製作を支援するようなシステムが必要である。

過去の研究ではこのメディアデータ処理と処理の時間的な管理を抽象化することに主眼を置いていた。その一つにメディアデータ処理を行なうサーバを作成し、それをユーザが利用するという形式を採用した

ものがあった。このシステムでは、アプリケーションプログラマがサーバに対してメディアデータ処理要求を通知すれば、あとはサーバがメディアデータ処理を代行するというシステムである。しかしながらこのような構成ではメディアデータ処理を一か所で集中して実装しているため、新たな機能を追加するためには巨大なサーバプログラムを書き換える必要がある。つまりこのようなメディアデータ処理の機能を単純にサーバにまとめるような実装は拡張性に乏しいといえる。

これを考慮して新たに提案されたのが連続メディアツールキットである。これは基本的に何らかの言語のライブラリの形で提供され、このライブラリを利用してメディアデータ処理を行なう。このライブラリはメディアデータ処理機能を細分化したもの (モジュール) を複数提供している。ユーザはこのモジュールを組み合わせて目的の処理を行い、また新たに機能を拡張する時はこのモジュールを新規に作成すれば良い。これによって、新機能を追加もメディアデータ処理機構そのものを改良する必要がなくなり、モジュールの追加だけで対応できるようになった。

しかしながら、この構成には問題がある。データ処理をモジュールに細分化したため、メディアデータはモジュールの間を流れるように動作するようになった。

過去の単純なサーバ型の実装の時はメディアデータ処理を制御する命令はサーバに通知するようになっていた。つまりメディアデータの処理開始命令や停止命令は全て単一のサーバに対して要求すれば良かった。しかしメディアデータ処理機能が Module に細分化され、それらを接続してメディアデータ処理構造を作成するようにした結果、メディアデータ処理を制御するような各種命令は沢山のモジュールの中から適切なモジュールだけを選んで適切な命令を送るようにしなければならない。このためアプリケーションプログラマへの負担は単純なサーバ型の実装よりも大きいといえる。

またコンピュータが扱う連続メディアデータは連続値を細分化したものであり、大容量となりやすい。時間依存特性を満たしつつ、大容量のデータを処理しなければいけないため、連続メディアデータの処理は CPU 資源を大きく消費することになる。連続メディアアプリケーションは常に時間に依存した処理を行っている為、CPU 資源が足りなくなって処理がサスペンドされると、時間要求を満たせなくなってしまう可能性がある。

このため、連続メディアアプリケーションは CPU 資源を過剰消費しないように自分自身の動作を見張って、過剰消費が起こったら処理量を自分で軽減するような機構が必要である。

1.2 目的

これまでの連続メディアアプリケーションが持っている問題点を考慮して、新たな連続メディアアプリケーションの構成を提案するのが本研究の目的である。

このために、連続メディアデータ処理の問題点を考え、それを解決できるようなアプリケーションの構成を提案、実際にツールキットを実装する。さらにそれを利用して連続メディアアプリケーションを作成し、正常に動作すること検証する。

本研究では連続メディアデータの例としてビデオデータを扱っている。しかしながら、本研究で提案する設計、作成するツールキットはビデオデータ処理専用のものではなく、あらゆる連続メディアデータを対象としている。

1.3 本論文の構成

本論文の構成を以下に示す。

第 2 章 連続メディアデータ

第 2 章では連続メディアデータの特徴とそれを処理する際に考慮しなければいけないことがらと実際の処理手法について述べる。

第 3 章 関連研究

第 2 章で示した問題を解決する為に作成された既存のシステムを考察し、その問題点を示す。その後、本研究で提案するツールキットがサポートすることがらを明確にする。

第 4 章 ツールキットの設計

本研究で提案する連続メディアアプリケーションの構成に付いて述べる。

第 5 章 ツールキットの実装

ツールキットを実装する環境、ツールキットが提供するオブジェクトを示し、各種操作に応じてどのような内部処理を行なうかを説明する。

第 6 章 ツールキットの応用

実装したツールキットの応用例と、そこからの拡張の例を示す。

第 7 章 評価

実装したツールキットを利用してビデオプレーヤを製作し、動作結果を考察して連続メディアアプリケーションへの要求を満たしていることを確認する。

第 8 章 結論と将来の課題

本研究で提案した連続メディアアプリケーション構成をまとめ、今後研究を発展させる上で問題となることについて述べる。

第 2 章

連続メディアデータ

2.1 連続メディアデータの特徴と処理の問題

2.1.1 一般的な連続メディアデータの特徴

本研究では連続メディアデータを扱う。

連続メディアデータは一般に以下の 2 つの特徴を持っている。

- 時間依存特性を持つ
- 大容量のデータとなりやすい

実世界には時間に依存して推移するデータが沢山ある。このようなデータの例として、気温や降水量、体温や血圧、そして本研究で中心的に扱う音声や同画像も時間に依存して推移する。このようなデータをコンピュータで扱う為には時間的に離散化する、いわゆる標本化という作業を行う。こうして一定時間間隔でサンプリングされたデータを本研究では連続メディアデータと呼ぶ。連続メディアデータはこのような手順で計算機に取り込まれたものなので、連続メディアデータには時間に依存しているという特徴がある。

たとえば、気温が××度であると言ってもそれだけでは意味がない。いつの気温なのかという時間的属性が必要である。また、気温はある瞬間だけ存在するのではなく、測定期間という幅をもって推移する情報である。つまり気温は朝から夜、翌日の朝…と連続的に変化する値である。もしその気温データが途切れていたりすればそれはデータの異常である。測定者がその途切れた部分の情報を欲していたのなら、それはデータそのものの価値が失われたことになってしまう。

つまり連続メディアデータは一定周期で観測されたデータである。このため、観測する周期が不安定になるとデータそのものの信頼性が低下してしまう。つまり連続メディアデータを取得する時はデータ取得処理を時間的に管理する必要がある。このように連続メディアデータを処理する時は常にデータの時間依存特性を考慮しなければならない。

連続メディアデータのもうひとつの特徴として、一般にデータ量が大きいということがある。

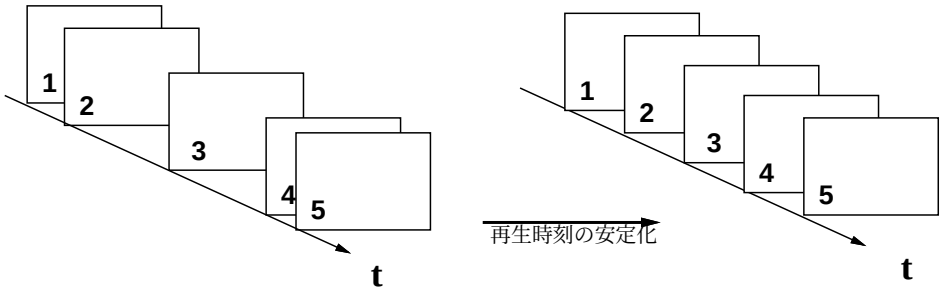

図 2.1: ジッタコントロールの概念図

ここでも同様に気温の測定を例にする。仮に単一の気温値のデータ量を 1 とする。気温を 1 分おきに測定し、それをデータストアに保存することを考える。もし、このままデータ圧縮などの操作を行なわないなら、24 時間測定するとデータ量は $1 \times 60 \times 24 = 1440$ となる。つまり、単一のデータとして測定した時の 1440 倍の大きさとなる。連続メディアデータを処理するアプリケーションを作成するのなら、大容量のデータを扱うことを考慮して設計しなければならない。

2.1.2 ビデオデータの特徴とビデオプレーヤが解決すべき問題点

本研究で扱うビデオデータも連続メディアデータであり、時間依存特性を持っている。

ビデオデータ再生処理を例に取って連続メディアデータの処理について考える。

動画データは静止画像を複数枚用意してそれを連続的に画面表示することで再生できる。もしディスクあるいはメモリなどに保存されたデータを時間を全く考慮せずに画面に出力したら、高速な計算機では本来何十分もあるデータの再生がほんの数分で終了してしまうだろうし、逆に低速な計算機であれば 1 秒分のデータを数 10 秒かけてゆっくりと再生してしまうことになる。圧縮された画像を展開するコストの変動も無視できなく、高速に展開できる部分はビデオが速く進んでしまい、展開に時間がかかる部分はゆっくりと再生されることになってしまう。

またビデオデータは非常に容量が大きく、現状ではデータを全てメモリ上に保持してそれを出力するということが難しい。つまり、ディスクなどの外部記憶からデータを読み込まざるを得ない。ディスク上のデータを読み込む場合、データを読み込むヘッドがディスク上を機械的に移動するせいで動画の各コマ (以下フレーム) の取得時刻が著しく変動する。

本来は一定の時間間隔でフレームを規則正しく出力しなければいけないが、上記のような様々な理由でフレームの出力時刻が変動する。この単位データの処理時刻変動をジッタと呼ぶ。このジッタの発生とその除去の様子を図 2.1に示す。

図 2.1中で、左側がジッタを除去する前の状態である。各フレームの時間間隔は不安定である。このジッタを除去し、右側のようにフレーム間の時間間隔を一定にしなければならない。このジッタを除去しなけ

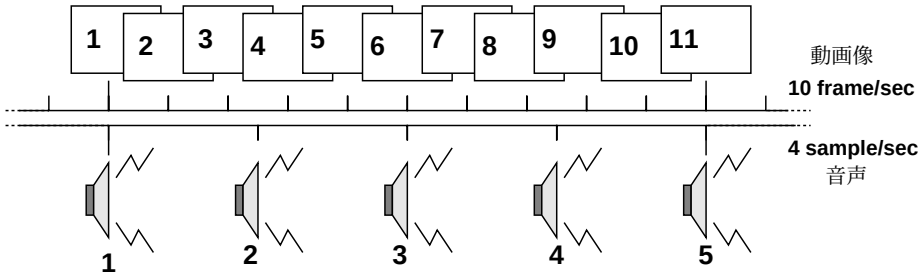

図 2.2: リップシンクの概念図

れば、動画像の再生速度は波を打ったものになってしまい、もとの動画像とは異なったものになってしまう。連続メディアデータの処理ではこのジッタ除去 (**ジッタコントロール**) が必ず必要になる。

ジッタコントロールは単一の動画像データを扱う為に必要な時間依存処理である。しかし、ビデオプレーヤは単一の動画像データを扱うだけではなく、動画像データと音声データの両方を扱わなければならない。

連続メディアデータ処理システムでは、単一のメディアデータ処理の流れをストリームと呼ぶ。本研究で扱うビデオプレーヤは、「動画像データ」と「音声データ」の 2 つの連続メディアデータを処理し、出力する物である。つまり「動画像データストリーム」と「音声データストリーム」の二つを構築し、それらを並行して動作させなければならない。

この複数のデータストリームを処理する際にも時間的問題がある。ビデオプレーヤでは、「動画像データ」と「音声データ」の二つを平行して出力するが、平行する二つのストリームの間で時間的な同期をとならなければいけない。つまり**ストリーム間の同期**を考慮しなければならない。

たとえば、人が喋っているシーンを盛り込んだビデオデータを考える。もし動画像と音声が同期していなければ、フレームの中で喋っている人の唇の動きと音声がずれて出力されることになってしまう。

この状況を例に取って「音声」と「画像」のような異なった連続メディアデータを処理する時に必要な同期処理、つまりストリーム間同期をリップシンクと呼ぶこともある。リップシンクの様子を図 2.2に示す。

図 2.2では、画像データは 1 秒の間に 10 フレームを出力する。これに対して音声データは 1 秒の間に 4 サンプルを出力する。この二つのデータは全く異なるタイミングでデバイスに出力するのだが、出力するペースは同期していなければいけない。ビデオプレーヤだけでなく、異なった複数の連続メディアデータを扱う場合は必ずリップシンクを考慮しなければならない。

このように、連続メディアデータを処理する時は必ず時間管理を行わなければならない。

2.1.3 連続メディアデータ処理と負荷

連続メディアデータの処理は以下のような問題をはらんでいる。

- 長時間動作する

- 処理内容によってはCPU 資源を大きく消費する

特にビデオデータの処理はこの問題が顕著に現れる。例えば、ビデオの単位時間あたりの出力フレーム数 (以下フレームレート) が 30frame/sec だとする。周期は 33msec であり 33msec ごとに画像データの取得、データのコンバート、ジッタ吸収機構への挿入と取り出し、画面への出力を行なうことになる。しかも、ビデオデータの再生中はこの作業をずっと続ける。

CPU 資源が十分にないと「ビデオデータの時間要求を満たすことが出来ない」だけでなく、「並行して動作する他の重要なプロセスが使用すべき CPU 資源を奪ってしまう」ことにもなる。

しかも、起動時には CPU 資源に十分な余裕があったとしても長時間メディアデータの処理が続くため動作中に急激に環境が変化する可能性もある。例えば、起動時には 30frame/sec のフレームレートでも処理が十分に間に合っていたとする。このような時にユーザが平行してプログラムのコンパイルを始める場合を考える。プログラムのコンパイルはCPU 資源を大きく消費するだけでなく、頻繁にディスク上のデータを読み書きするためディスクやインターフェースカードのバンド幅も大きく消費するため、結果として各種計算機資源が急激に不足することになる。

このような場合は、ビデオプレーヤの処理かコンパイルの処理かどちらかを優先させる必要がある。もしビデオプレーヤを優先的に使用したいのならビデオプレーヤの優先度を高くすればよい。こうすることでビデオプレーヤの動作を優先的に行い、余った CPU 資源でコンパイルを行うようになる。

しかしこの逆のコンパイラを優先させたいという時はビデオプレーヤは特殊な処理が必要になる。もし単純にビデオプレーヤの優先度を下げると、コンパイラが優先的にCPU 資源を消費する。このためビデオプレーヤの処理に必要な CPU 資源が確保できなくなり、結果としてメディアデータ処理の時間管理に失敗する。このようなことをなくす為に、ビデオプレーヤは自身が使用して良いCPU 資源を認識して、実際に消費する CPU 資源に応じてメディアデータ処理を軽減するような機構を用意しなければならない。つまりビデオプレーヤは計算機資源の予約や監視とそれに応じた対策が必要である。

2.2 連続メディアデータ処理の時間管理

2.1.2節で挙げた連続メディアデータ処理の問題点をまとめると以下のようになる。

- ジッタコントロールが必要
- ストリーム間同期 (リップシンク) が必要
- 各種資源の過剰消費を抑える機構が必要

これらの解決法を以下で示す。

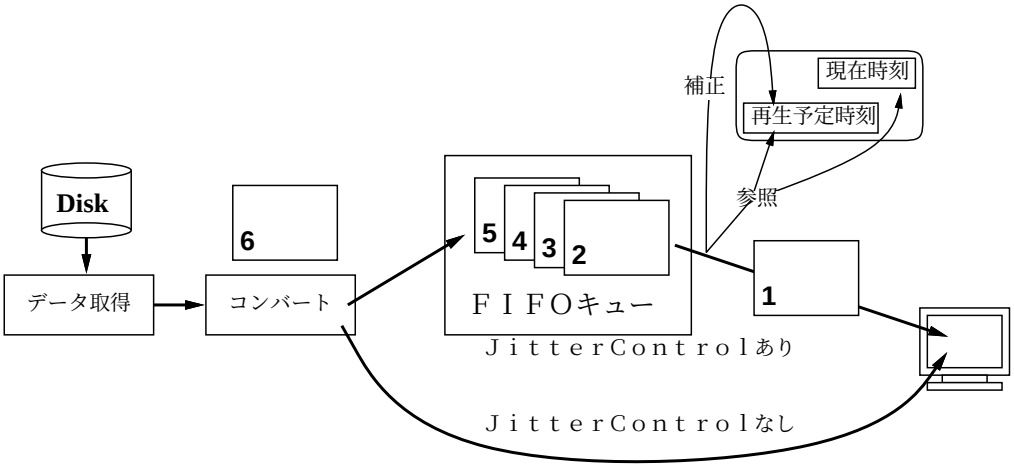

図 2.3: FIFO キューを利用したジッタコントロール

2.2.1 ジッタコントロール

ジッタの除去手法には、QtPlay[11][12] で提案された Real Time FIFO がある。これはデータを出力の直前で一旦 FIFO キューに保存し、一定の時間間隔で FIFO キューからデータを取り出して出力するものである。これを図 2.3に示す。

図 2.3では、まずディスクからデータを所得し、データを出力可能な形式にコンバートしている。このあとすぐに画面に出力するのではなく、一旦 FIFO キューにデータを格納する。これと並行して、一定時間毎に FIFO キューの先頭からデータを取り出し、画面に出力する。

例えば、図 2.3で FIFO キューが存在しなかった場合を考える。この場合、ディスクからデータを取得、データをコンバートし、すぐに画面に出力することになる。この時 6 番のフレームのデータコンバート処理で特に時間がかかったとする。もしそのまま画面に出力していたら、第 6 フレームの画面への出力は遅れることになってしまう。このような時間のずれがジッタである。ジッタを残したままデータを出力すると、出力画像の速度が不安定になってしまう。

このジッタを除去するのが図 2.3の FIFO キューである。メディアデータの再生予定時刻を設定しておき、メディアデータをディスクから取得、コンバートし、FIFO キューに保存する。FIFO キューの先頭では、別のスレッドが待機しており、フレームをキューの先頭から取り出し、その再生予定時刻を計算する。そして再生予定時刻と現在時刻の差をとり、その時間だけ待った後で画面に出力する。

図では第 1 フレームが取り出され、画面に出力されようとしている。それと並行して FIFO には 2〜5 のフレームが存在しており、更に後方で第 6 フレームのコンバートを行なっている。ここで第 6 フレームのコンバートに時間が掛かったとする。しかし、FIFO キューの中のデータが空になるまで、つまり第 2〜5 フレームの出力が終了するまでに第 6 フレームを挿入すればジッタは発生しない。

データコンバートなどの処理にひどく時間が掛かり、再生予定時刻を超過したデータがFIFOキューに到着した場合は、時間を待たずに出力すると同時に再生予定時刻を遅れさせるようにする。

このように再生予定時刻を遅れさせるということは、データ取得からの画面出力までのレーテンシを大きくするということである。レーテンシが大きくなればよりFIFOキューにデータが蓄積されるようになり、より大きなジッタを吸収できるようになる。

このような機構を利用することで一定時間毎にメディアデータを処理し、更にジッタを吸収できるようになる。

この手法はジッタを吸収できる程度の時間だけデータをFIFOキューで保存するため、読み込みから画面出力までの遅延がどうしても増加してしまう。しかしながら、ビデオプレーヤなどはこのようなレーテンシはあまり問題にならない。しかし、リアルタイム性が重要なシステムでは過剰な遅延は問題となる。例えば、ビデオ会議システムでレーテンシが異常に増大すると会話に支障をきたすことになる。このような場合は多少のジッタを許してでも、レーテンシを小さく保つほうがよい。

つまり、連続メディアアプリケーションの目的と、発生し得る最大のジッタ、FIFOを確保できる空きメモリ量のトレードオフでバッファ量や最大遅延時間を決定すべきである。

2.2.2 ストリーム間同期

異なったメディアデータ間で同期を取る為に、前節で挙げたジッタコントロールの機構を利用する。

この構成を図2.4に示す。

図では、音声ストリームと動画像ストリームの2つが存在し、どちらもジッタコントロールを行なう為、FIFOキューをストリームの中間に持つ。

メディア間の同期を保つにはこの再生予定時刻を同じ値に保てば良い。

このため、再生予定時刻を一括して管理する部分を新たに増設する。この部分で同期をとるジッタ除去機構の再生予定時刻が全て同じ値になるようにする。つまり、複数のストリーム間で再生予定時刻が同一になるように管理する。こうすることでメディアデータが出力される時間が同じになり、複数ストリーム間で同期した出力が得られるようになる。

2.3 CPU資源管理

2.3.1 リアルタイムOSの利用

第2.1.2節で説明したとおり、連続メディアアプリケーションはCPU資源を過剰に消費する可能性がある。このCPU資源の過剰消費を抑制する為に本研究ではリアルタイムOSが提供している機能を利用することを提案する。

本研究では、リアルタイムOSのRT-Mach上で連続メディアアプリケーションを実装する。RT-Mach

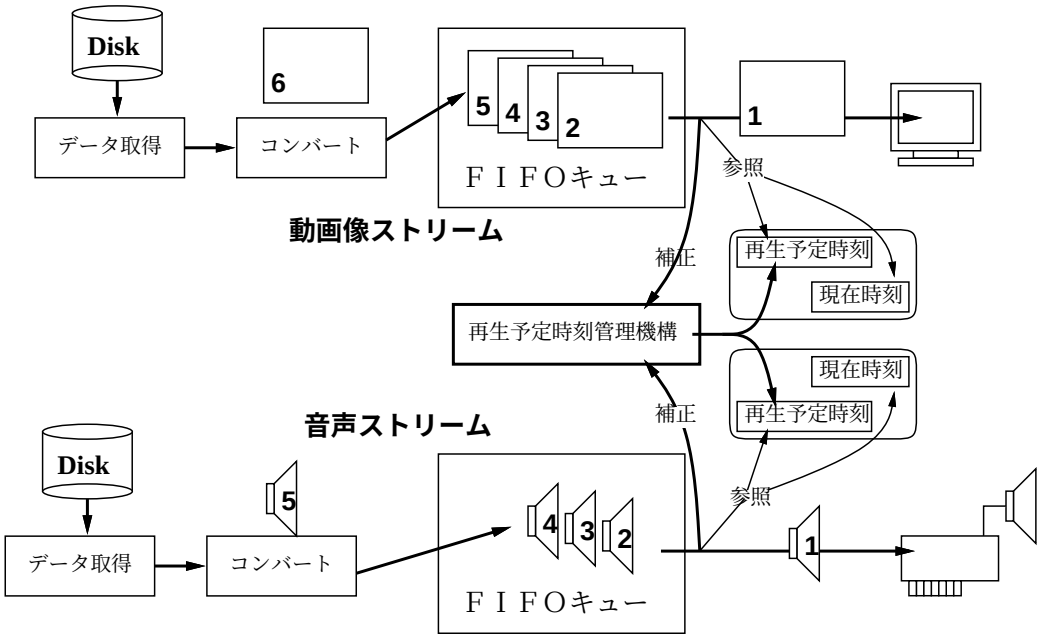

図 2.4: ストリーム間同期

にはCPU 資源管理機構があり、スレッド単位でCPU 資源の消費量を監視し、予約した量以上のCPU 資源を利用するとアプリケーションに対して通知するという設定が可能である。また、通知だけではなく、システム側が自動的に「処理の優先度を落す/落さない」「処理を強制的にサスペンドする/しない」といった設定も可能である。

これらの機能を利用してアプリケーションがCPU 資源を過剰に使用しているかどうかを知ることが出来る。また、現在のCPU 資源消費量などの情報も取得できる為、これらの値を利用してCPU 資源の過剰消費をなくすようにすることも可能である。

2.3.2 動的 QoS 制御機構

第 2.1.2節で説明したとおり、連続メディアアプリケーションはCPU 資源を過剰に消費する可能性がある。つまり並行して動作する他の重要なプロセスが使用すべきCPU 資源を奪ってしまうことがある。

このため、CPU 資源の過剰利用の通知を受ける為にリアルタイム OS を利用することを 2.3.1節で述べた。実際に過剰利用の通知を受けた場合、出力するメディアデータの品質そのものを低下させればよい。メディアデータの品質を低下させることでメディアデータの処理量が減少し、CPU 資源の消費量を減らすことができる。

これを動的QoS(Quality of Service) 制御と呼び、提供するサービスの質つまりここではメディアデータの品質を動的に制御するというものである。本研究ではこのような動的QoS 制御もサポートする。

例えば、ビデオプレーヤがフレームレート 30frame/sec のデータを再生しているとする。ビデオデータ

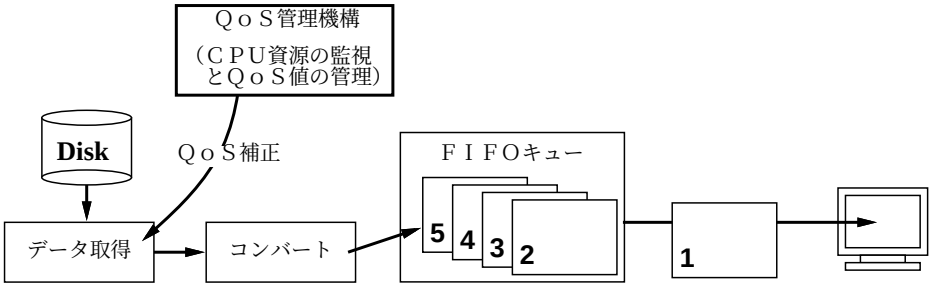

図 2.5: 動的 QoS 制御機構

の始めの方は画像展開が簡単で十分にその速度でも再生が出来ていたとする。しかし、ビデオデータの中盤にさしかかり、データ展開に必要な CPU 資源量が急激に上昇したような場合はそのままだと並行して動作する他の処理に影響を与えかねない。また、自分自身の実時間性を維持することすら難しくなってしまう。

このような場合はビデオプレーヤ自身が自動的に「提供するサービスの質を低下させる」ようにする。具体的には初期の 30frame/sec の品質を維持できないと自動的に認識してフレームレートを 20frame/sec に落すということが考えられる。このような処理で CPU 資源の過剰消費を抑えることが出来る。この処理するメディアデータを軽減することをメディアデータのスケーリングと呼ぶ。

ビデオプレーヤがどの程度の CPU 資源を消費するかは実行してみなければわからない上に、単一のビデオデータでも再生部分によって消費 CPU 資源量は大きく変わる。しかも他の処理の負荷も動的に発生する。このため、QoS 制御は動的に行なわなければならない。

この動的 QoS 制御を行なう為の構成を図 2.5 に示す。

メディアデータのストリームとは別に QoS を管理する部分を追加し、ここでメディアデータの QoS を一括管理する。

アプリケーションが CPU 資源を過剰使用しているという通知を受けると、管理している QoS 値の補正を判断して、ストリームにその旨を通知する。また、この部分で CPU 資源の消費量を常に監視すれば、単純に過剰使用の通知を受けてメディアデータのスケーリングを行なうだけでなく、アプリケーションの CPU 資源を常に監視しつつ、最適な QoS を維持するという処理も可能になる。

2.4 既存の連続メディアアプリケーションの問題点

以上であげた連続メディアデータ処理の問題を解決したのが本研究室で作成された QtPlay である。QtPlay はビデオプレーヤであり、リアルタイム OS の機能を利用してジッタコントロール、メディア間同期 (リップシンク)、動的 QoS 制御を実装した高機能ビデオプレーヤである。QuickTime 形式のムービーデータをサーバ (CRAS:Constant Rate Access Server) を介してディスクから読み込み、動画像、音声の両データに

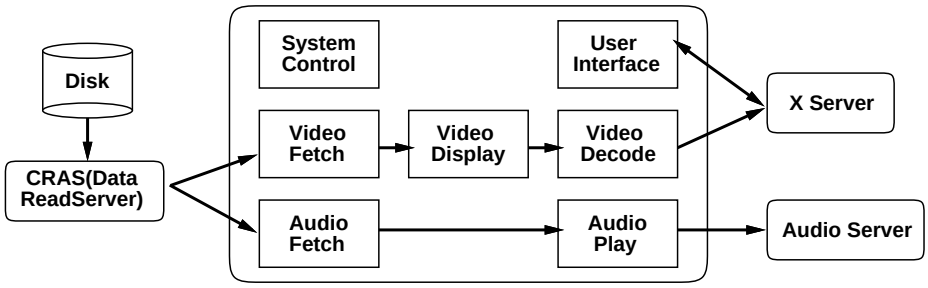

図 2.6: QtPlay の内部構成

分割、それを画面、ないしは音声サーバに出力するシステムである。QtPlay の構成を図 2.6に示す。

QtPlay は図 2.6で示すようにいくつかのコンポーネントに分割されている。各コンポーネントは以下のような役割を持っている。

ビデオフェッチ、オーディオフェッチ CRAS を介して動画データ、音声データを読み込む部分で、周期的に動作する。

ビデオデコード 動画像データを ZPixmap 形式に変換する。

ビデオディスプレイ、オーディオプレイ 動画像データを画面に出力、また音声データをサウンドボードから出力する。

ユーザインターフェース ムービーデータの出力を制御する為のユーザインターフェースをここで構築する。

システム制御 ムービーデータに合わせて各種入出力サーバを初期化したり、動的 QoS 制御などの自動制御を行なう。

しかしながらこのシステムは単純なモノリシックなシステム構成であり、拡張が難しいという欠点がある。QtPlay は新たな QoS 制御ポリシーを追加するような拡張や、これを利用してビデオコンファレンスシステムを構築するといった再利用性を考慮して作成されているわけではない。

既に述べたとおり、QtPlay のような連続メディアデータを処理するアプリケーションの構築は非常に難しく、作業量も膨大になる。このため、連続メディアアプリケーションの制作を支援する何らかのシステムが必要である。本研究では、拡張可能な連続メディアアプリケーションの構成を提案し、その構築を支援するツールキットを製作する。

このシステムを利用することで連続メディアアプリケーションの製作が容易になり、処理の時間管理や動的 QoS 制御をアプリケーションプログラマが直接記述しなくても連続メディアアプリケーションが構築できるようになる。

第 3 章

関連研究

本章では連続メディアアプリケーションの制作を支援するシステムをいくつか挙げ、その特徴と問題点を考察する。

3.1 ACME

ACME[3][4] は UCBerkley で作成された連続メディアアプリケーションの作成を支援するシステムである。

この研究では連続メディアデータを処理するソフトウェアの構成を提案して、更にこのシステムを実際に X Window System の拡張として実装している。ACME はメディアデータ処理機能付の X Window System と、そのプリミティブを利用する為のライブラリからなる。

ACME を利用すれば、アプリケーションプログラマは提供されているライブラリを利用してメディアデータ処理要求をサーバに通知するようなプログラムを作成するだけで連続メディアアプリケーションが構築できる。つまり連続メディアデータの加工や入出力、更にそれらの処理の時間管理機構を記述することなく連続メディアアプリケーションを作成できる。

ACME では入出力先の物理的デバイスをいくつかグルーピングした論理デバイスを定義しており、複数メディアのデータ処理を一括して処理する枠組を提供している。また LTS という論理時計を実装し、これらを利用してメディアデータの時間依存処理を行なっている。

ACME でビデオオンデマンドシステムを構築した時のシステム状況を図 3.1に示す。

ACME では X Window Server と平行して ACME Server が存在しており、この ACME Server でメディアデータの処理やその制御を行なう。

このシステムを利用することでアプリケーションプログラマは面倒なメディアデータ処理を直接記述することなく、ACME Server にメディアデータ処理要求を出すだけで連続メディアアプリケーションが作成できる。しかしながらこのようなサーバ型の実装では新規の物理デバイスを追加するといった機能拡張が

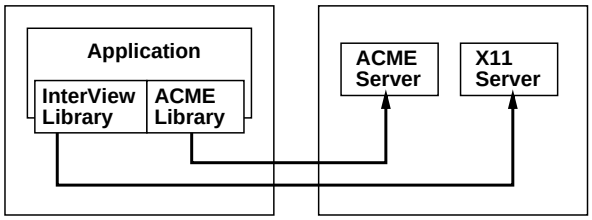

図 3.1: ACME の構成

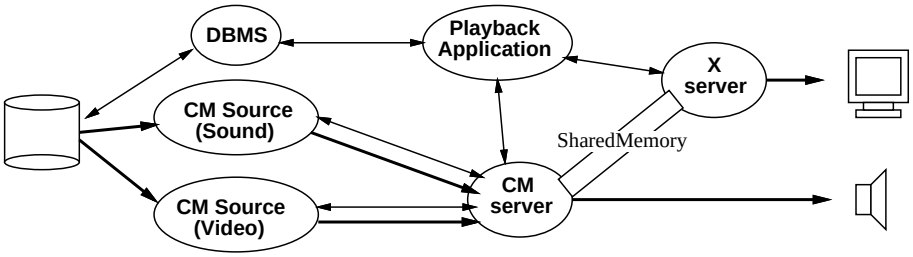

図 3.2: CMPlayer の構成

非常に難しく、拡張性が乏しいと言わざるをえない。

3.2 CMPlayer

CMPlayer[2] も UCBerkeley で作成されたシステムである。

CMPlayer は複数のメディアデータ間の同期処理や、メディアデータ処理状況に応じてフレームレートや画像データのレゾリューションを変化させてシステムの状況に積極的に対応する動的 QoS 制御をサポートしている。

CMPlayer の構成を図 3.2に示す。

図中の丸で囲まれた部分がサーバを示しており、太い矢印がメディアデータの流れを、細い矢印が制御命令の流れを示している。

CMPlayer では Playback Application 以外の部分を提供しており、アプリケーションプログラマは Playback Application だけ作成すればよい。この Playback Application では各サーバにの制御命令を要求する作業や、ユーザインターフェースの構築などを行う。

図中の CMSource サーバはメディアデータを二次記憶から取得するものである。更にそれを CM Server に渡す。CM Server では、メディアデータをコンバートするなどの処理と、メディア間同期やジッタコントロールといった時間管理を行い、X Server を介して画面に出力したり、音声デバイスを通してスピーカに出力する。

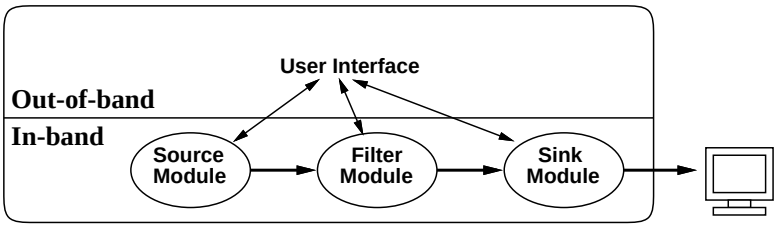

図 3.3: VuSystem の構成

さらに、CMPlayer ではアプリケーションの製作を容易にする為に CMToolkit と呼ばれるライブラリを提供しており、CMToolkit を利用して各サーバに処理要求を通知する。

この CMPlayer も ACME と同様にクライアント、サーバ型の実装になっている。このようなメディアデータを処理する部分を一か所に集中させる構成はは拡張性に乏しく、サーバが持つ機能以外に新機能を追加しようとすればサーバそのものを作り直さなければいけなくなる。

本研究ではこのようなクライアントサーバ型の実装は行なわず、後述する ToolKit の形式で実装を行なう。こうすることでサーバとの通信コストを減少出来る上にメディアデータ処理機能を柔軟に構築、拡張できるようになる。

3.3 VuSystem

VuSystem[1] は MIT の TNS で製作された連続メディアアプリケーションの作成を支援するツールキットである。

VuSystem の構成を図 3.3に示す。

VuSystem では、連続メディアアプリケーションを「In-band(メディアデータを処理する部分)」と「Out-of-band(In-band を利用する部分)」の 2 つに分割することを提案している。In-band は更に Module と呼ばれるメディアデータ処理を行う要素に分割されている。Module はメディアデータ処理を行う機能の単位であり、データを取得する Module、データをコンバートする Module、データを出力する Module というように様々な Module が提供されている。

Module は入出力のポートを持っており、ある Module の出力ポートと別の Module の入力ポートを接続することができる。このように複数の Module を接続することで、目的のメディアデータ処理を行う一連の Module 列が出来上がる。こうして Module 列を作成し、各 Module に対して処理開始の命令を与えれば、実際にメディアデータが Module 列を流れて目的の処理が行われる。

例えば、ディスクから動画像データを読み込む Module、データを出力可能な形式にコンバートする Module、動画像データを出力する Module の三つを接続する。そしてこれらモジュールに対してメディアデータ処理開始の命令を通知することでビデオデータをディスクから取得し、画面に出力するビデオプレー

ヤが出来上がる。またカメラから画像データを取得するモジュールを新たに作成して、それをディスクからデータを取得する Module と交換すればライブビデオを取得し出力するシステムができあがる。

Module は大きく分けて以下の 3 つに分類されている。

Source 入力デバイスからデータを取得する

Filter データの変換などをする

Sink データを出力する

Source Module は入力ポートは持たず、出力ポートを一つ持った Module である。これは Module 列の起点となるモジュールであり、ディスクからのデータの読み込みや、カメラからの画像の取得といった一連の Module を流れるメディアデータを作成する役割を持つ。

Sink Module は出力を行なう Module であり、出力ポートは持たず、入力ポートだけを持った Module である。この Module は前段の Module で作成、処理されたメディアデータを画面などに出力する役割を担っており、一連の Module 列の終点となる Module である。

Filter Module は入力ポート、出力ポートを各々1 つ以上持った Module である。この Module は一般に Module 列の中観点に配置され、データの変換を行なったり画像の変化を検出して Out-of-band に通知するなどの処理を行なう。

このような In-band の Module はあらかじめいくつかのものが定義されており、アプリケーションプログラマは既に用意された Module を利用することで簡単に連続メディアアプリケーションが構築できる。また Module は全て共通の形式に沿って構築されており、既存の Module では目的のメディアデータ処理機能が実現できない場合は、この共通の形式に沿って新たな Module を製作し、システムに追加すれば良い。

アプリケーションプログラマが実際にコードを書く部分は Out-of-band であり、ここではユーザインターフェースの処理を列記する。例えばユーザインターフェースを構築するツールを使い、メディアデータ処理開始ボタンなどを配置する。このボタンのハンドラで In-band(Module) に対してメディアデータ処理開始命令を送る処理を書く。

このようにして連続メディアアプリケーションが製作できるが、Out-of-band にはスクリプト言語などを利用したより高度なアブストラクションを提供するシステムを記述することも出来る。

以上のように、VuSystem はツールキットの形で実装し、更にメディアデータ処理を Module という処理要素に分割したことで大きな拡張性を得ることが出来たといえる。

しかしながら、複数のメディアデータ間で同期を取るなどの処理を実装しようとすると、複数の Module 間で同期処理を行わなければいけない。VuSystem の In-band はこのような Module 間で関連を持つ処理はサポートしておらず、結局 Out-of-band で作業しなければならない。また、それだけでなく動的 QoS 制御のような高度な制御も全て Out-of-band で実装しなければならない。

VuSystem はメディアデータ処理のアブストラクションがあり、データ処理機能の拡張も容易である。しかし、メディアデータの一連の流れとその制御のアブストラクションが十分ではない。

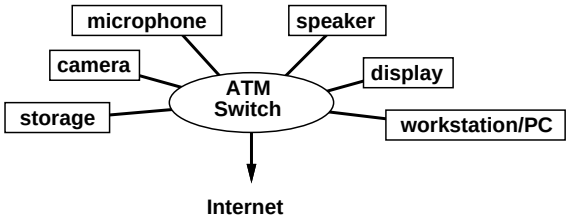

図 3.4: Medusa の構成

3.4 Medusa

Medusa[5] は Olivetty、ロンドン大、ケンブリッジ大で研究されている分散マルチメディアシステムである。

Medusa の構成を図 3.4に示す。

Medusa は多様なメディアデータを分散環境下で扱うことができるシステムである。

各種データを統一的に扱う、複数のストリームを同時に扱う、セキュリティを確保する、インタラクティブな操作感を提供するということに主眼を置いており、高度なマルチメディア環境を形成している。

分散環境下での通信はクライアントサーバではなく Peer To Peer の構成を持っており、メディアデータをローカルのデータとリモートのデータを全く同じようにユーザに見せている。

このように Medusa では高度な連続メディアアプリケーション環境を提供している。しかし、Pandora[6] の研究から生まれた Medusa はメディアデータ処理のかなりの部分をハードウェアに頼っており、本研究のようなメディアデータの処理構造に主眼を置いた研究ではなく、マルチメディア環境でのプログラミングについての研究である。つまり、Medusa は連続メディアアプリケーションの構成やメディアデータの処理方法を扱っている本研究とは基本的な部分で異なっている。しかしながら、Medusa の研究で提案しているプログラミング環境は重視すべきである。

3.5 CINEMA

CINEMA[7][8] も VuSystem と同様にツールキットで実装した連続メディアアプリケーションの製作を支援するシステムである。CINEMA の概要を図 3.5に示す。

VuSystem ではメディアデータ処理の各種機能を Module に細分化し、それを組み合わせることで目的のメディアデータ処理を形成することを提案していた。しかしながらこの方式では複数の Module 間の連係が必要なメディアデータ間同期処理などのアブストラクションはユーザが作成する Out-of-band で実装しなければならなくなる。

CINEMA は特にメディア間同期処理のための枠組を中心にこのような問題を解決している。図 3.5のよう

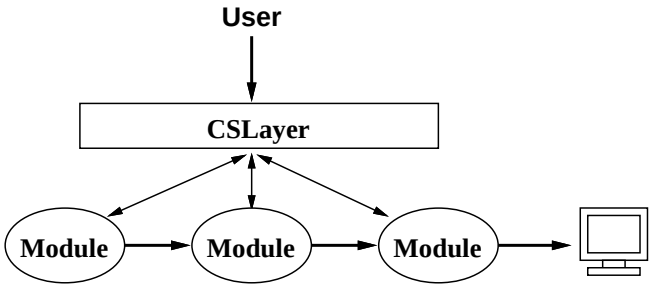

図 3.5: CINEMA の構成

に VuSystem と同様に複数の処理要素を定義し、それらの間の連係動作を行なうために CS-Layer(Control and Synchronization Layer) というものを定義している。このレイヤでは同一メディアストリーム内、あるいは異なったメディアストリーム間での Module 同士の同期の処理を行なっている。

つまり VuSystem の In-band と Out-of-band の処理の中簡に制御のためのコンポーネントを配置し、そこで同期処理を行なっている。この構成を利用することで Out-of-band に対してストリーム間同期のアブストラクションを設定することが出来る。更に、メディア間で同期を取る為の構造として media clock と呼ばれる実時間-メディアデータ時間変換対を各ストリーム毎に用意して、これを利用してメディア間の同期を行なうことを提案している。

このように、CINEMA では複数の Module 間の同期管理を行う為に CS-Layer という部分を追加している。しかしながら、CINEMA ではシステムの負荷を検出したり、メディアデータの処理状況に応じた動的 QoS 制御を行なうことはしていない。

3.6 問題点のまとめと本研究での対処

関連研究の特徴と問題点を以下にまとめる。

ACME,CMPlayer

クライアントサーバ型の実装であり、CMPlayer は動的 QoS 制御を実装している。また複数メディア間の同期処理もサポートしている。しかし、メディアデータ処理がサーバ内でモノリシックに実装されているため、拡張性に欠けている。

VuSystem

VuSystem はメディアアプリケーションの製作を支援するツールキットである。面倒なメディアデータ処理は In-band と呼ばれる部分に押し込めており、ツールキット側で最初から提供している。連続メディアアプリケーションを作成する時は In-band を利用する Out-of-band の部分を作成するだけで良い。In-band は更に Module と呼ばれる複数の機能単位に分割をしており、これらの組み合わせ

て目的の機能を実装する他に、既定の形式で新たに Module を製作することで新たな機能が実現できる。しかしながらこの複数の Module を統括する部分は持っていない。このため複数の Module が関連を持ち、それらを統括して管理したい場合、管理部分は Out-of-band で実装しなければいけない。

CINEMA

CINEMA は VuSystem の構成に CS-Layer という部分を追加し、各 Module 間での同期処理を行なうことを提案している。しかしながら、動的 QoS 制御やシステムの負荷の監視などの作業は実装されていない。

以上の事柄をまとめて、本研究で考慮しなければいけないことを以下に列挙する。

- 拡張性が高いツールキットの形で連続メディアアプリケーションの作成を支援するシステムを製作する
- 複数のメディアデータ間で同期をとることができるようにする
- 動的 QoS 制御をサポートする
- システムの負荷を監視して、それに応じた処理を行う

第 4 章

ツールキットの設計

4.1 現状の問題点と新設計の目標

これまでに提案されている連続メディアアプリケーション構築支援システムはメディアデータを処理する部分のアブストラクションは存在するが、メディアデータ処理を制御する部分のアブストラクションは充分であるとはいえない。

ストリーム間同期や動的 QoS 制御は異なった複数のメディアデータ処理を統合して処理するものである。つまりストリーム間同期や動的 QoS 制御を行なう為には複数のモジュールを一括管理するような機構が必要である。

VuSystem の実装では、細分化された Module のメソッドをアプリケーションプログラマが直接制御する形式をとっており、モジュールを一括管理するようなアブストラクションがない。この設計だと、「どのモジュールがどの命令を必要としているか」をアプリケーションプログラマが逐一記述しなければならない。つまり、ひとたびユーザからメディアデータ処理開始の要求があれば、その要求を満たす為に、たくさんある Module の中から適切なモジュールに適切な命令を通知しなければならない。

更に、ストリーム間の同期機構や動的 QoS 制御機構と言った全システムを統括して制御する機構は In-band ではできないため、結局 Out-of-band でアプリケーションプログラマが実装しなければいけない。このためアプリケーションプログラマの作業量が多く問題を起こしやすい。

たとえば、動的 QoS 制御を実装では、どの様な情報を制御のトリガにするか、データの品質を低下させるタイミングはどうするか、データのどの品質を低下させるかなどを考慮して実装しなければならない。これらの作業は複雑であり、アプリケーションプログラマがこれらの処理を記述しなくてすむような機構が必要である。また、実装するアプリケーションの特性に応じて動的 QoS 制御のポリシーも変更する可能性があり、QoS 制御のポリシーは自由に変更できるような構造でなければならない。

以上の事柄を考慮して本研究では連続メディアアプリケーションを以下の三つに分割することを提案する。

In-band メディアデータを処理する部分

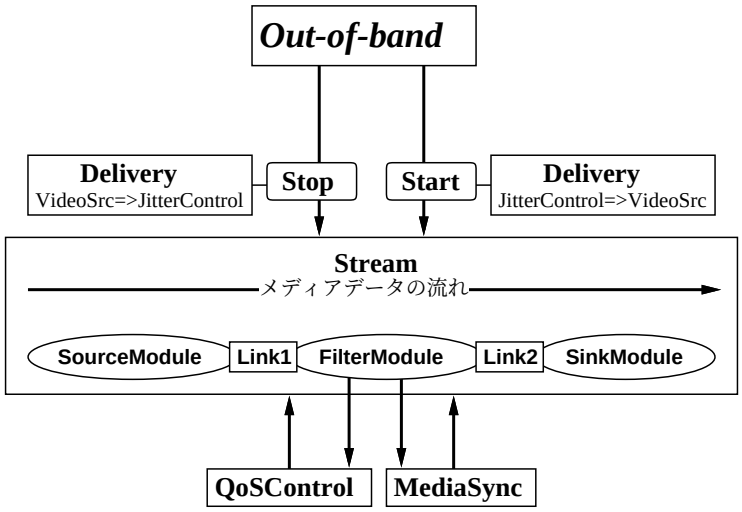

図 4.1: 連続メディアアプリケーション例

StreamManager メディアデータの処理を制御する部分

Out-of-band ユーザインターフェース等の処理でアプリケーションプログラマが記述する部分

VuSystem の構成では In-band と Out-of-band の 2 つだけだったが、本研究ではその中間に StreamManager という部分を追加する。StreamManager は Out-of-band に対して In-band の Module 列を抽象化する役割を担う。さらに複数の Module を統括した制御ができるような機構も提供する。

4.2 連続メディアデータ処理システムの構成

本研究では、連続メディアデータの処理をするために図 4.1 ようなシステムの構成を提案する。

まず、メディアデータ処理を Module と呼ばれるオブジェクトに細分化する。いくつかの Module を連続して並べ、それらの間をメディアデータが流れるように接続することで希望のメディアデータ処理を行う In-band 部分を実現する。図 4.1 で楕円で示しているのが Module である。Module はメディアデータ処理の開始点となる「SourceModule」、手前の Module からメディアデータを受け取り、処理をして次の Module に渡す「FilterModule」、そして一連のデータ処理の終端となる「SinkModule」の 3 種類がある。

Module 間でデータの受渡しをさせるために Module 同士を接続する。Module 同士の接続には Link オブジェクトを利用する。つまり Module から Module へのデータの受渡しは Link を介して行なう。

このように Module と Link を利用してメディアデータを処理する機構を構築する。例えば、ビデオデータをディスクから取得するモジュール、ビデオデータをコンバートするモジュール、ビデオデータを画面に出力するモジュールを作成して、各々Link で接続すればビデオプレーヤができあがる。

メディアデータ処理機能を Module という枠に押し込め、分割して管理することで、目的のメディアデー

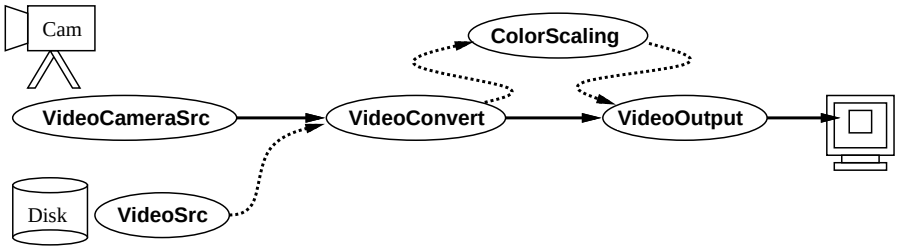

図 4.2: モジュールの組み換えによる機能切替えの例

タ処理機能を柔軟に構築することができるだけでなく、各種機能を動的に切替えることで様々な機能が実現できる。

この様子を図 4.2に示す。

たとえば、ビデオコンファレンスシステムのカメラから動画像データを取ってくる Module と、ビデオプレーヤのディスクからデータを取ってくる Module を切替えることで会議中にビデオデータを差し挟むことができる。また、同じくビデオコンファレンスシステムで、通常はカラー映像を送るが、回線状況が悪く、大容量のデータを遅れない時はカラー画像から白黒画像に変換する Module を挟んで白黒画像に変換してから送信するなどの拡張も可能である。

ここまでは、メディアデータを処理するための機構である。連続メディアアプリケーションでは、メディアデータ処理機能だけでなくメディアデータ処理を制御する機構も必要である。

まず、メディアデータの流れを定義する。Module 同士を Link で接続することで、SourceModule から SinkModule まで一連のメディアデータの流れが出来上がる。このメディアデータの流れを Stream と定義する。そして、この Stream に対し各種命令を送ってメディアデータ処理を制御するものとする。つまり、メディアデータ処理の制御命令は Stream に対して通知するものとする。本論文では以降この Stream に対するメディアデータ処理命令を Command と呼ぶ。

例えば Stream1 のデータの処理を開始したい時は Stream1 に対してデータ処理開始の Command を通知すれば良く、またデータの処理を停止させたい時は Stream1 に対してデータ処理停止の Command を通知すれば良い。

なお、Command を実行してもそのストリームに属する全てのモジュールに通知されるわけではない。Stream が自分に対する Command を受けると、Command に付属されている Delivery オブジェクトを利用して、適宜必要なオブジェクトに適切な順番で配送するようになっている。つまり命令配送は Delivery に行なってもらい、Command は Module のメソッド呼び出しの作業のみを行なう。こうすることで Command の配送ポリシーを柔軟に設定できるようになる。

また、動的 QoS 制御を実現する為に、QoSControl オブジェクトを定義する。Stream に対して動的 QoS 制御命令が下るとその Stream が持っている QoSControl オブジェクトがどの Stream のデータ品質をどの

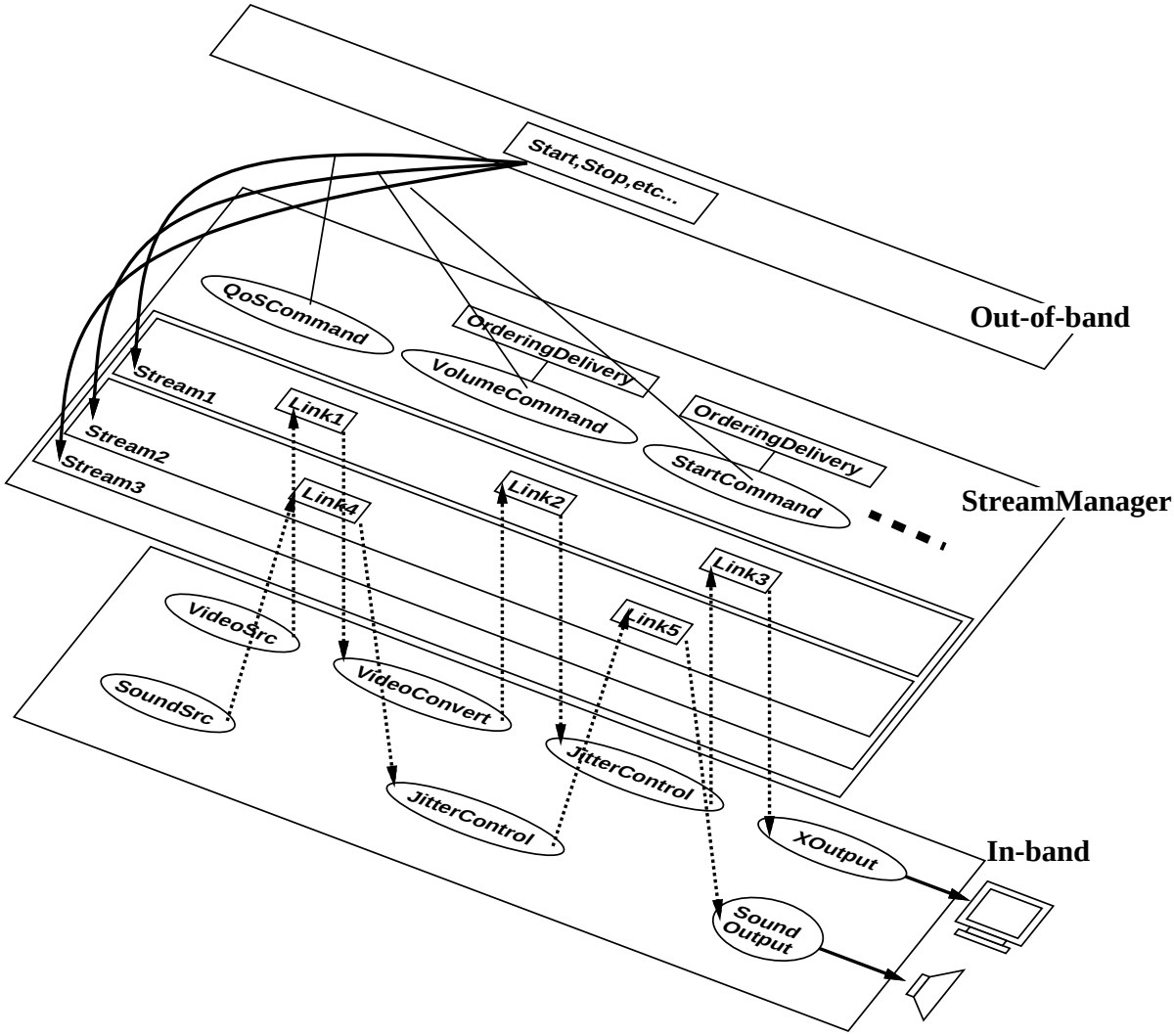

図 4.3: 連続メディアデータ処理システムの構成

様に変化させるかを算出し、Stream に対して動的 QoS 制御を行なう。

メディア間の同期処理を行なう為に、MediaSync オブジェクトを定義する。Stream に対して同期をとる命令を送るとその Stream が持っている MediaSync オブジェクトが適宜同期処理を行なう。

これら二つの命令はその他の命令とは異なり、Stream 全てを統括する処理を実行する。

4.3 連続メディアデータ処理構造の作成と制御

本研究で提案する連続メディアデータ処理システムの構成を図 4.3に示す。

本システムを利用して連続メディアデータアプリケーションを製作するアプリケーションプログラマは基本的に図 4.3内の Out-of-band と呼ばれる部分のプログラミングを行なう。Out-of-band でアプリケー

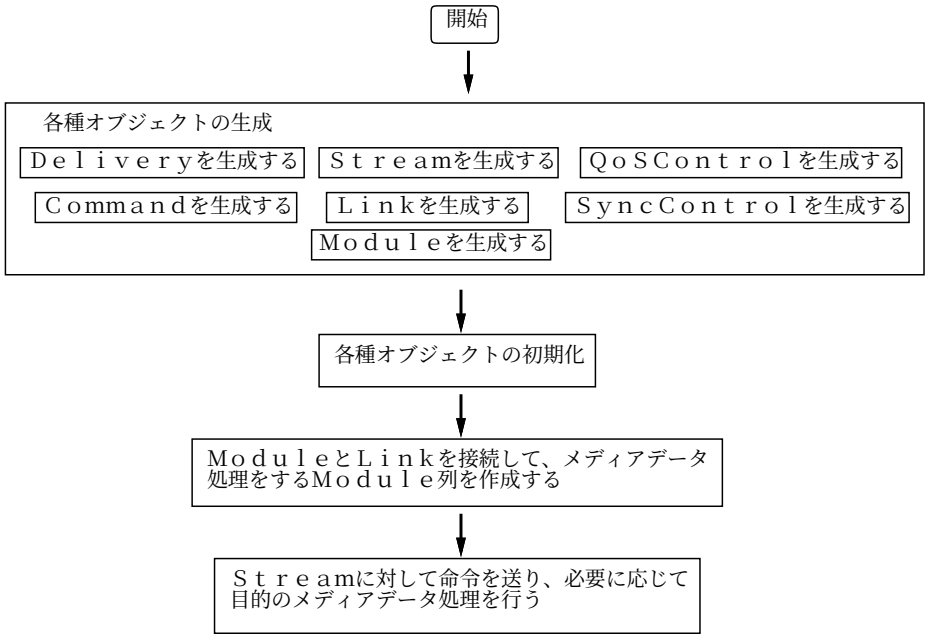

図 4.4: 連続メディアデータ処理システムの作成手順

ションプログラマが記述しなければ行けない処理を以下に列記する。

- 本ツールキットが提供している各種 Module を生成し、それらを組み合わせてメディアデータ処理構造を構築する。
- メディアデータ処理構造に対して「処理開始」や「処理停止」などといったメディアデータ処理の命令を通知する。
- ユーザインターフェースの構築などの処理

本節ではこのメディアデータ処理構造の作成、メディアデータ処理の命令について説明する。
Out-of-band でメディアデータ処理構造を作成し、動作させるまでの手順を図 4.4に示す。

4.3.1 メディアデータ処理構造の構築

アプリケーションプログラマはまず、アプリケーションの構成要素である各種オブジェクトを作成しなければならない。
図 4.4で示すシステムは以下のオブジェクトで構築できる。

Module メディアデータ処理を行なうオブジェクト

Link Module 同士を接続するオブジェクト

Stream メディアデータの流れを管理するオブジェクト

Command Stream に対して処理制御命令を行なうオブジェクト

Delivery Command をモジュールに配送するオブジェクト

QoSControl 動的 QoS 制御を行なうオブジェクト

SyncControl Stream 間の同期を行なうオブジェクト

メディアデータ処理構造を作成する為にこれらのオブジェクトを必要に応じて生成する。

オブジェクトの生成後にそのオブジェクトの初期化なども行なっておく必要がある。たとえばビデオデータを画面に出力するオブジェクトは画面に表示する時の縦横幅などの情報が必要であるし、音声データを取得するオブジェクトは音声データのサンプリングレートなど扱う音声データについて知らなければならない。Delivery オブジェクトは Command の配送ポリシーを定めた物だが、この配送ポリシーによっては命令を配送する順序をユーザが直接設定する場合も考えられる。これらはの設定はオブジェクトのアトリビュートに対する操作で実現する。

オブジェクトの初期化を行なった後、Module 同士を接続する。本システムではメディアデータ処理機能を複数の Module オブジェクトに細分化して管理しており、Module を複数接続させて目的のメディアデータ処理を行なうような Module 列を作成する。この Module 同士の接続には Link オブジェクトを利用する。たとえば、Module A と Module B の接続は、Module A の出力ポートと Link の入力側を接続し、さらにその Link の出力側と Module B の入力ポートと接続する。

このようにして一連の Module 列を作成する。メディアデータはこの Module 列を流れることになる。本システムでは、メディアデータの流れを管理する Stream オブジェクトを定義しており、Module 同士を接続すれば、自動的に接続された Module が Stream に属するようになる。

本システムでは Link は必ず何らかの Stream に属する物として扱う。そして、Link を Module に接続すると、接続された Module は Link が属する Stream の配下に入る。こうすることで、Module 同士を接続するだけで Module は自動的に Stream の配下に入ることになる。

以上をまとめると、以下のようになる。

1. Module 同士を接続することで目的のメディアデータ処理機能を実現する。
2. さらに接続された Module は特定の Stream の配下に入る。

この様子を図 4.5に示す。

このようにしてメディアデータを処理する Module 列を作成する。

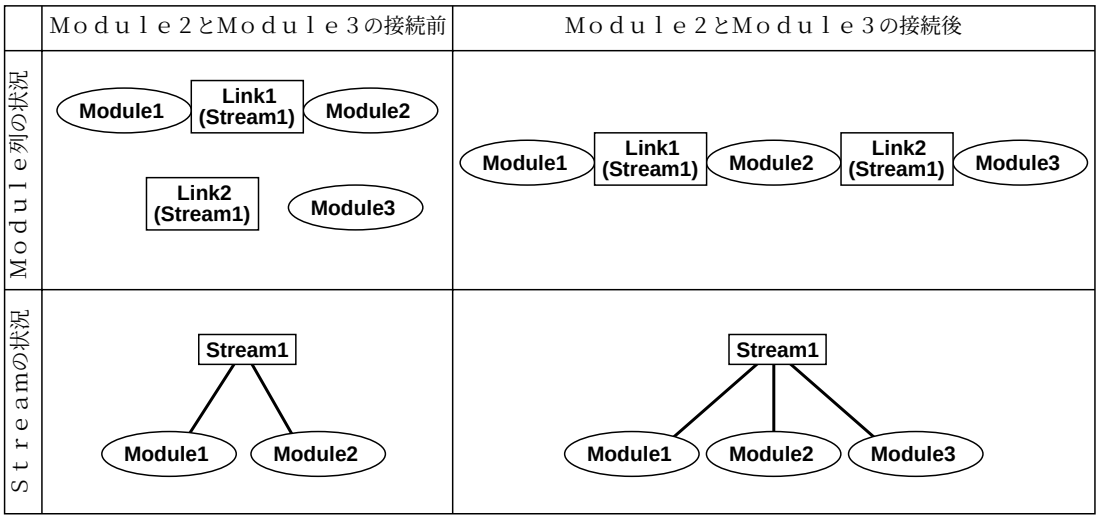

図 4.5: モジュール間の接続の様子

4.3.2 メディアデータ処理の制御命令の通知

メディアデータ処理構造を構築すれば、あとはその Stream に対して処理開始命令を通知することでメディアデータ処理を実行できる。命令の通知には Command オブジェクトを利用する。

アプリケーションプログラマが Stream に命令を通知するには、Stream が提供している命令呼び出しのメソッドを呼ぶ。この時、呼び出す Command オブジェクトを与えておく。後は Stream の命令呼び出しメソッドが、Command に付随している Delivery オブジェクトを呼び出して、配下の Module の中から目的のメソッドを適切な順番で選択する。そして Command オブジェクトがそのメソッドを呼び出す。こうすることで Module に目的の命令が通知される。

このように Command の配送を Command や Stream から分割し、Delivery オブジェクトとして独立させることで、命令の配送ポリシーが拡張可能になる。命令の配送順序を制御する必要について図 4.1 を例にして考えてみる。

図 4.1 のシステムには SourceModule, FilterModule, SinkModule の三つの Module がある。これら全ての Module が処理開始命令 (以下 Start) と処理停止命令 (Stop) の二つを必要としているとする。このようなシステムでは連続的にメディアデータが流れる為、Start 命令は出力側を先に処理中の状態にしてから入力側を処理中の状態にしたい。もし入力側に先に処理開始命令が到着するとメディアデータはすぐに流れだす。しかし出力側の Module に処理開始命令は到着していないと、その間ずっとメディアデータの流れが停止してしまうか、あるいは捨てられてしまうことになる。これとは逆に Stop 命令は入力側の Module から順に停止命令を通知したい。出力側から Stop 命令が配送されると、出力側の Module は既に停止しているのに入力側は動き続けるような状態が発生する。また、これらのポリシーとは違って「ソース側から配送する」とか「シンク側から配送」するというだけでなく、ユーザがモジュール間の配送順序を適宜半

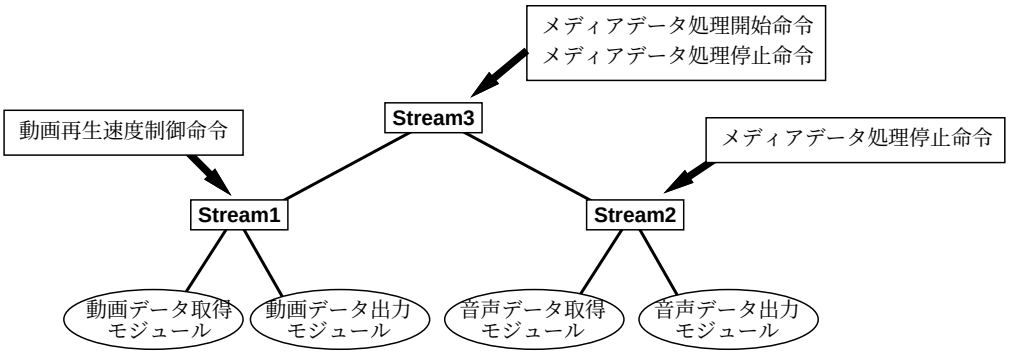

図 4.6: ツリー構造を持つ Stream の様子

順序で指定したい場合も考えられる。

このような様々なポリシーを Delivery オブジェクトのサブクラスに定義しておき、その中から適切なポリシーを持つ Delivery オブジェクトをインスタンシエーションして Command に関連付けておく。

なお、Stream はツリーの形式で管理するようになっている。

複数のメディアデータを処理する場合はあらかじめ別々の Stream を定義しておき、ことなったメディアデータを処理するモジュールは異なった Stream の配下に入るようにしておく。そして、これらの Stream をグルーピングして、親の Stream を作成する。

この例を図 4.6に示す。

図 4.6では音声と動画像の 2 種類のメディアデータを処理するシステムを示している。動画像データを取得、出力するモジュールは Stream1 の配下に、音声データを取得、出力するモジュールは Stream2 の配下にいる。また、これらをまとめた Stream3 が存在する。

このシステムでビデオデータを再生したい場合は Stream1,2 の両方の親である Stream3 に再生開始命令を送ればよい。こうすることで再生開始命令は Stream のツリーを辿って配下にいる 4 つのモジュール全てに再生開始命令が到着する。

ここで、ビデオデータを早送りしたいとする。早送りは「動画像データを通常の 2 倍の速度で再生」して「音声データの出力は行なわない」とする。このような場合は、音声データの Stream2 だけ停止命令を送信して処理を停止しておき、動画像データの Stream1 には再生速度変更命令を送る。このように Stream は命令配送のスコープとして利用できる。

本システムは通常のデータ処理制御命令 (Command) の他にメディア間同期や動的 QoS 制御もサポートする。メディア間の同期、動的 QoS 制御は Stream に専用の呼び出しメソッドを用意しており、これを呼び出すと Stream が保持しているメディア間同期、動的 QoS 制御を行なうオブジェクトに命令が渡る。これらの制御でも Stream を辿りつつ Module のメソッドを呼び出すのだが、この特殊な制御命令は各 Stream に専用のデータバッファを持っており、Stream 単位で制御情報を管理している。

本システムではこのメディア間同期を司るオブジェクトを SyncControl と呼び、動的 QoS 制御を行うオブジェクトを QoSControl オブジェクトと呼ぶ。これらの内部ではメディア間同期や動的 QoS 制御のために Module に与える各種パラメータを算出しており、Stream 毎に用意されたバッファエリアはこのパラメータ算出に利用する。パラメータを算出するとその値に応じてその Stream の配下の Stream に制御が渡るか、配下の Module に QoS 制御要求や同期要求が通知される。

4.3.3 コールバックの実現

本システムでは Module 内部でメディアデータを処理する設計になっており、メディアデータを処理する Module とユーザインターフェースなどアプリケーションプログラマが作成する部分とは完全に分離されている。

Out-of-band から In-band に対してメディアデータ処理の制御や、各種 Module にパラメータを通知する手段として、Command オブジェクトやオブジェクトの SetAttribute メソッドが提供されている。この機構はメディアデータ処理の開始、停止命令や Module が処理するデータの詳細を指定する時に利用できる。

しかしながら、この Out-of-band から In-band への通知とは逆に、In-band から Out-of-band に対して通知を行なう必要がある。実際には「ビデオプレーヤで、現在何秒後のデータを再生しているかを通知する」とか、「ビデオカメラによる監視システムで、撮影している映像になんらかの変化が起こったらその旨を通知する」が考えられる。これらはどれも In-band から Out-of-band への通知である。本システムではこのような In-band から Out-of-band、ないしは In-band から StreamManager への通知をコールバックと呼んでいる。

コールバックは Module に対して Out-of-band が提供している関数のポインタを SetAttribute で通知しておき、必要に応じて Module がこの関数を呼び出す。このようにして Module から各種通知を受けられるようにする。このような構成にすることで、Module ごとにコールバック関数の引数などを自由に設定でき、コールバックは関数呼び出しで実装されるので引数を自由に設定できる上に情報を高速に通知することができる。

第 5 章

ツールキットの実装

5.1 実装環境

実装は、Real-Time Mach[15] 上で C 言語を使用して実装する。

RT-Mach はカーネギーメロン大学で開発された Mach3.0 にリアルタイム拡張を施した OS である。この OS では多彩なリアルタイムプリミティブを提供しているが、本研究の実装では特にリアルタイムスレッド、リアルタイムスケジューラ、CPU 資源予約機構、リアルタイム IPC、リアルタイム同期プリミティブを利用している。

リアルタイム OS 上で連続メディアアプリケーションを構築する利点を以下に列挙する。

- 連続メディアアプリケーションでは単位メディアデータを高速に処理する必要がある。RT-Mach はマイクロカーネルであり、OS 内のプリエンプト出来ない部分を走行する時間が短いためメディアデータ処理のような短い時間間隔で高速にアクティベートするような処理に向いている。
- リアルタイムスレッドを提供している為、並行した Stream を形成するなどの処理を構築しやすい。また、そのための同期プリミティブや通信プリミティブも優先度逆転問題の回避を行っている。
- CPU 資源予約の機能を持っており、連続メディアアプリケーションが CPU 資源を過剰使用した時にはユーザ側にその旨が通知されるなどの処理が実装可能である。

5.2 本システムで提供するオブジェクト

前章で述べたメディアデータ処理オブジェクトの詳細を本節で示す。

5.2.1 概要

本システムで提供するクラスうち、基本的なものを図 5.1に示す。

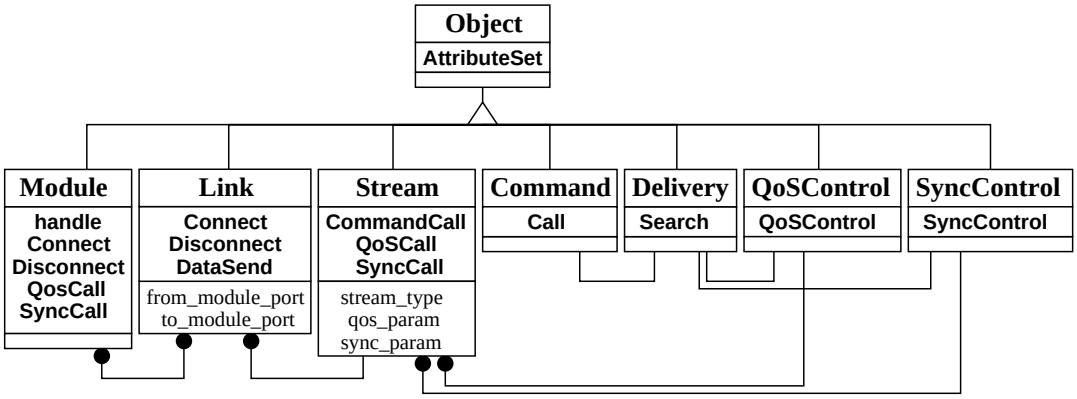

図 5.1: システムが提供している最上位層のクラス

本システムではこれらのクラスかまたはそのサブクラスのインスタンスをユーザが生成して利用する。

Object

本ツールキットで提供するあらゆるオブジェクトの親となるものであり、アトリビュート操作用メソッドを持つ。

Module

連続メディアデータ処理の単位であり、実際に連続メディアデータを処理する物である。メディアデータの処理を細分化した機能を持ったクラスを Module クラスのサブクラスとして定義しており、ビデオデータをディスクから取得する Video Cras Input Module、ビデオデータをコンバートする Video Convert Module、ジッタを除去する Jitter Control Module、ビデオデータを画面に出力する Video X Output Module などがある。

Link

メディアデータ処理を細分化した Module 同志を接続することで目的のメディアデータ処理機能を実現する。この Module 同志の接続を行うのが Link オブジェクトである。Module との接続を司る connect,disconnect の他に、Module からデータを受け取り、接続先の Module にデータを渡す dataSend などのメソッドを持つ。

Stream

Module 同士を接続することで、一連のデータの流れができる。このメディアデータの流れをストリームと呼び、この Stream オブジェクトでメディアデータの流れを管理する。メディアデータ処理の開始や終了といった命令は「メディアデータの流れ」に対する制御であり、命令は全てこの Stream に対して通知する。Link は必ず一つの Stream と関連を持っており、Module 同士を Link で接続することで接続された Module が自動的にその Stream に属する。

Command

Command はメディアデータの流れである Stream に対する制御命令である。Command の生成時には Command の配送ポリシーを持った Delivery のインスタンスが必要である。メディアデータ処理の開始や停止はこの Command のサブクラスとして定義する。

Delivery

Stream 内には複数の Module が存在する。ある Stream に対して Command を呼び出すと、最終的にはその Stream に属する Module のメソッドが呼ばれる。そのメソッドを呼び出すポリシーを定義しているのが Delivery である。実際には Delivery のサブクラスを定義し、サブクラス側で Delivery のポリシーを記述する。

SyncContorl

Stream 間同期を実装する為のオブジェクトである。Stream は各個に SyncControl のオブジェクトを保持し、Stream に対して同期要求がある度にこの SyncControl オブジェクトで同期処理を行なう。

QoSContorl

本システムは汎用的な連続メディアツールキットを目指すため、動的 QoS 制御のポリシーも拡張可能な形で管理できなければ行けない。例えば、音声の QoS 制御と動画の QoS 制御は全く異なる。更に、同一の動画 Stream でも、Stream 間でプライオリティを設定し、プライオリティに応じて QoS スケーリングを行なわなければならない場合も考えられる。このポリシーの可換性を実現する為に QoSControl クラスが存在する。様々な QoS 制御ポリシーはこの QoSControl クラスのサブクラスという形で提供する。

5.2.2 Stream

Stream オブジェクトの構成を図 5.2に示す。

Stream は図 5.2(a) で示すようにツリーの構造を持っている。Stream にはツリーのリーフにあたる Stream とノードにあたる Stream の 2 種類がある。リーフの Stream は自分の配下にいる Module と関連を持っており、ノードの Stream は自分の配下の Stream への関連を持っている。

つまり、本システムでは Stream を子として持つ Stream と、Module を子として持つ Stream の 2 種類が必要になる。この構成を示したのが図 5.2(b) である。このクラスが提供している各メソッドの役割を表 5.1に示す。

5.2.3 Link

Link オブジェクトは、基本的に図 5.1で提供している Link クラスをそのまま継承したクラスを利用して生成する。

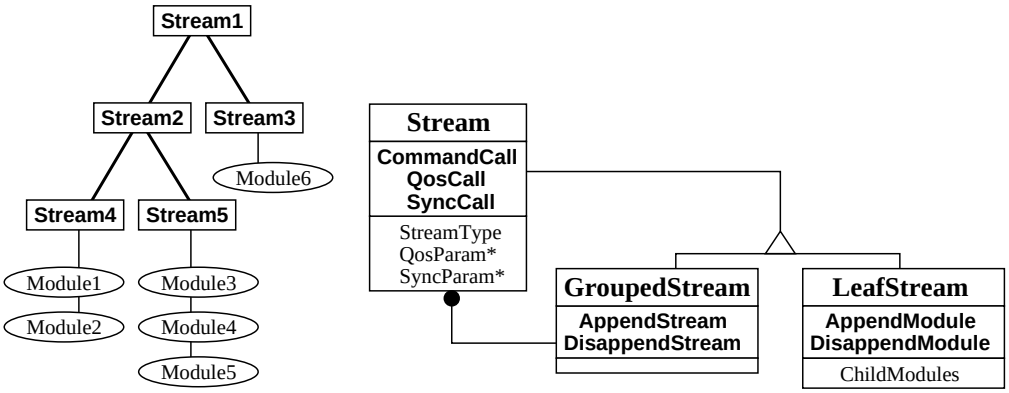

図 5.2: Stream クラス

表 5.1: Stream クラスが提供するメソッド

メソッド名	機能
`command_call`	Stream に対して命令を呼び出す。ただし、GroupedStream と LeafStream では処理内容が異なる。
`QosCall`	動的 QoS 制御命令を呼び出す。
`SyncCall`	メディア間同期命令を呼び出す。
`AppendStream`	属性 (child_streams) に Stream へのポインタを追加する
`DisappendStream`	属性 (child_streams) から Stream へのポインタを削除する
`AppendModule`	属性 (child_modules) に Module へのポインタを追加する
`DisappendModule`	属性 (child_modules) から Module へのポインタを削除する

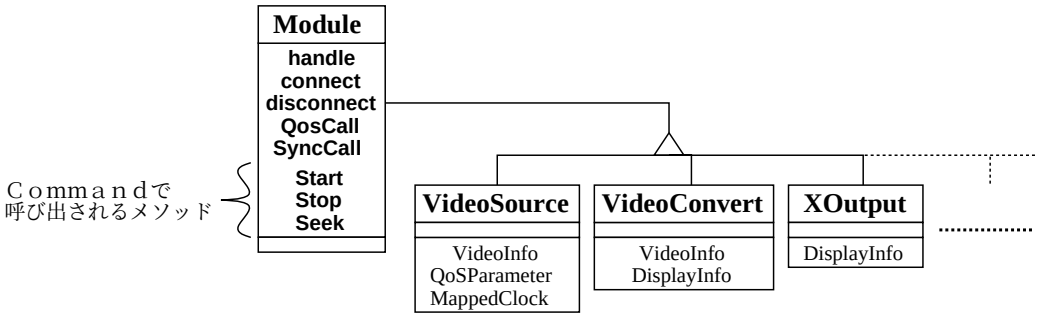

図 5.3: Module クラス

本システムでは Link は Module 同士を接続する為の仲介役を果たす。また、Module を Link に接続させるということは Module を Stream に属させることにもなる。

5.2.4 Module

Module オブジェクトの構成を図 5.3に示す。

Module クラスはメディアデータを処理する各種モジュールの親クラスとなるもので、全ての Module が共通して持つべきメソッド、アトリビュートを持っている。実際にデータをディスクから取得、コンバート、ジッタコントロール、出力するといった In-band を形成する Module はこの Module クラスを継承して作成する。

Module クラスは最低限、データ処理を行なう Handle メソッド、Link との接続を管理するための Connect,Disconnect メソッドを持つ。また本システムでは動的 QoS 制御やメディア間同期の命令をサポートするため、この命令を受ける QosCall メソッド、SyncCall メソッドを持つ。更にその他のメディアデータ処理制御の Command を利用するなら Command から呼び出されるメソッドもここで提供する。例えばメディアデータ処理の開始命令を実装するならそれを受け取る為の Start メソッドを作成しておく必要がある。Module クラス内では Command によって呼び出されるメソッドは NULL にしておき、各サブクラスでオーバーライドして定義する。

図 5.3で Module のサブクラスの実例として動画データをディスクから取得する VideoSource、動画データをコンバートする VideoConvert、動画データを出力する XOutput を示している。図 5.3ではメディアデータ処理に必要なアトリビュートを各 Module ごとに定義している。またメソッドは通常のメソッドの他に Start, Stop, Seek のメソッドがある。このことから、メディアデータ処理の制御 Command として Start, Stop, Seek があるのが判る。

5.2.5 Command

Command オブジェクトの構成を図 5.4に示す。

表 5.2: Module クラスが提供するメソッド

メソッド名	機能
handle	メディアデータを処理する
connect	Link を Module に接続する。
disconnect	Link を Module から外す。
QosCall	QoS を設定する。
SyncCall	同期処理を行なう。

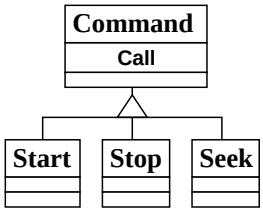

図 5.4: Command クラス

Command オブジェクトは Module が提供するメディアデータ処理の制御メソッドを呼び出すもので、Stream に通知するものである。Stream の CommandCall メソッドを呼び出す時に Command オブジェクトを引数として渡すと、Stream のメソッドで Command が保持する Delivery オブジェクトを利用して Module を検索して、Command オブジェクトの Call メソッドが Module のメソッドを呼び出す。

5.2.6 Delivery

Delivery オブジェクトの構成を図 5.5に示す。

Delivery オブジェクトは Stream に対して Command が与えられた時に利用するオブジェクトであり、その Stream の配下にある Module を検索するオブジェクトである。Stream に命令が通知されると、Delivery オブジェクトがその Stream 内の Module を探索し、Stream 内のどのモジュールをどの順番で呼び出すかを決定する。それに沿って Command モジュールの Call メソッドを利用して、Module のメソッドを呼び出す。

表 5.3: Command クラスが提供するメソッド

メソッド名	機能
Call	Module のメソッドを呼び出す。

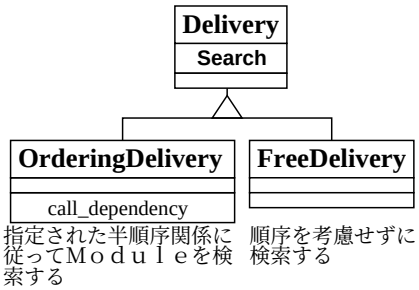

図 5.5: Delivery クラス

表 5.4: Delivery クラスが提供するメソッド

メソッド名	機能
`Search`	Stream 内の Module を検索して、引数で指定された Command を実行する。

Delivery クラスは Module を呼び出す順番を指定するものである。このため、Delivery クラスが持つメソッドは基本的に Stream 内の Module をサーチするメソッドのみである。図 5.5では、Delivery ポリシーの例を 2 つ示している。OrderingDelivery は SetAttribute で Module 間の呼び出し順序を半順序で指定して、その半順序に従って Module のメソッドを検索する。また、FreeDelivery は検索順序を一切考慮せずに Module を順に検索する。

このように命令配送のポリシーをいくつか定義しておき、各種 Command に最も適したポリシーを与えるようにしておく。

5.2.7 SyncControl

Stream 間同期の処理を行なうオブジェクトであり、実際には図 5.1で示している SyncControl クラスを継承して実装したサブクラスをインスタンシエーションして利用する。

SyncControl クラスは Stream 間の同期をとるために作成された特殊な制御命令である。Stream オブジェクトの内部でこの Module を保持し、その子供の Stream に対して時間的な同期をとるように命令を伝搬させる。図 5.2(b) の Stream クラスで示しているように、本システムでは Stream オブジェクトの内部に`sync_param` という変数を持たせてある。このアトリビュートは SyncControl オブジェクトが使用するものであり、Stream 間の同期処理を行なう為に各 Stream が持つパラメータをこの変数に格納する。このようなストリーム固有の値を持つことで複数の Stream の間で共有の値を保持できるようにして、複数の Stream にまたがった制御を行うことができるようにしている。

表 5.5: SyncControl クラスが提供するメソッド

メソッド名	機能
SyncCall	指定された Stream 以下で同期処理を行なう。

表 5.6: QoSControl クラスが提供するメソッド

メソッド名	機能
QosCall	QoS パラメータを算出し、指定された Stream 以下に QoS パラメータを設定する

5.2.8 QoSCotnrol

動的 QoS 制御を行なうオブジェクトである。このオブジェクトも図 5.1で示しているクラスの構成をそのまま継承して利用する。動的 QoS 制御を行うオブジェクトである。このオブジェクトも図 5.1で示している QoSControl クラスを継承して実装したサブクラスをインスタンシエーションして利用する。

QoSControl クラスは動的 QoS 制御の命令を伝搬させる為に作成された特殊な命令である。CPU 資源の使用状況を監視し、またユーザからの QoS 制御要求を受けて、Stream に対して動的 QoS 制御を行なう。SyncControl と同様に、各 Stream は`qos_param` という変数を属性に持っている。QoSControl は各 Stream の QoS 係数など Stream ごとに管理すべき情報をここに保存し、その値を利用して制御する。

5.3 Module の接続とデータ受渡し

本システムで、Module 間の接続を行なう手順を図 5.6に示す。

Module 間の接続を行なう為に、Out-of-band から Link オブジェクトの Connect メソッドを呼び出す。そのメソッドでは以下の作業を行なう。

- Link オブジェクトが Module へのポインタを保持する
- Module オブジェクトに接続要求があったことを通知する
- Stream オブジェクトに Module が所属したことを通知する

なお、Module オブジェクトの Connect メソッドでは自身のポートに Link オブジェクトを保持しておくなどの作業を行ない、Stream オブジェクトの AppendModule メソッドでは新たに自分の配下になった Module を保持する。

次に、Module 間でデータを送受信する手順を図 5.7に示す。

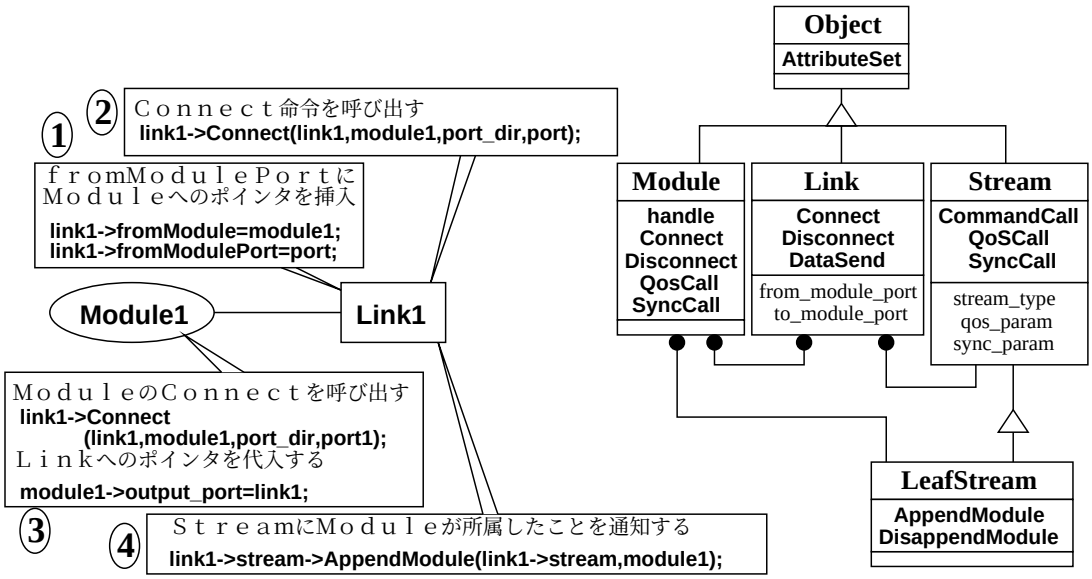

図 5.6: Module 同士の接続手順

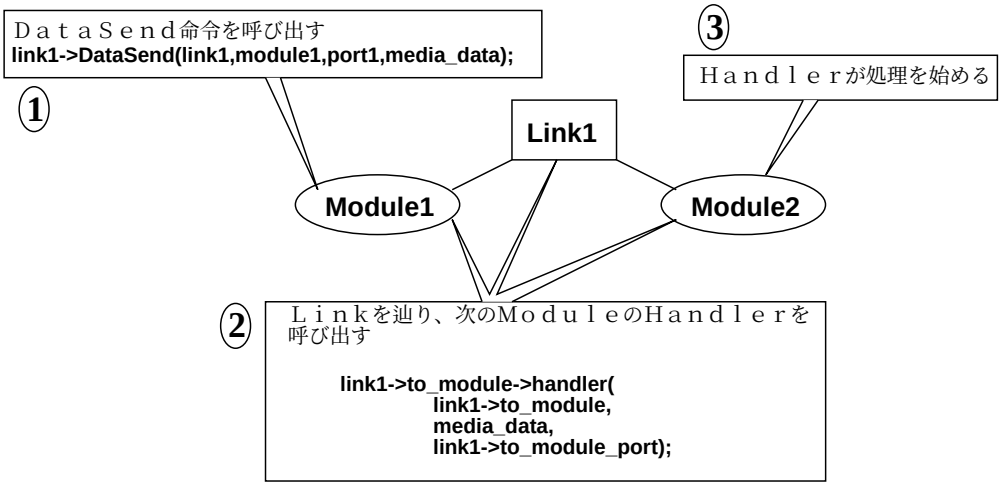

図 5.7: Module 間でデータを受渡す手順

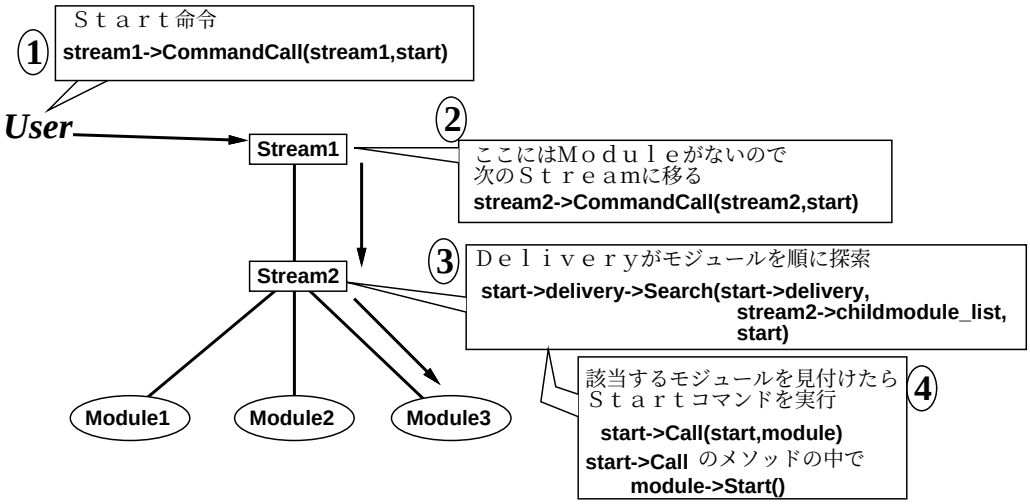

図 5.8: Start 命令呼び出しの様子

本設計では基本的に Module の Handler メソッドでメディアデータ処理を行なう。そして一連のメディアデータ処理が完了したら、次の Module にそのデータを渡す作業が必要になる。Module1 から Module2 へのメディアデータの送信は、通知された Link のメソッド DataSend を呼び出せば良い。

Link の DataSend は次の Module の Handler メソッドを呼び出す。引数ではメディアデータへのポインタを渡す。こうすることで、次の Module の Handler メソッドが実行できる。

Module 間でメディアデータを受け渡す作業はメディアデータの処理中に何度も発生するため、高速に処理できなければならない。本システムでは、このように Module 間のデータ受渡しを関数呼び出しで実装しているため、非常に高速にデータの受渡しができる。

5.4 Command の呼び出し手順

本システムで、Command を呼び出す手順を図 5.8に示す。

まず、Out-of-band から Stream に対して命令を送る。これが図 5.8 であり、このために使用するオブジェクトのクラスを図 5.9 で示す。呼び出しは Stream のメソッド CommandCall を呼び出すようにする。たとえば、stream1 に対して start 命令を呼び出すのは以下の形式で記述する。

```
stream1->CommandCall(stream1,start);
```

stream の CommandCall メソッドの内部は LeafStream と GroupedStream で内容が異なる。この stream1 は GroupedStream であるため、自分の子供である Stream2 の CommandCall メソッドを呼び出すようにする。つまり、このメソッドの内部では自身のアトリビュート `child_stream` をサーチして stream2 を見付け、

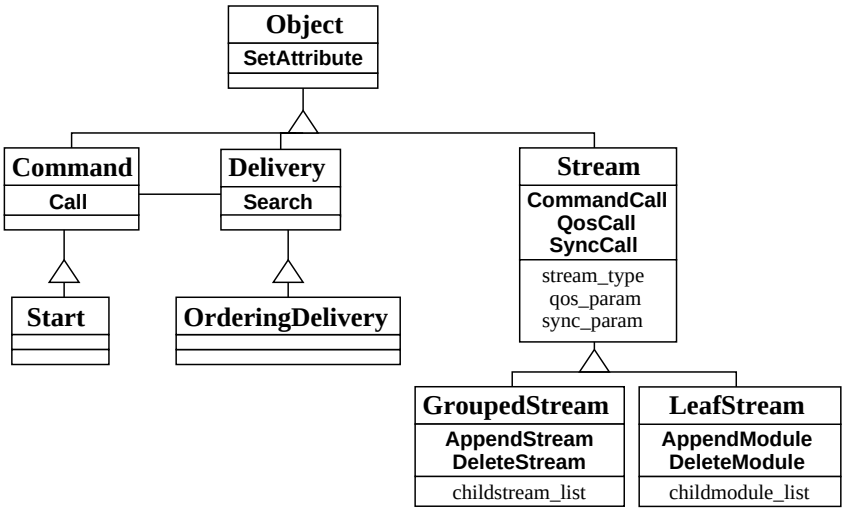

図 5.9: Start 命令呼び出しで使用されるオブジェクトのクラス

```
stream2->CommandCall(stream2,start);
```

という処理を行なう。stream2 は LeafStream である。この CommandCall メソッドの内部は Module を保持している。この Module の中から Start 命令を受け取る Module のみを、delivery が要求する順番で呼び出す。Start 命令を受け取る Module を順番に並べかえるのが delivery のメソッド Search である。

```
start->delivery->Search(start->delivery,
                        stream2->child_module,
                        start);
```

この Search メソッドで Start 命令を必要とする Module を捜し出す。そして、それらを規定の順番で呼び出す。これは

```
start->call(module);
```

である。このメソッドの内部では

```
module->start();
```

を行なっている。以上の手順で Module が提供している Start メソッドが呼び出される。

QoS 制御の要求は上記とは異なった手順で処理する。QoS 制御命令はユーザが特定の QoS 値を直接指定する目的で Out-of-band から要求されたり、メディアデータの処理が追い付かないので QoS を下げる為に In-band(Module) から QoS 制御要求を出す場合がある。また QoS 制御オブジェクトが CPU 資源を監視し、その状況に応じて QoS 制御自身が制御を要求するということが考えられる。この状況を示したのが図 5.10である。

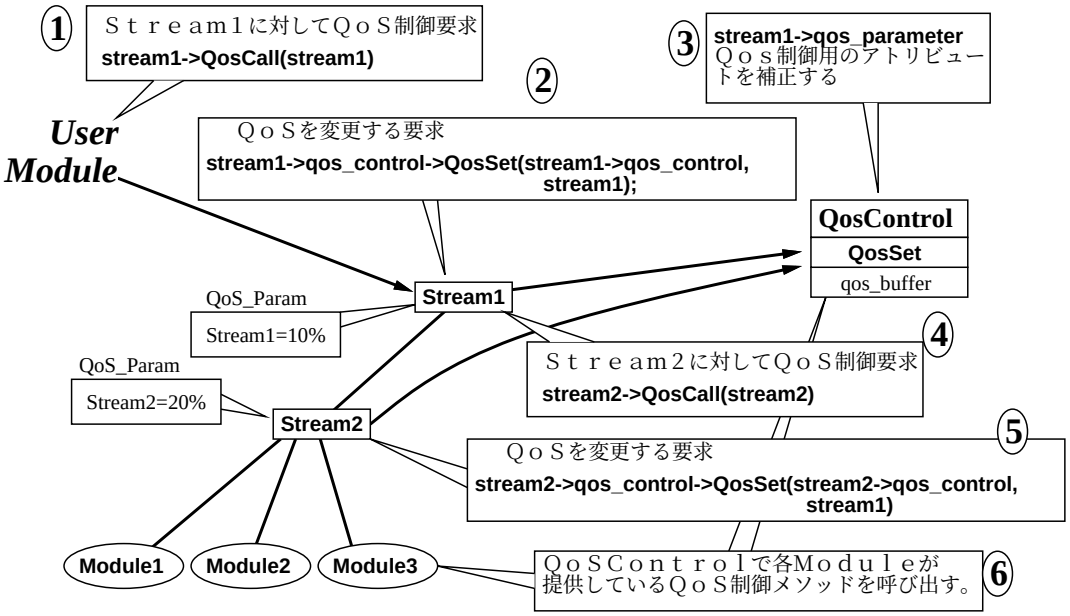

図 5.10: QoS 制御命令呼び出しの様子

QoS 制御命令は通常の Command とは異なり、Stream が提供している QoS 制御専用のメソッドを利用して、呼び出しを行なう。この QoS 制御命令の呼び出しは以下のようになる。

```
stream1->QoSCall(stream1)
```

stream1 の QoSCall メソッドはまず QoSControl オブジェクトを呼び出す。

```
stream1->qos_control->QosSet(stream1->qos_control,stream1);
```

QoSControl のメソッド QosSet では、実際に QoS パラメータをどの様に変化させるかを算出する。各 Stream は`qos_parameter` という値を保持している。これは QoS 算出時に使用する領域で、ここには Stream 固有の数値が入る。また、QosControl の内部のアトリビュートも参照する。これらの値から新たな QoS パラメータを算出する。この Stream は GroupedStream であるため、自分の子供の Stream の QoSCall メソッドを呼び出す。

```
stream2->QosCall(stream2)
```

ここの内部でも同様に QosControl の QosSet メソッドを呼び出す。

```
stream2->qos_control->QosSet(stream2->qos_control,stream2);
```

stream1 の時と同様に処理を行ない、今度は QoS パラメータを必要な Module に設定して終了する。以上の手順で QoS 制御を行なうことが出来る。

このような構成を取ることで以下のような柔軟性が生まれる。

- QosControl が単一のオブジェクトとなっているため、QoS 制御のポリシーを交換しやすい。
- 各 Stream ごとに変数を保持し、それを統一した形式でアクセスできる。また、全 Stream 共通の領域は QoSCotnrol のアトリビュートに保持することができる。つまり、QoS 制御のための情報を管理しやすい。
- QoS 制御命令は Out-of-band、In-band の両方から呼び出される可能性があるが、これらの要求を統一されたインターフェースで扱うことが出来る。

また、メディア間同期の呼び出しも QoSControl と同様に実装し、各 Stream には SyncCall メソッドというメディア間同期処理専用の命令を持つ。これを呼び出すことで、Stream 単位でメディアデータ出力予定時刻を一括管理し、同期をとる Stream でメディアデータ処理予定時刻が同じになるように管理する。これでメディア間同期処理が実装できる。

第 6 章

ツールキットの利用例

本システムの適用例を挙げ、本ツールキットの汎用性について考える。

6.1 ビデオプレーヤ

本ツールキットを利用してビデオプレーヤを作成する。ビデオプレーヤは以下の機能を持っている

- 動画像データと音声データを再生する
- メディア間の同期を取る
- メディアデータ再生の開始、停止、シークが可能である

ビデオプレーヤの実装を図 6.1に示す。

このシステムでは、メディアデータを CRAS を経由してディスクから読み込む。CRAS[14] は Constant Rate Access Server の略であり、利用して一定時間毎にビデオデータをディスクから取得、メモリ上に配置するサーバである。CRAS はディスクのシークタイムなどの情報を保持しており、これらの値を利用してメディアデータの先読みを行なっている。このサーバを利用することでディスクからの読み込みで発生するジッタを最小限に抑えることが出来る。

ビデオプレーヤを作成する為に以下の Module を利用する。

VideoCrasInputModule

ビデオデータを CRAS から取得する。

VideoCrasInputModule はビデオデータの 1 フレーム分を読み込み、それに時刻情報を付加して次の Module に渡す。この Module ではメディアデータ処理のスレッドを生成している。

VideoConvertModule

1 フレーム分のビデオデータを QuickTime 形式から Zpixmap 形式に変換する部分である。変換

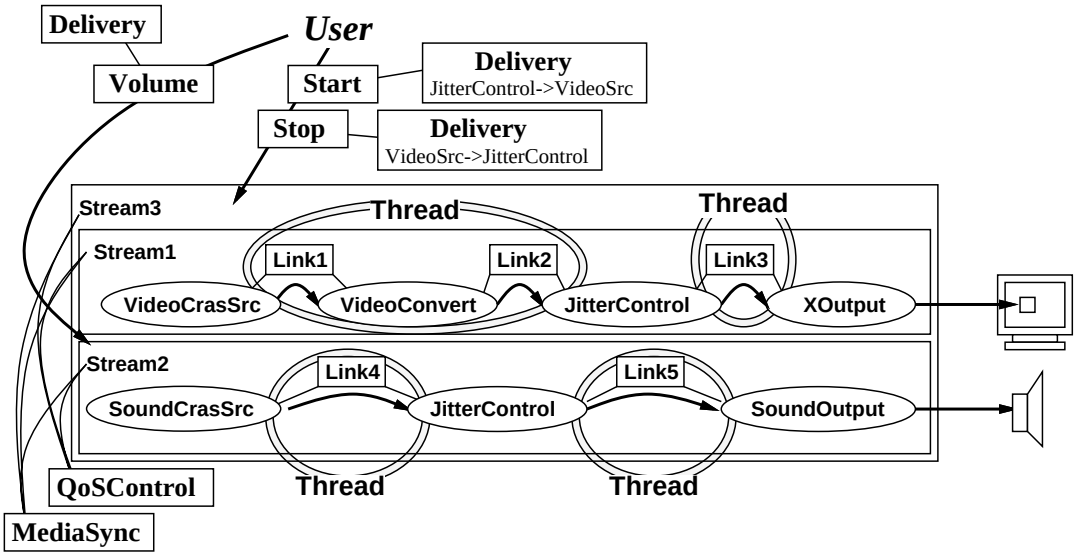

図 6.1: 連続メディアアプリケーション例

である為、メディアデータのコピーが発生するが、時間情報の操作などは行なっていない。この Module ではスレッドの生成を行なっておらず、前段の Module で DataSend を行なったスレッドが VideoConvertModule の Hander メソッドを直接実行する。

JitterControlModule

ジッタ除去を行なう。内部で FIFO キューを持っており、手前の Module で DataSend を行なうと、FIFO 挿入ルーチンが呼び出され、このキューの最後にデータを格納してリターンする。並行してこの Module 内ではスレッドを生成しており、一定時間毎にキューの先頭のデータを DataSend で次の Module に送信する。

VideoXOutputModule

ウインドウを一つ生成し、そこに ZPixmap 形式のデータを表示する。

SoundCrasInputModule

CRAS を介して音声データを取得する。VideoCrasInput と同様に、音声データを取得後、時間情報をメディアデータに書き込んで次の Module に渡す。この Module ではスレッドを生成し、そのスレッドがメディアデータ処理部を実行する。

SoundPasOutputModule

音声データを PAS サーバ [16] に渡す。PAS サーバは ProAudioSpectrum カードにデータを渡して音声を再生する作業を行なうサーバである。

これらの Module を作成する。音声 Stream と動画 Stream の両方とも、SourceModule と JitterControlModule の内部でスレッドを生成している。これらの Module を図 6.1のように接続する。

動画側の Stream を Stream1 とし、音声側の Stream を Stream2 とする。更にそれらをグループ化した Stream を Stream3 とする。

Start,Stop,Seek の三つの Command を実装する。StartCommand,StopCommand,SeekCommand 全て音声、動画像両方の Stream に対する命令である。これらはいずれも、単純に Module のメソッドを呼び出すだけで、命令の配送は Delivery が行なう。

Delivery はユーザが明示的に半順序を指定して順序制御を行なうものを利用する。この実装では、1 種類の DeliveryClass だけを用意し、そのクラスからインスタンスを 3 つ生成し、各 Command にアサインするようにした。

この Delivery では Delivery に対して直接半順序関係を指定する。Delivery 内部ではその半順序関係を保持して、指定された半順序に沿って Module のメソッドを呼び出す形式にしている。

StartCommand はシンク側→ソース側の順番に呼び出すべきであり、StopCommand はソース側→シンク側の順番に呼び出すべきである。また、SeekCommand の順序設定は必要ない。このため、各 Module の Delivery は図 6.1のように指定している。

また、QoSControlCommand に CPU 資源の最大値を与えておくことで CPU 資源監視を行なうようになる。

以上のように In-bandModule を作成することでビデオプレーヤが完成する。

Stream に対して StartCommand を通知すればメディアデータの処理を開始する。また、StopCommand、SeekCommand も同様に動作する。

6.2 ビデオコンファレンスシステムへの拡張

ここで仮にビデオコンファレンスシステムを作成することを考える。ビデオコンファレンスシステムは以下のように実装すれば良い。

図 6.2のように、図 6.1とほとんど変化はない。違うのは SourceModule が、CRAS からデータを取得する Module から カメラやマイクからデータを取得する Module に変わるだけである。

その他の時間依存処理部分などは全く変更する必要がない。このように本システムを利用することで、連続メディアアプリケーションの拡張性、再利用性が非常に高くなる。

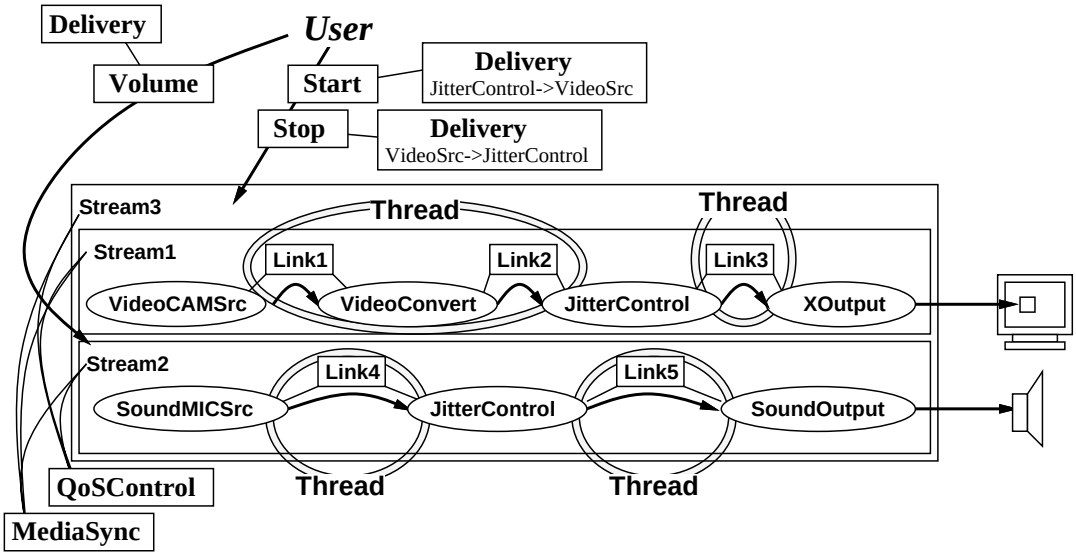

図 6.2: ビデオコンファレンスシステム

第 7 章

評価

この章では本研究で実装したシステムの検証を行なう。第 2 節では連続メディアデータを扱う上で必要な処理を列記している。この各処理が正常に動作するかの検証するために 2 つの実験を行ない、その結果を考察する。

7.1 ジッタコントロール、メディア間同期の検証

本研究で提案したシステムでジッタコントロールや Stream 間同期機構が正常に動作することを検証するため、以下の実験を行なう。

この実験ではメディアデータを流す Stream を 2 つ作成して動作させる。そしてその 2 つの Stream の処理状況を記録し、考察する。

実験のためのシステム構成を図 7.1に示す。

Stream1、2 にはメディアデータを取得 (VideoCrasSrc)、コンバート (VideoConvert)、ジッタコントロール (JitterControl)、画面への出力 (XOutput) の処理を行なう Module それぞれが存在する。この 2 つの Stream を並行して動作させ、メディア間の同期処理を行なう。実行すると図 7.2の図のような出力をする。

この処理状況を検証するためのデータを得る為に、XOutputModule に以下の変更を加えた。

- Module の生成時に、動作ログの一時保管用メモリを確保する。
- メディアデータが到着すると、確保したメモリに「メディアデータの時刻 (最初のフレームを 0 として何秒後のデータかという時間情報)」と「出力を行なった時刻」の 2 つを保存する。
- 停止命令が来たら保存したログをディスクに書き込む。

以上の処理で「あるフレームデータがいつ再生されたか」のデータをディスクに書き込むことができる。この処理を 2 つの XOutputModule が同時に行なうことで、2 つの Stream でメディアデータがいつ再生されたかを知ることができる。

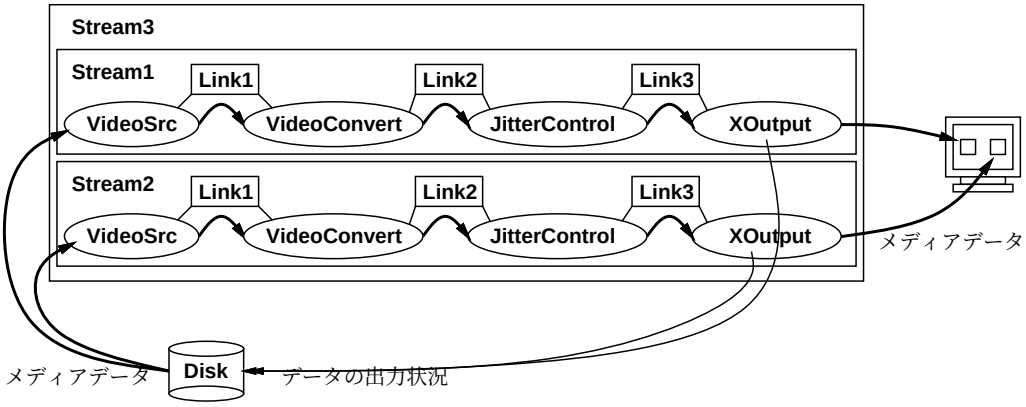

図 7.1: 実験システムの In-band の構成

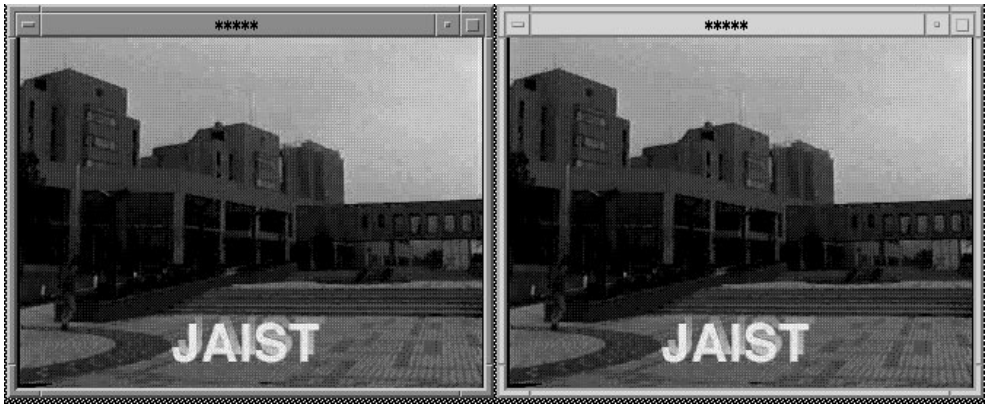

図 7.2: 動画再生の様子

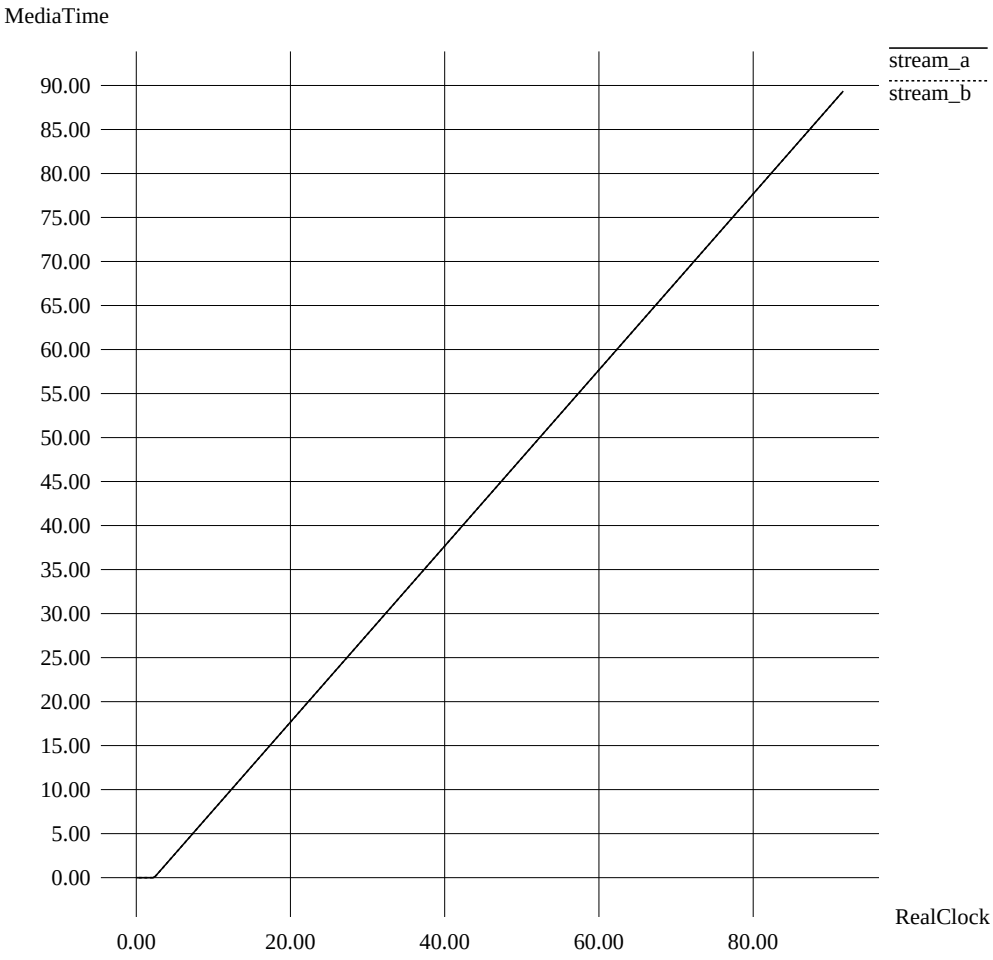

図 7.3: メディア間同期の処理状況 1

結果を図 7.3に示す。

グラフの X 軸が実際の時刻である。また、Y 軸は出力されたフレームのメディア時刻である。

メディアデータの再生を開始した直後に線形でない部分が存在する。これを拡大したのが、図 7.4である。

この再生直後に非線形になった部分は除いて、連続再生した部分は直線となっている。つまり、ジッタが取り除かれているといえる。また、Stream1 と Stream2 の結果を同時にグラフ化したが、1 本の線に重なっていることから、Stream 間同期もきちんと行なわれていることが判る。

実験では、2682 フレームを出力した。二つの Stream 間の時間のズレは約 1.0 から 4.0 の範囲に収まっている。

フレームの再生時刻のずれを表 7.1に示す。

この誤差の原因だが、本プログラムの他にも同時に様々なプロセスが動作しているため、このビデオプレーヤよりもプライオリティの高いプロセスの影響をうけたことが考えられる。

表 7.1: Stream 間のズレ

全フレーム数	ズレ範囲 [sec]	範囲内のフレーム数	フレーム全体に占める割合
2682	0.002 以内	369	0.138
2682	0.004 以内	2680	0.999

メディアデータの再生開始直後に非線形の部分がある。これを拡大し、解説を加えたのが図 7.4である。

図 eva1-data-b で、まず最初の再生開始点が遅れている。これは CRAS の初期化、命令の伝搬のために遅れた物と考えられる。

次に、プログラム起動後 2.33 秒付近から急にデータの再生が遅れている

この非線形部分は再生予定時刻の自動補正によるものである。本プログラムでは、第 2 章で述べたように、再生予定時刻の補正を行なうことでメディアデータの時間依存処理を満足するようにしている。この再生予定時刻の増加量は本実装では 100msec としている。

つまり、このフレームは再生予定時刻を超過して JitterControlModule に到着したものと考えられる。このためコールバックが発生し、メディアデータ再生予定時刻が 100msec 遅くなった。また、次のフレームまでの時間差はもともと 33msec なので、この 2 者を足した 133msec 後に次のフレームが再生されている。この制御のため、以降フレームの再生時刻は 100msec 遅くなっている。

この実験の結果、本システムのアプリケーション構成を利用して、メディアデータ処理時のジッタコントロール、ストリーム間同期の機構が正常に動作することがわかった。

7.2 動的 QoS 制御の検証

この実験は動的 QoS 制御機構を実装し、その動作を確認することで動的 QoS 制御機構の有効性と、本システム構成でこのような処理が実装できるということを検証するものである。

動的 QoS 制御を実装する為に、メディアデータを流す Stream をひとつと CPU 資源の過剰使用を通知する QoSControl を作成した。この構成を図 7.5に示す。

実験では、アプリケーションのメディアデータ処理に消費した CPU 資源を監視し、消費 CPU 資源が一定以上になるとメディアデータの品質を低下させるような制御を行う。こうすることで過剰に CPU 資源を利用しないようにしている。

本実験ではメディアデータのフレームを 60frame/sec で読み込み、画面に出力する。

まず、ビデオデータをそのまま出力した時の CPU 資源使用量の推移を図 7.6に示す。図中、x 軸は時刻、y 軸は CPU 資源の使用量を示している。

このようにビデオデータの処理のために消費する CPU 資源量は大きく変動している。

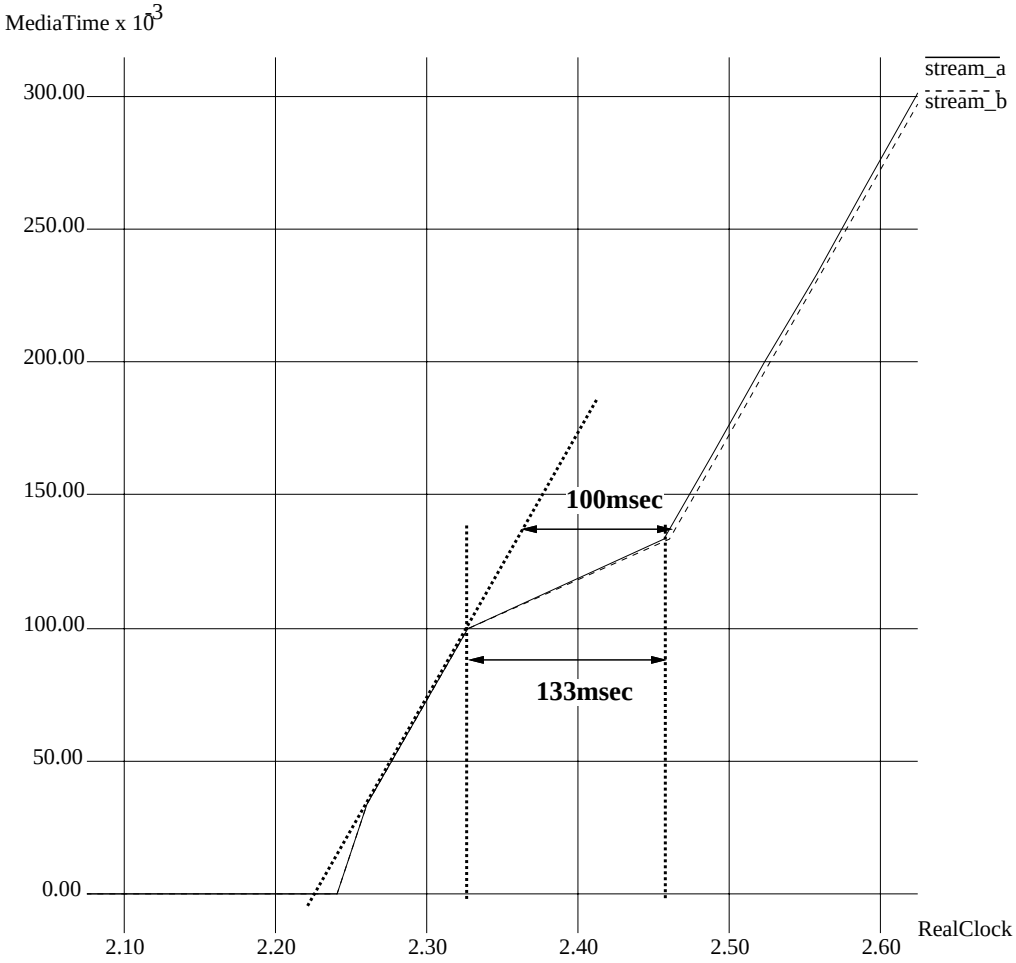

図 7.4: メディア間同期の処理状況 2

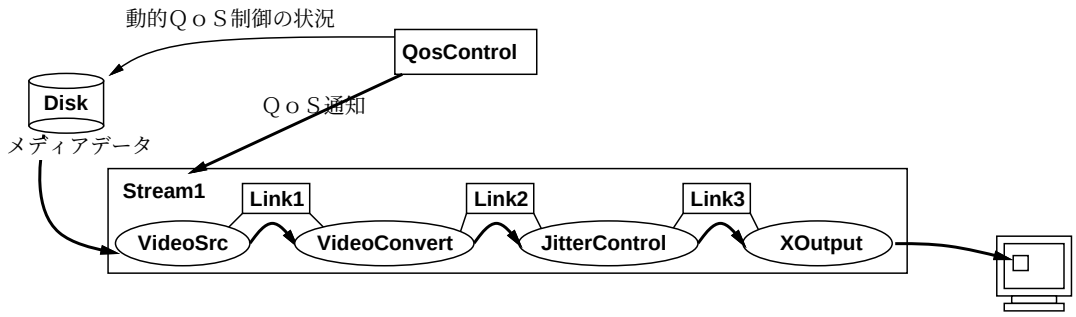

図 7.5: 実験システムの In-band の構成

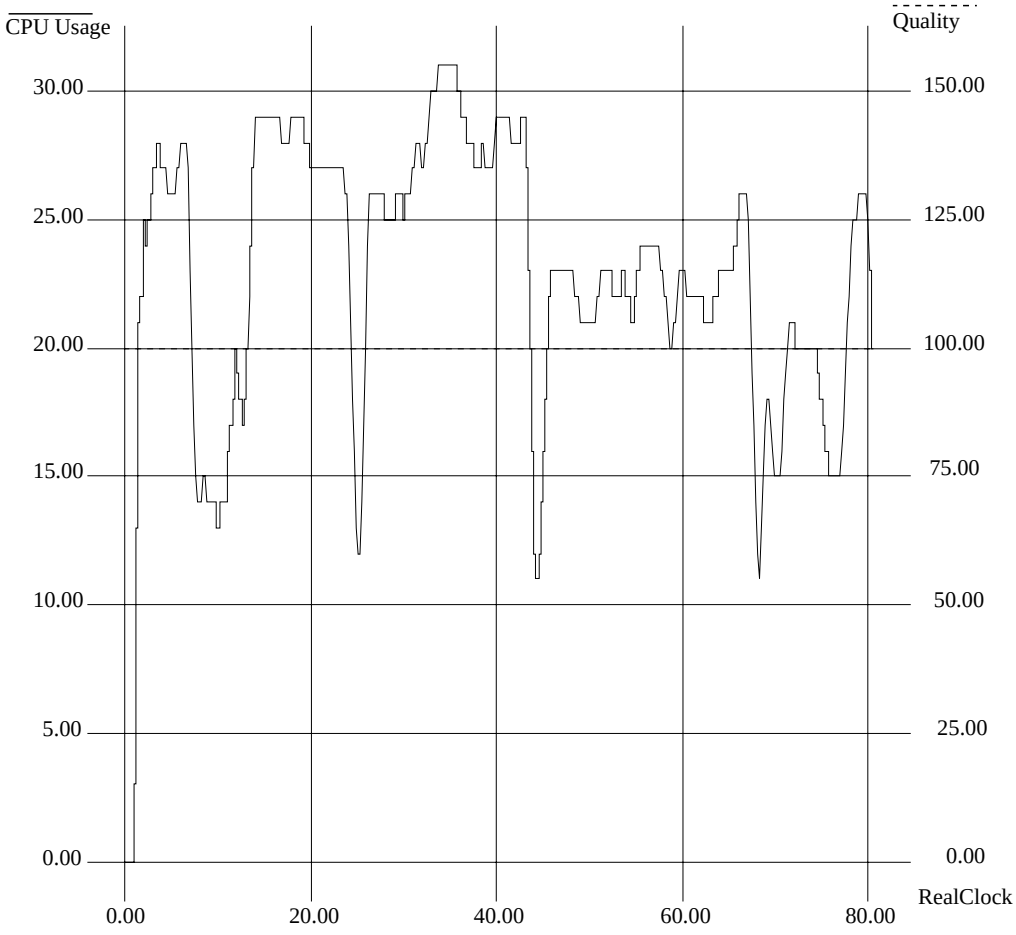

図 7.6: CPU 資源消費量の推移 (スケーリングなし)

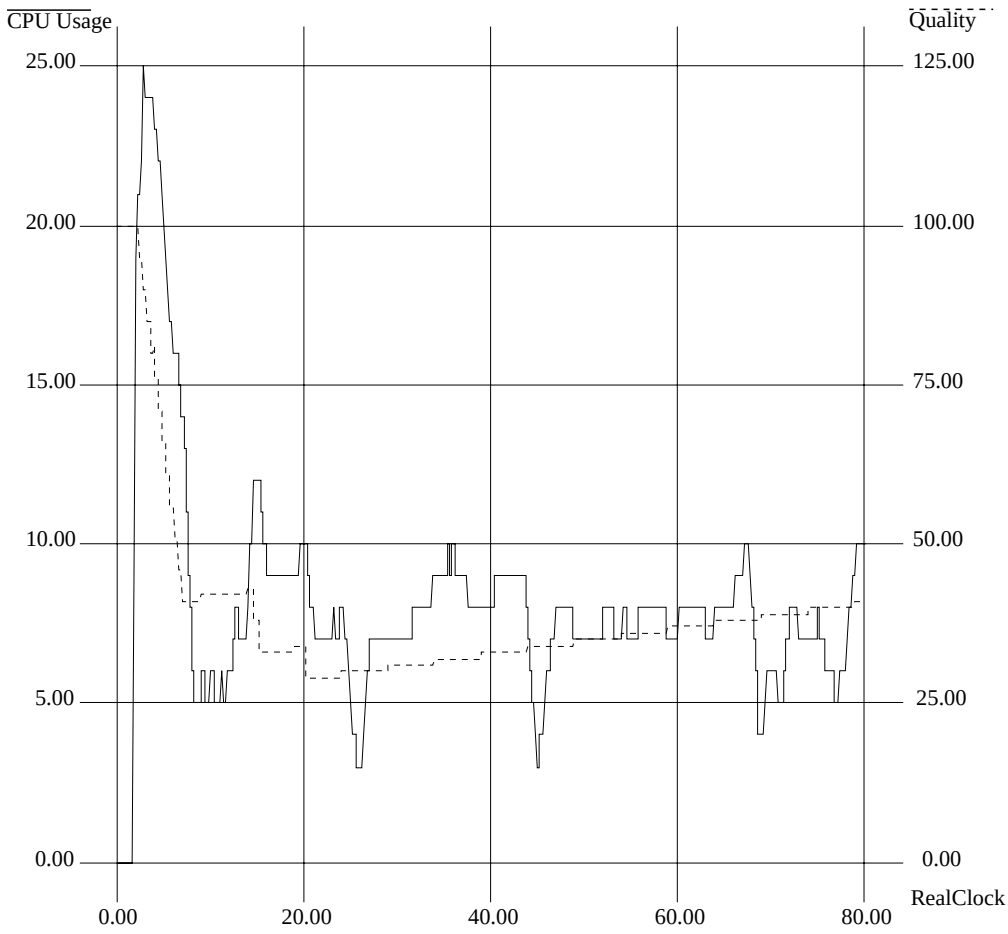

図 7.7: CPU 資源使用量の推移 (スケーリングあり)

ここで、CPU 資源量の上限を 10%とする。そのまま出力したのでは、図 7.6で示すように CPU 資源は 10%を超過してしまう。(最大で 32%)

QoSControl は規定以上の CPU 資源を消費するとメディアデータの品質 (QoS) を低下させる命令を送る。これに応じて VideoCrasSrcModule でビデオデータの再生フレームレートを落すようにする。

この制御をおこなった結果、ビデオデータ再生時の CPU 資源消費量は図 7.7のようになった。

グラフの横軸は時間、実線は CPU 資源使用量で左の縦軸を、破線は QoS で右の縦軸で示している。処理開始直後は CPU 資源使用量が予約した 10%を大きく越えているが、その後 QoS 制御命令が発生し、急激に CPU 資源量が低下する。メディアデータの品質は最低で 28%程度 (17frame/sec) であるが、クオリティの最適値を探る為にすこしずつメディアデータの品質を上昇させている。処理開始直後を除けば CPU 資源消費量は指定された 10%を常に下回ることが出来た。

この実験の結果、本研究で提案したメディアデータ処理プログラムの構成を直接採用することで動的 QoS 制御機構が行なえることが判った。このように本システムでは複数の Stream を統括した自動制御をサポー

トしている。

第 8 章

結論と将来の発展

8.1 結論

本研究では、連続メディアアプリケーションの構成を提案し、その構成にそったアプリケーションが構築できるようなツールキットを作成、使用して評価を行なった。このツールキットは、拡張性を考慮しつつ連続メディアデータを処理する際に生じる問題の解決を狙ったものである。

本ツールキットでは、まずメディアデータの処理を細分化しそれらを接続することで目的のメディアデータ処理を実現するようにした。このような構造を利用することで、連続メディアアプリケーションに要求される様々な事柄に柔軟に対応できるようになった。新たな機能を要求されても、一部分の追加だけで目的のソフトウェアを製作できる上に、動作中に Module 間の接続を切替えることでメディアデータ処理機能を動的に切替えることができるようになった。

更に本ツールキットでは連続メディアデータ処理機能だけでなく、このメディアデータの処理の流れを Stream として定義しており、この Stream に対して高度な制御が行なえるような構造を提供している。この機構を利用して、メディアデータの統合的な流れの制御である「複数のメディアデータ間での同期処理」「CPU 資源の状況を監視し、それに応じて提供するメディアデータ品質を制御する動的 QoS 制御」の実装が可能となった。また、本システムでは、クラスの継承を利用してこの制御部分 (SyncControl、QoSControl) のメソッドを置き換えることで新たなポリシーを実装できる。

上記の QoSControl や SyncControl の他にメディアデータ処理を制御する命令 (Command) をサポートしている。Command も QoSControl や SyncControl と同様に Stream に対して通知するものであり、メディアデータの処理開始や処理停止などの基本的な操作を Command として実現する。このような構成を利用することで、各 Module がどのような命令を欲しているかをユーザが直接知る必要がなく、ユーザは Module を接続してメディアデータの流れを作り、そのメディアデータの流れに対して命令を通知すれば後は自動的にツールキット側でそれに適した処理を行うようになる。本システムではこれらの Command をメディアデータを処理する Module からも呼び出すことができ、これによってフィードバック制御を実現

できる。

更に Command の配送の作業を行なう部分を Delivery という単一のオブジェクトが管理するようにしている。Command の配送先やその順序を制御する時はこの Delivery オブジェクトのサブクラスを新たに製作すれば良い。

また、このツールキットを実際に利用してビデオプレーヤを作成し、このメディアデータ間の同期処理と動的 QoS 制御機構を確認する実験を行い、メディアデータ間の同期処理や、動的 QoS 制御の確認を行った。

8.2 今後の研究の発展

最後に今後本研究を発展させる課題に付いて述べる。

8.2.1 分散環境への対応

本システムは集中環境下でのメディアデータ処理機構を提案し、実装した物である。しかしながら、分散環境は考慮できておらず、その対応が必要である。特に、本システムを分散環境下に適用する場合は以下の問題が発生する。

命令配送の問題 分散環境下ではメディアデータ処理システム自体が複数の計算機にまたがって実装されることになる。このため、Stream も計算機間をまたがって存在することになる。現在は Command の配送は同じ仮想アドレス空間にあるデータをオブジェクト間で受け渡すという操作を行っているが、分散環境下ではこのような実装ができなく、何らかの通信手法が必要である。

このような分散環境下での命令管理の方式として本研究室の古野氏はネットワーク上に連続メディアアプリケーション用のネームサーバを配置し、それに対して Stream の情報などを登録したり質問することを提案している。

クロックスキュー問題 分散環境下ではメディアデータが複数の計算機の間を流れることになる。計算機が異なるため、利用する計算機の内部時計の時刻や時間の進む速度が異なる。これをクロックスキュー問題と呼ぶ。現在の実装では、単一の計算機内だけで処理を行なっているので、システムが提供している時計を絶対時刻としてそのまま利用しているが、分散環境ではこのクロックスキュー問題の解決が必要になる。

ネットワーク資源の問題 分散環境ではネットワークを介してメディアデータの受渡しをすることになる。しかしながら、連続メディアデータは大容量であることが多く、そのようなデータを他の計算機も繋がっているネットワークに何の制限もなく送るのは問題がある。

このため、ネットワーク資源予約を利用し、一定量以上のネットワーク資源を使用しないようにしたり、或はメディアデータを流すのに充分なネットワーク資源が確保できない場合はメディアデータに

変換を加え、メディアデータの品質を送出側で管理する必要がある。このネットワーク資源予約をサポートしたシステムに、本研究室の村上敦氏が提案した NPL がある。分散型のメディアデータプレーヤを実装する際にはこのシステムを利用することでネットワーク資源の予約が可能になる。

8.2.2 In-band の拡張性

また、本システムは、現在はメディアデータを処理する Module は C 言語で作成するようにしている。こうすることで、メディアデータ処理は高速に行なえるようになっている。しかしながら、昨今の計算機環境の向上の御蔭でよりアブストラクションの高い言語でメディアデータ処理部分を記述できるようになりつつある。

このため、Out-of-band にスクリプト言語を提供するアブストラクションレイヤを生成しておき、そこから自由に操作するだけでなく、このスクリプト言語で In-band Module の追加が可能なようにできれば、本システムの拡張性は更に高くなる。

8.2.3 動的 QoS 制御のポリシー

本研究では、実際に動的 QoS 制御を実装して、消費する CPU 資源を制限することに成功した。

しかしながら、この動的 QoS 制御のポリシーを複数実装して比較検討するということは行なっていない。メディアデータを処理する際の消費 CPU 資源量は一定ではなく、再生部分によって大きく変動する。また、同じメディアデータを再生しても、平行して動作する他のアプリケーションの影響も考えられ、常に同じ状態で再生できるとは限らない。動的 QoS 制御のポリシーとその効果についてまとめることができれば常に最適の動的 QoS 制御が行なえるようになる。

謝辞

本研究を遂行するにあたり、様々な面でバックアップして下さった中島研究室の皆さんと中島研究室OBの皆さんに感謝します。殊に研究上貴重な御意見を下さった保木元晃弘氏、新情報処理開発機構の手塚宏史氏に感謝いたします。最後になりましたが、長期間にわたり御指導、御教授を下さりました中島達夫助教授に深く感謝申し上げます。

参考文献

[1] Christopher J.Lindblad,David J.Wetherall,and David L.Tennenhouse, The VuSystem: A Programming System for Visual Processing of Digital Video. 1994

[2] Lawrence A.Rowe and Brian C.Smith , A Continuous Media Player", 1992

[3] David P.Anderson, ACME IMPLEMENTATION NOTES,

[4] David P.Anderson, ACME PROGRAMMER'S GUIDE, 1990

[5] Stuart Wray , Tim Glauert ,and Andy Hopper, The Medusa Applications Environment, 1994

[6] Alan Jones , Andrew Hopper, Handling Audio and Video Streams in a Distributed Environment, 1993

[7] Kurt Rothermel Ingo Barth Tobias Helbig , CINEMA - An Architecture for Configurable Distributed Multimedia Applications April 1994

[8] Tobias Helbig, Timing Control of Stream Handlers in a Distributed Multi-Threaded Environment, 1996

[9] CU-SeeMe, ftp://gated.cornell.edu/pub/video/html/Welcome.html

[10] hintaro Furuno, Tatsuo Nakajima, "A Toolkit for building Continuous Media Applications using a New Dynamic QoS Control Scheme", Multimedia Japan'96, March,1996.

[11] Hiroshi Tezuka, Tatsuo Nakajima, Experiences with building a Continuous Media Application on Real-Time Mach , 2nd International Workshop on Real-Time Computing Systems and Applications, October, 1995.

[12] T.Nakajima and H.Tezuka, A Continuous Media Application supporting Dynamic QoS Control on Real-Time Mach In Proceedings of the ACM Multimedia'94, 1994.

[13] Shintaro Furuno and Tatsuo Nakajima , A Toolkit for building Continuous Media Appilcations using a New Dynamic QoS Control Scheme,1995.

[14] Hiroshi Tezuka, Tatsuo Nakajima, Simple Continuous Media Storage Server on Real-Time Mach, Usenix Technical Conference Winter, January, 1996.

[15] Real-Time Mach Porject , Real-Time Mach 3.0 User Reference Manual, School of Computer Science Carnegie Mellon University , June 1995

[16] Jim Zelenka Real-Time Mach Porject , RT-Mach Audio Server Pro-Audio Spectrum 16(pas)version, Department of Computer Science Carnegie Mellon University , November 1994

第 A 章

メソッド呼び出し管理例

本研究で実装したシステムでは、Out-of-band から半順序の指定を受け、それに応じて配送を行なう Delivery オブジェクトを実際に利用している。この半順序の管理方法について説明する。

Module1〜4 が存在したとする。初期の状態ではこれらの呼び出し順序は指定されていない。そのため、図 A.1(a) のような状態である。つまり、Delivery の内部にある Dependency リストには何も格納されていない。

この状態で Stream を呼び出すと、Module1 から 4 の呼び出し順序は完全に任意となる。

ここで

Module3⇒Module2

という依存関係を SetAttribute を利用して設定する。

リストの内部に存在しない Module はエントリを生成され、リストに挿入される。ここでは Module3 と Module2 の両方が挿入される。Dependency を示す矢印は必ずリストの後方を差すように保存する。この状態では、Module2 は必ず Module3 の後に呼び出されるようになる。その他のリストにない Module1、4 も呼び出しはするがその順序は不定である。

さらに半順序

Module3→Module4

を与える。

Module3 は既に存在するので新たにエントリを生成することはない。Module4 はエントリを生成され、やはり矢印を後方に向けるように Module を配置する。この状態では、Module2 及び Module4 は必ず Module3 の後に呼び出される。但しその他の状態については指定していないので、Module1 と Module2 と Module4 の間の呼び出し順序は不定である (図では Module2 の方が先になっている)。

さらに半順序

Module4→Module2

が与えられると、Module4 から 2 の方向に矢印が追加される。しかし、Module4 から Module2 の方向に

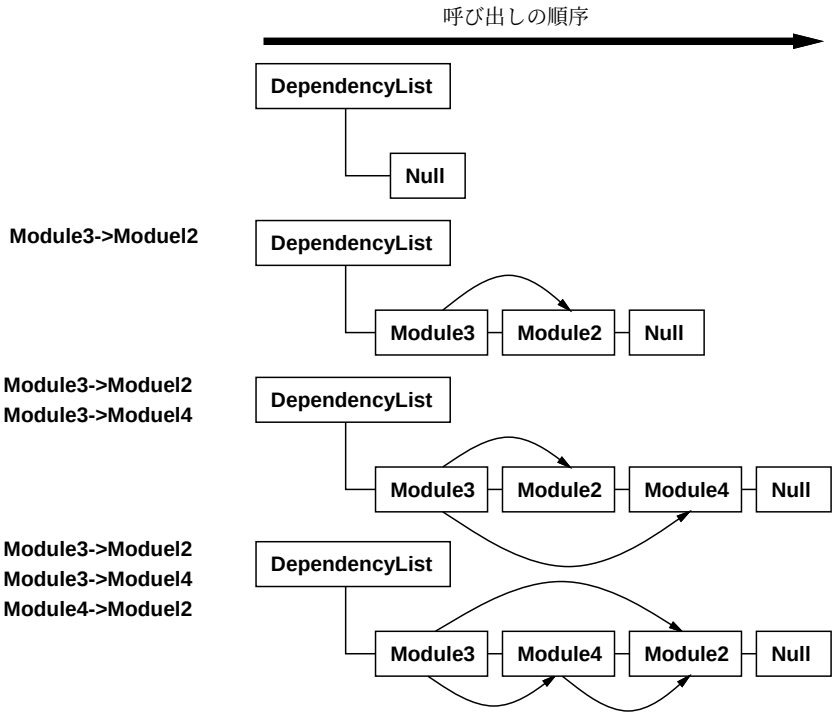

図 A.1: 半順序を与えた例

矢印を置くと、依存の矢印が Command 呼び出しを遡る方向に延びてしまう。これを避ける為に、Module2 と Module4 の順序を入れ替えて順序を修正する。この時、Stream を呼び出されたら、Module3,4,2 という順に呼び出され、Module1 はこの順序のどれかの間に呼び出されることになる。

このような方式で半順序を管理することで、順序を辿るのは通常の線形探索でよくなる。結果として命令配送などの処理が高速に行なえるようになる。

第 B 章

Module 作成例

本システムで作成したempty_module オブジェクトのソースリストを掲載する。

empty_module はmodule を継承して作成されており、入力されたメディアデータを何もせずにそのまま出力するだけの Module である。このため、新たな Module を作成する時のスケルトンとして利用できる。

B.1 empty_module.h

```
/*
  CMT-2c CMT:objects-Interface
  EmptyModule
  MMMC 1997
 */

#ifndef _EMPTY_MODULE_H_
#define _EMPTY_MODULE_H_

#define CMT_EMPTYMODULE_MAX_PORT_NUM 5

typedef struct EMPTYMODULECLASS EMPTYMODULECLASS;
typedef struct EmptyModule EmptyModule;

#include <module.h>

#define CMT_EMPTY_MODULE_TEMPL \
\
CMT_MODULE_TEMPL \


#include <cmt_error.h>
#include <link.h>


struct EMPTYMODULECLASS
{
  struct MODULECLASS *parent;
  cmt_return_t (*Construct)(EmptyModule* );
  cmt_return_t (*Destruct)(EmptyModule* );
```

```
  cmt_return_t (*SetAttribute)(CmtObject* ,char* ,
                               void* ,void* ,void* );
  cmt_return_t (*Connect)(Module* ,ModulePortDir ,int ,Link* );
  cmt_return_t (*Disconnect)(Module* ,ModulePortDir ,int );
  cmt_return_t (*Handle)(Module* ,int ,MediaData* );
  cmt_return_t (*QosControl)(Module* ,void* );
  cmt_return_t (*SyncControl)(Module* ,void* );
  cmt_return_t (*Start)(Module* ,void* );
  cmt_return_t (*Stop)(Module* ,void* );
};


struct EmptyModule
{
  struct EMPTYMODULECLASS *class;

  CMT_EMPTY_MODULE_TEMPL

};


#ifdef _CMT_EMPTYMODULE_IMPLE_
EMPTYMODULECLASS* EmptyModuleClass;
#else
extern EMPTYMODULECLASS* EmptyModuleClass;
#endif /*_CMT_EMPTYMODULE_IMPLE_*/


extern EMPTYMODULECLASS*
EmptyModuleClassCreate(MODULECLASS* parent);
extern cmt_return_t
EmptyModuleClassDelete(void);



#endif /*_EMPTY_MODULE_H_*/
```

B.2 empty_module.c

```
/*
  CMT-2c CMT:objects-Implementation
  EmptyModule
  MMMC 1997
 */

#define _CMT_EMPTYMODULE_IMPLE_

/* unix */
#include <stdio.h>
#include <string.h>

/* cmt */
#include <cmt_error.h>
#include <memory.h>
```

```
#include "module.h"
#include "empty_module.h"



static cmt_return_t
EmptyModuleConstruct(EmptyModule* new_object);
static cmt_return_t
EmptyModuleDestruct(EmptyModule* object);
static cmt_return_t
EmptyModuleSetAttribute(CmtObject* object,char* key,
                  void* arg1,void* arg2,void* arg3);
static cmt_return_t
EmptyModuleConnect(Module* module,ModulePortDir port_dir,int port,Link* link);
static cmt_return_t
EmptyModuleDisconnect(Module* module,ModulePortDir port_dir,int port);
static cmt_return_t
EmptyModuleHandler(Module* module,int port,MediaData* mediadata);
static cmt_return_t
EmptyModuleQosControl(Module* module,void* arg);
static cmt_return_t
EmptyModuleSyncControl(Module* module,void* arg);
static cmt_return_t
EmptyModuleStart(Module* module,void* arg);
static cmt_return_t
EmptyModuleStop(Module* module,void* arg);


extern EMPTYMODULECLASS*
EmptyModuleClassCreate(MODULECLASS* parent)
{
  EMPTYMODULECLASS *new_class;

  new_class = (EMPTYMODULECLASS*)mem_alloc(sizeof(EMPTYMODULECLASS));

  new_class->parent = parent;
  new_class->parent             = parent;
  new_class->Construct          = EmptyModuleConstruct;
  new_class->Destruct           = EmptyModuleDestruct;
  new_class->SetAttribute       = EmptyModuleSetAttribute;
  new_class->Connect            = EmptyModuleConnect;
  new_class->Disconnect         = EmptyModuleDisconnect;
  new_class->Handle             = EmptyModuleHandler;
  new_class->QosControl         = EmptyModuleQosControl;
  new_class->SyncControl        = EmptyModuleSyncControl;
  new_class->Start              = EmptyModuleStart;
  new_class->Stop               = EmptyModuleStop;

  EmptyModuleClass = new_class;

  return new_class;
}

extern cmt_return_t
EmptyModuleClassDelete(void)
{
  mem_free(EmptyModuleClass);
```

```
  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleConstruct(EmptyModule* new_inst)
{
  EMPTYMODULECLASS* class;

  class = EmptyModuleClass;

  /* call parent's constructor */
  if(class->parent != NULL){
    class->parent->Construct((Module*)new_inst);
  }

  /* Set Class */
  new_inst->class = class;

  /* Set Methods (override) */
  new_inst->SetAttribute        = class->SetAttribute;
  new_inst->Connect             = class->Connect;
  new_inst->Disconnect          = class->Disconnect;
  new_inst->Handle              = class->Handle;
  new_inst->QosControl          = class->QosControl;
  new_inst->SyncControl         = class->SyncControl;
  new_inst->Start               = class->Start;
  new_inst->Stop                = class->Stop;

  /* Set Attributes (override) */

  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleDestruct(EmptyModule* inst)
{
  EMPTYMODULECLASS* class;
  int i;

  class = EmptyModuleClass;

  /* error check */
  for(i=0;i<CMT_EMPTYMODULE_MAX_PORT_NUM;i++){
    if((inst->input[i]!=NULL)||(inst->input[i]=NULL)){
      return CMT_FAILURE;
    }
  }

  /* remove parent */
  if(class->parent != NULL){
    class->parent->Destruct((Module*)inst);
  }

  /* remove EmptyModule */

  return CMT_SUCCESS;
```

```
}

static cmt_return_t
EmptyModuleSetAttribute(CmtObject* object,char* key,
                  void* arg1,void* arg2,void* arg3)
{
  EmptyModule* empty_module;
  MediaData test;

  empty_module = (EmptyModule*)object;

  /* for test */
  printf("EmptyModule:[%s]-%d-%d-%d-\n",key,(int)arg1,(int)arg2,(int)arg3);

  if(strcmp(key,"Handle")==0){
    test.id = 100;
    test.payload = NULL;
    EmptyModuleHandler((Module*)empty_module,0,&test);
  }

  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleConnect(Module* module,ModulePortDir port_dir,int port,Link* link)
{
  EmptyModule* empty_module;
  cmt_return_t ret;

  empty_module = (EmptyModule*)module;

  ret = empty_module->class->parent->Connect(module,port_dir,port,link);

  return ret;
}

static cmt_return_t
EmptyModuleDisconnect(Module* module,ModulePortDir port_dir,int port)
{
  EmptyModule* empty_module;
  cmt_return_t ret;

  empty_module = (EmptyModule*)module;

  ret = empty_module->class->parent->Disconnect(module,port_dir,port);

  return ret;
}


static cmt_return_t
EmptyModuleHandler(Module* module,int port,MediaData* mediadata)
{
  EmptyModule* empty_module;

  empty_module = (EmptyModule*)module;
```

```
  printf("EmptyModuleHandler:called!\n");

  printf("port=[%d]\n",port);
  printf("mediadata->id=[%d]\n",mediadata->id);

  printf("module->output[1] = [%d]\n",(int)module->output[1]);

  if(module->output[1] != NULL){
    module->output[1]->Send(module->output[1],mediadata);
  }

  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleQosControl(Module* module,void* arg)
{
  EmptyModule* empty_module;

  empty_module = (EmptyModule*) module;

  printf("EmptyModule:QosControlCall!arg = [%d]\n",(int)arg);

  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleSyncControl(Module* module,void* arg)
{
  EmptyModule* empty_module;

  empty_module = (EmptyModule*) module;

  printf("EmptyModule:SyncControlCall!arg = [%d]\n",(int)arg);

  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleStart(Module* module,void* arg)
{
  EmptyModule* empty_module;

  empty_module = (EmptyModule*) module;

  printf("EmptyModule:StartCall!arg = [%d]\n",(int)arg);

  return CMT_SUCCESS;
}

static cmt_return_t
EmptyModuleStop(Module* module,void* arg)
{
  EmptyModule* empty_module;

  empty_module = (EmptyModule*) module;
```

```
  printf("EmptyModule:StopCall!arg = [%d]\n",(int)arg);

  return CMT_SUCCESS;
}
```