卒業論文　　2004年度 (平成16年度)

LOLCAST: Layered Overlay Multicast Protocol

慶應義塾大学 環境情報学部

氏名：小椋 康平

指導教員

慶應義塾大学 環境情報学部

村井 純

徳田 英幸

楠本 博之

中村 修

南 政樹

平成16年12月29日

卒業論文要旨 2004 年度 (平成 16 年度)

LOLCAST: Layered Overlay Multicast Protocol

本論文では, 階層構造を持つデータの配信に適したオーバーレイ・マルチキャストプロトコルである LOLCAST を提案し, その設計, 実装を行った. オーバーレイ・マルチキャスト技術は IP マルチキャスト技術の代替技術として研究が行なわれており, その適用性の高さが注目されている. しかし, 既存のオーバーレイマルチキャストプロトコルでは, 単一的なデータの配信しか考えられていないため, 受信者の品質に対する多様な要求に応えることができない. 本研究では, この問題を解決するために, 階層構造を持つデータの配信に適した新たなオーバーレイマルチキャストプロトコルによりこの問題を解決した.

キーワード
1. オーバーレイ・マルチキャスト, 2. 映像配信, 3. 資源環境

慶應義塾大学 環境情報学部

小椋 康平

Abstract of Bachelor's Thesis

Academic Year 2004

LOLCAST: Layered Overlay Multicast Protocol

This paper proposes a new overlay multicast protocol for delivering hierarchical structured data and discussed on capability of the protocol. Overlay multicast has been proposed for alternative technology to IP multicast. Existing overlay multicast protocols are only designed for delivering single quality data. Therefore, existing overlay multicast protocol could not offer a service to the recipients, which could apply to the request on quality of data. This paper proposed overlay multicast protocol for delivering hierarchical structured data to solve this problem.

Keywords :
1. overlay multicast, 2. video transfer, 3. resource environment

Keio University, Faculty of Environmental Information

Kohei Ogura

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

インターネットは社会インフラとして成り立ち, その上で無数の人々が活動を行なっている. これらの人々は, 活動の表現の場としてインターネットを用い, 絵画や音楽など様々な創作活動を行ない, その成果を個々に発信している. 本研究では, これら「インターネット上で共通の興味・目的を持ち活動している個人あるいは集団」を, インターネット・コミュニティ (以後, コミュニティと記述) と定義する.

既にこれらのコミュニティは, インターネットを表現の場として利用し, 多くの作品を発信している. それは詩や小説などの文字メディア, 音楽や落語などの音声メディア, 映画や演劇などの映像メディアまでに及ぶ. 将来的には 3 次元映像 [1] を利用した創作活動も登場すると考えられる.

しかし現状では, これらのコミュニティがリアルタイム性の高いコンテンツ配信を行なうことは困難である. 多くの場合, 個々のコミュニティを構成する個人は一般利用者であり, 利用可能な計算機資源やネットワーク帯域資源に限界がある. そのため, 提供可能な受信者数や映像品質の種類には限界があり, 多くの受信者を対象とすることができない. したがって, 現状において, 少年野球の試合中継や地元のお祭りの中継といった地域に根差した報道活動等を個人レベルで成り立たせることは非常に難しい.

これらの問題点の本質には二つの側面がある. 一つは上述の通り, 発信者側の計算機資源・ネットワーク帯域資源に限界があることである. これによって, 受信者の数や提供可能なコンテンツの品質が限定されてしまう. もう一方は, 受信者側の計算機資源と通信路の品質である. 発信者からインターネット上に分散する多くの受信者までの通信路はそれぞれ異なる性質をもつ. したがって, 単一的な品質のサービスでは, 受信者までのネットワーク環境あるいは受信者の計算機資源の多様性に適応できない.

本研究では, 「多くの受信者を対象とするコンテンツ配信を行なうコミュニティの活動を支援すること」を目標とし, 上述した問題点をオーバーレイ・マルチキャスト技術に階層符号化を適用する手法によって解決する. 本研究により「発信者が一般利用者にとって現実的な資源で, 多くの受信者に対し, 個々の受信者が要求する品質でサービスを提供できる環境」という理想的な環境を実現する.

1.2　目標環境の必要要件

第 1.1節で示した理想とする環境を実現するために必要な要件を以下に示す. 映像配信を行なう人々を発信者, 映像を受信し視聴を行なう人々を受信者とする.

1. **一般的なネットワーク, 計算機資源で配信網を構築**
   個人の発信者であっても, 一般利用者が持ちうる現実的な資源環境で配信網を構築できる. 資源環境とは, ネットワーク資源, 計算機資源を指し, 一般利用者が容易に保持できるも

のを対象とする. また, 発信者の運用コストに関しても一般利用者にとって許容できる範囲である必要がある.

2. **自由な参加, 脱退**
   受信者が自由に映像配信網に参加, 脱退できる. これには受信者が, 自分のネットワーク的な位置, ネットワーク資源, 計算機資源の環境に関わらず配信サービスを受けられることを意味する.

3. **受信者の要求に基づく映像品質**
   それぞれの受信者は, 所有する資源の範囲内であれば, 望んだ品質の映像を受信できる. 受信者の映像品質に対する要求は, 受信者の持つネットワーク資源, 計算機資源, もしくは映像への関心度等により常に変化する. 例として受信者が他のアプリケーションでネットワーク資源を消費することを予め認知している場合には, 受信者自ら映像品質を下げることができる. このように受信者は, 自らの要求に基づく映像品質で映像を視聴できる.

4. **受信者へ途切れの無いな映像を提供**
   受信者は, 発信者から提供される映像を途切れなく視聴できる. これは映像というメディアが時系列に連続するデータから構成されることに起因する. 映像の途切れが発生する理由として, 発信者のネットワークからの切断, ネットワークの輻輳等の理由が考えられる.

1.3 論文の構成

本論文は 8章から構成される. 2章で目標環境の実現における既存配信技術の問題点を述べる. 3章で, 既存の配信技術に替わるマルチキャスト技術であるオーバーレイ・マルチキャスト技術について述べる. 4章で新たなオーバーレイ・マルチキャストプロトコルである LOLCAST の提案を行う. 5章で LOLCAST プロトコルの設計を行い, 6章で LOLCAST システムの実装を行う. 7章では, 設計実装を行った LOLCAST の評価を行う. 最後に 8章において, LOLCAST プロトコルの総括と今後の課題を述べる.

第2章　既存配信技術の問題点

本節では, 既存の放送型の通信を実現する技術である, サーバークライアント型モデル, CDN, IP マルチキャストを取り上げる. それぞれの技術において, 1.1節で述べた目標環境の実現にあたって発生する問題を述べる.

2.1　サーバクライアント型モデル

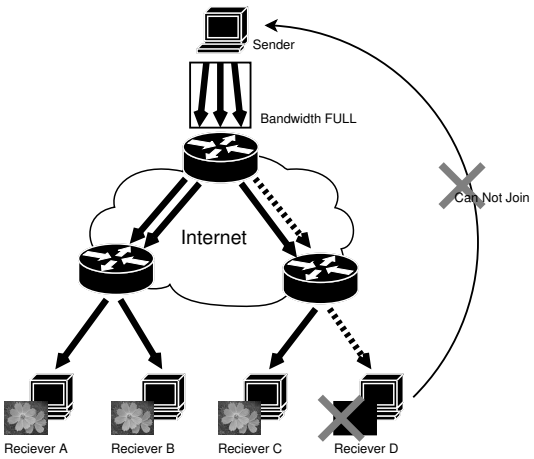

図 2.1: サーバクライアント型モデルにおける問題点

一般的に利用されている配信手法として, 複数のユニキャストにより通信を行うサーバクライアント型モデルが挙げられる. 特殊な配信環境を必要としない点から, 多くの映像配信サービスで利用されている. 図 2.1に示すようにサーバクライアント型モデルでは, 発信者からそれぞれの受信者に対し, ユニキャスト通信を用いてデータの配信が行われる. この際問題となるのが, 発信者は一般利用者であり, ネットワークの帯域的資源が限られている点である.

図 2.1において, 発信者は受信者 A,B,C に対し配信を行っている. 発信者の所持するネットワーク資源は受信者 A,B,C に配信を行っている状態ですでに枯渇しており, 新たに配信を希望する D は配信を受ける事ができない. これにより, 対象とできる受信者の数が限られてしまい, 多くの受信者を対象とした配信ができない.

上述したように, サーバークライアント型モデルを利用した配信では, 受信者の数に比例したネットワーク資源が必要である. よって, 現状で行われているサーバクライアント型モデルを利用した個人による配信は十数人を対象とした小規模なものや, 配信データの利用帯域を制限したものが殆どを占める.

2.2 CDN

CDN (Contents Distribution Network) は一対多モデルの配信元を複数箇所に分散させることにより配信元の負荷を軽減させ, 受信者に安定した映像を提供するための技術である. 具体的には, 配信元に存在するコンテンツを各 ISP に分散させたサーバに対してキャッシュし, 受信者に最適なサーバを伝達することによって配信の効率化を行なう. しかし, 一般の配信者は, システムを構築するコストの問題から CDN を気軽に映像配信に利用することができない. また, 各 ISP が CDN 網 に参加していることを前堤としており, 広域に分散している受信者に対して映像配信を行なうことは困難である.

2.3 IP マルチキャスト技術

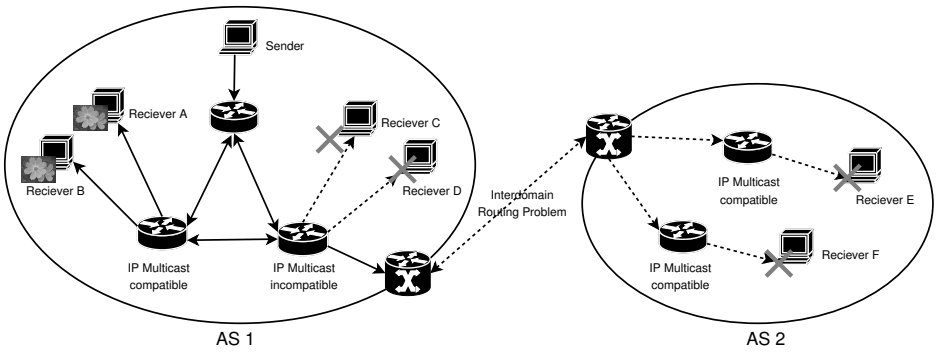

図 2.2: IP マルチキャストにおける問題点

従来考えられてきたグループ通信の手法として IP マルチキャスト技術がある. ネットワーク層で実現する IP マルチキャストを広域に適用するためには, マルチキャストアドレス予約, グループの管理, ISP 間のポリシーの違い, インタードメインルーティング, 拡張性, 全ルータのマルチキャスト対応といった諸問題を解決しなければならない [2].

例として, 図 2.2のように, 発信者は受信者 A,B,C,D,E に対し配信を行っている場合を考える. 受信者 C,D は IP マルチキャスト非対応なルータの下にいるため, 配信を受けられない. さらに, 受信者 E,F は AS 間のポリシーの違いにより, AS1 からの IP マルチキャストのトラフィックが流れず, 配信がされない.

上述した理由から IP マルチキャストは広域に分散して存在する発信者が自由にグループ通信を行うことができない. 現在, IP マルチキャスト技術は, 特定 ISP 内等の限られた範囲のネットワークでの映像配信サービスにおける利用に留まっている.

2.3.1 既存配信技術の限界

上述したようにインターネットにおいて放送型の通信を実現する技術はこれまでも研究・開発がさかんに行なわれてきた.

しかしこれらの技術は, 一般利用者が手軽に利用できるものではない. 1.2節で述べたように, 一般利用者はサーバクライアント型のモデルを利用してコンテンツ配信が可能である. しかし発信者側の帯域的資源限界から受信者の数が制限され, 自由な配信網の構築は不可能である. CDN

は運用に発生するコストの問題から自由に配信網が構築できない. また, CDN に参加していない ISP に位置する受信者を対象とできない. IP マルチキャスト技術はグループ通信を実現する技術として従来より研究が進められているが, 前述した技術的な問題, 運用上の問題からインターネットへの広域な適応が為されていない.

第3章　オーバーレイ・マルチキャスト技術

近年, IP マルチキャストの代替技術として, オーバーレイ・マルチキャスト技術が登場し, 多くの研究が進められている [3, 4, 5, 6, 7, 8, 9, 10, 11, 12].

図 3.1に示すように, オーバーレイ・マルチキャスト技術は, 実ネットワークの上で構築された論理的なトポロジであるオーバーレイ・ネットワーク上で, マルチキャストを実現する. このオーバーレイ・ネットワークはエンドホスト間のユニキャスト通信によって構築され, 通信基盤として利用される. これに対し, IP マルチキャストではルータを基礎とする IP ネットワークを通信基盤として利用する.

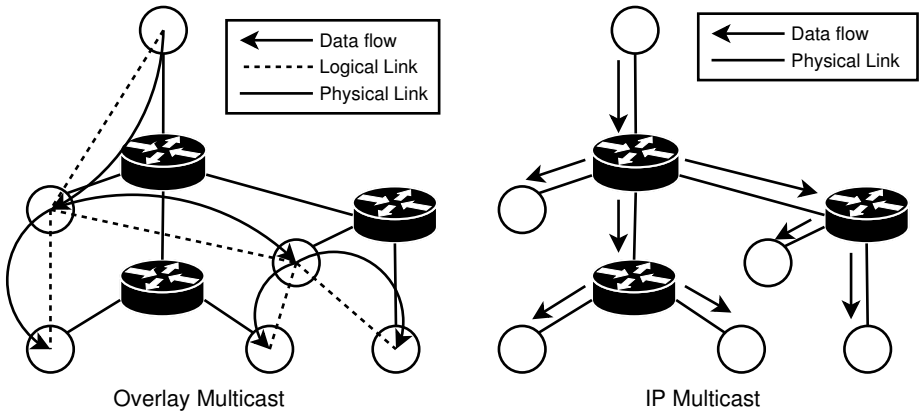

図 3.1: オーバーレイ・マルチキャストと IP マルチキャストの比較

マルチキャストを行なうためのボトルネックとなる処理としてデータの複製が挙げられる. IP マルチキャストではデータの複製をルータが行なう. これに対し, オーバーレイ・マルチキャストでは配信網に参加するそれぞれのエンドホストがデータの複製を行なう. データの配信元となるエンドホストは子となるエンドホストにデータを複製してもらうことにより配信に伴う負荷を抑えることができる.

一方で, 実ネットワーク上に論理的なトポロジを形成したことによるオーバーヘッドも存在する. まず, 実リンク上で重複したデータが転送される可能性がある. これは実ネットワーク上に被さる形で構築されたオーバーレイ・ネットワークからは実ネットワークの状態を考慮できないことに起因する. また通信基盤として利用しているエンドホストは, その稼働状況に関して保証ができないため, 予期せぬマルチキャストツリーの分断が起こりうる.

このようにオーバーレイ・マルチキャスト技術の特徴は, ネットワーク管理機能, データの複製, グループの管理機能の全てをアプリケーションレイヤなどの上位レイヤに任せる点にある.

オーバーレイ・マルチキャスト技術では, オーバーレイ・マルチキャストプロトコルによって上述したマルチキャストと同等の機能が実装されている. オーバーレイ・マルチキャストプロトコルについて述べる上で用語を以下の様に定義する. 配信網をマルチキャストツリー, 配信

網に参加しているエンドホストをノード, 配信元となるノードをソースノードとする.

3.1 オーバーレイ・マルチキャストプロトコル

既存研究として, 様々なオーバーレイ・マルチキャストプロトコルが提案されている. 主要な例として, ALMI[13], Narada[5, 6], Scattercast[14], Overcast[15], HMTP[9], Yoid[3], Hostcast[10], CAN[7], NICE[8] 等が挙げられる. 各々のオーバーレイ・マルチキャストプロトコルは異なる設計目標を基に考えられている. この設計目標は, 対象とするアプリケーションによって決定され, その特性を考慮した設計がされている. また, それぞれの設計目標によって, マルチキャストツリーの構成手法も異なる.

ALMI では構築したマルチキャストツリー上の各ノード間の遅延が最小となるマルチキャストツリーの構成を目標としている. これに対し Narada, Scattercast では, 各ノードとソースノード間の遅延が最小となるようにマルチキャストツリーを構成することを目標とする. 遅延以外のメトリックとして帯域が挙げられるが, Overcast ではソースノードから各ノードまでの利用可能な帯域が最大となるマルチキャストツリーの構成を目標としている. 最後に, Hostcast では, マルチキャストツリーの自体の安定性を高めることを目標としている.

3.1.1 参加ノードの体系化

オーバーレイ・マルチキャストプロトコルは 2 種類の論理的なトポロジをもつ [4]. 各ノードの管理を行なう Control Topology と実際のデータを送信するデータ転送のための Data Topology である. 図 3.2に両者の一例を示す. Control Topology の主な目的は, 各々のノードの状況を把握し, ノードの予期しない切断への対処を行なうことである. ノードの予期しない切断とは, マルチキャストツリーに参加するノードが, 計算機やネットワークの障害等によってマルチキャストツリーから決められた切断手順を踏まずに切断されてしまうことを表す. Data Topology は, Control Topology の一部である場合が多く, 実際のデータの流れを規定するために存在する. Control Topology は, その形態から Mesh と呼ばれることが多く, Data Topology は, Tree と呼ばれることが多い.

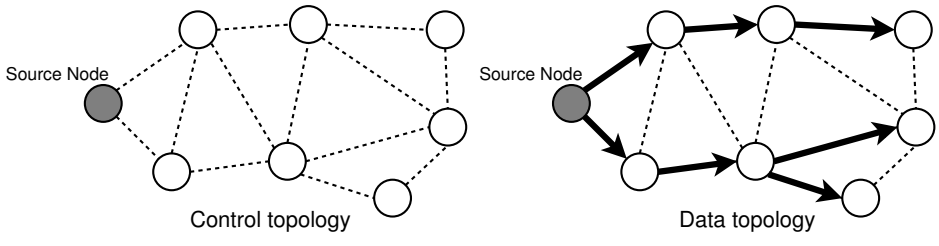

図 3.2: オーバーレイ・マルチキャストにおけるノードの体系化

3.1.2 マルチキャストツリーの構成手法

オーバーレイ・マルチキャストプロトコルは, 上述したトポロジの構成手法から 3 種類に大別可能である [4, 11, 16]. メッシュ状の制御トポロジを最初に形成する Mesh-first 型, 分散的にデータ転送の為のツリーを最初に形成する Tree-first (Direct) 型, 制御トポロジを何らかのメトリックに沿って形成する Implicit 型である.

Mesh-first 型

Mesh-first 型では, メッシュ状の Control Topology を最初に形成する. この際, 対となるノード同士に複数のパスが存在していてもよい. それぞれのノードは Control Topology 上でルーティングを行い, 分散的に一意なパスを決定することにより Data Topoloy を形成する. 多くのプロトコルにおいて, ソースノードから RPF (Reverse Path Forwarding) を行う事により Data Topology を形成している. Mesh-first 型のプロトコルのとして, Narada, Scattercast が挙げられる.

Tree-first 型

Tree-first 型では最初に分散的に Data Topology の形成を行う. 次にそれぞれノードが Data Topology 上で対となっていないノードを発見し追加的な制御用のパスを張ることにより Control Topology を形成する. Tree-first 型のプロトコルのとして, ALMI, Overcast, HMTP, Yoid, Hostcast が挙げられる.

Implicit 型

Implicit 型のプロトコルでは, ある特定のメトリックを用い Contorl Topology を形成する. Data Topology は規定された Control Topology により暗黙的に決定づけられる. そのため, Implicit 型のプロトコルでは Contorl Topology と Data Topology が同時に規定され, 追加的な処理を行わずにそれぞれのトポロジが形成される. Implicit 型のプロトコルとして, CAN, NICE が挙げられる.

3.2 既存オーバーレイ・マルチキャスト技術の問題点

3.1節で述べたように, 現在多くのオーバーレイ・マルチキャストプロトコルが提案されている. 一方で, 本研究の目標環境の一つである, 受信者の要求に基づく映像品質を提供する上で問題が生じる. それは, 単一的なデータの配信しか考えられていない点である.

映像や音声等のコンテンツは品質を制御可能である. 品質とは, コンテンツの情報量や, 解像度, 画質等を指す. 現状のコンテンツ配信においても, 受信者の品質に対する様々な要求に応じるために複数の品質でこれらのコンテンツを提供している. この例を図 3.3に示す. 映画の予告編を配信している Web サイトでは ”Mid”, ”High”, ”光” のように受信者の資源環境を回線の種類によって抽象化し提供を行っている. インターネット・ラジオのような音声配信を行うサー

図 3.3: 既存配信サービスにおける映像音声の品質制御

ビスでは, ”24kbps”, ”56kpbs”, ”96kbps”のように受信者の帯域を指標として利用し, 複数品質でのサービスを行っている.

ここでの品質に対する要求は, 受信者の計算機環境, ネットワーク環境や, 受信者の品質要求に影響される. このように, 受信者のコンテンツの品質に対する要求は様々である.

現状のオーバーレイ・マルチキャストプロトコルを利用し, 複数品質のサービスを提供する場合, 発信者は受信者の品質に対する様々な要求に応じるために, 品質毎にマルチキャストツリーを構築することになる. これには発信者への帯域的な負担, 複数のマルチキャストツリーの管理に対するオーバーヘッドの増加が伴う. また, 図 3.4に示すように発信者は, 無線端末のような狭帯域なノードから FTTH を利用しているような広帯域なノードまで多種の資源環境に適応した配信を行わねばならない. さらに, 受信者の要求に基づいた配信を行なうためには品質を選択可能でなければならない. よって, ソースノードは一種類のコンテンツに対し品質毎に資源環境を要求され, 現実的ではない.

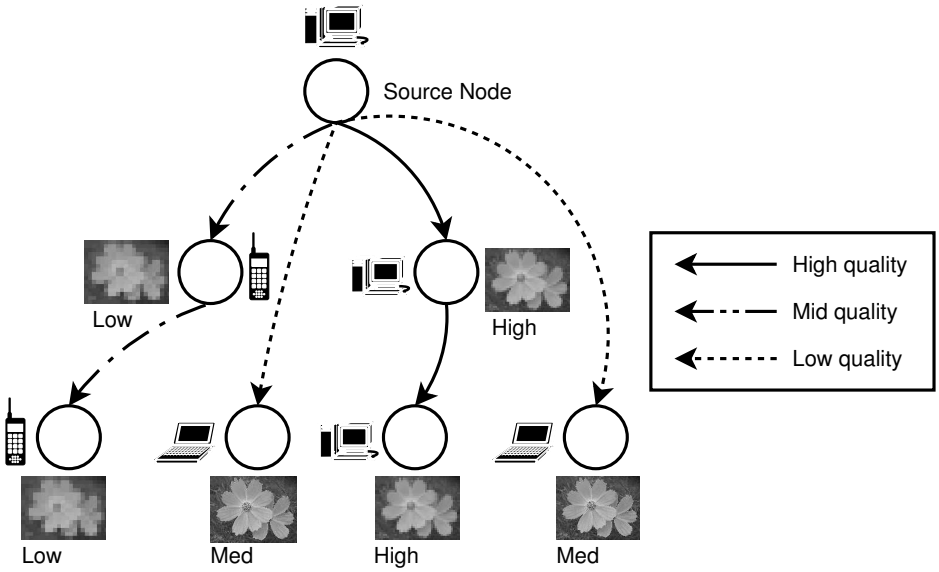

図 3.4: 既存オーバーレイ・マルチキャスト技術を利用した配信に関わる問題

第4章　LOLCASTの提案

本節では 3.2節で述べた問題点を解決し目標環境を実現するため, 階層的な構造を持つデータの配信を行う新たなオーバーレイ・マルチキャストプロトコルの提案を行う. 次に LOLCAST で用いる階層的な構造を持つデータの解説を行い, LOLCAST により構築されるマルチキャストツリーについて述べる. 最後に LOLCAST により新たに提供可能となる機能を述べる.

4.1　階層的な構造を持つデータ

本節では,LOLCAST で利用する, 階層的な構造を持つデータについて述べる. 階層構造を持つデータの例として階層符号化された映像データを取り上げる.

4.1.1　階層符号化

階層符号化とは, 解像度等を変化させた画像を階層的に複数用意し, 画像の階層数に応じて品質を選択できる符号化方式である. それぞれの階層ごとに符号化を行ない, 下位レイヤを補完する形で上位レイヤが存在する. 画像は, 一つの Base レイヤと複数の Enhanced レイヤに分割される. 最低限の画像を提供する階層として Base レイヤがある. Base レイヤに Enhanced レイヤを追加することにより, より良い映像品質を持つ画像を取得できる. 図 4.1 に示すようにレイヤの数によって LOW から HIGH にかけて, 映像品質が上がっていく. 階層符号化を利用した映像メディアの例として MPEG2 SNR Scalable/Spatial Scalable Profile[17], JPEG2000 EBCOT (Embedded Block Coding with Optimized Truncation)[18, 19] の二つが挙げられる.

4.2　LOLCAST 概要

本プロトコルにより提供される映像等のサービスは複数のレイヤで表現される. サービスを受ける各ノードは品質の要求に応じて受信するレイヤ数を指定する. そして, 発信者であるソースノードを頂点とし, 多くのレイヤを要求するノードが上位に位置するようなマルチキャストツリーを構築する.

本プロトコルを利用した際の特徴は, マルチキャストツリーに参加する各ノードの所持するサービスの品質がそれぞれ異なる点である. 各ノードは品質要求を行なうことで, 望んだ品質でサービスを受けられる. 階層的な構造を持つデータの利用により, ソースノードは品質毎のマルチキャストツリーを管理する必要がなく, データの送信も最高品質のデータを一本行えば良い. よって, 品質に対してデータを冗長に送る必要が無くなり既存技術と比べネットワークの帯域的に有利な配信が行える. さらにこの特徴を利用することで新たに, Base レイヤを用いたマルチキャストツリーの効率的な冗長化やレイヤの間引きによる輻輳回避の機能を提供できる.

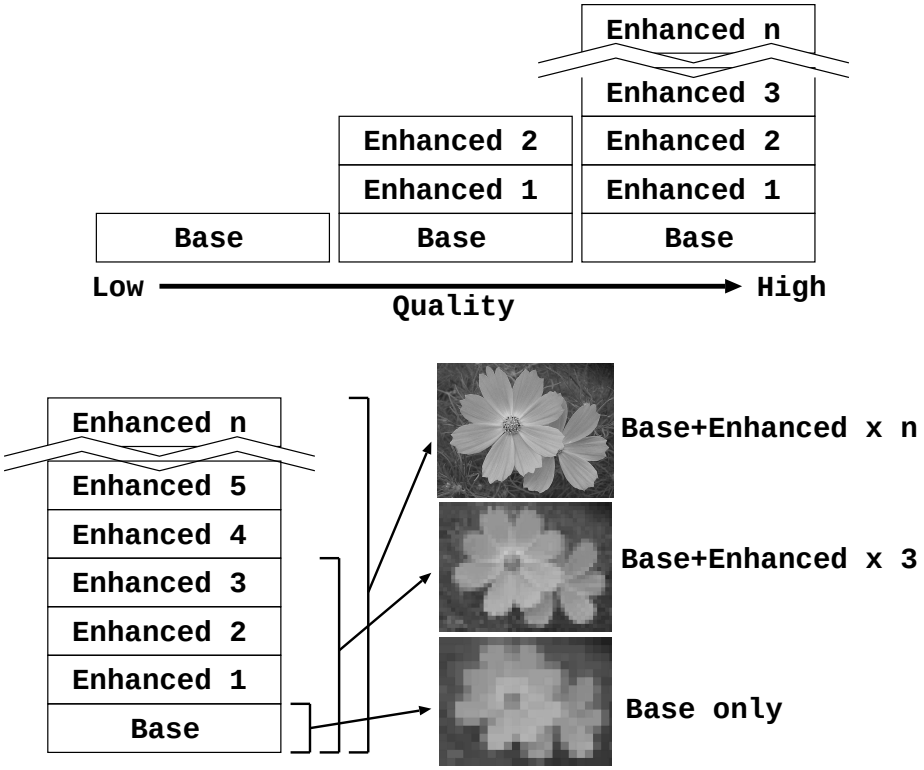

図 4.1: 階層符号化方式を利用した際の映像の変化

本プロトコルを利用することで,「それぞれのノードの要求した, あるいはそれの資源環境に適応した複数の品質」を単一のマルチキャストツリーで提供できる.

4.3 マルチキャストツリーの構成

本プロトコルの設計目標は「マルチキャストツリーに参加する各ノードによる映像の品質の制御が可能」なことである. そこで, 本プロトコル (LOLCAST) は各ノードの要求するレイヤ数をメトリックとしたマルチキャストツリーを構築する. LOLCAST はレイヤという単位で各ノードの所持する情報量に差異が存在するため, メッシュ状の Control Topology を構成できない. よって Tree-first 型のマルチキャストツリーの構成手法を用いた.

4.3.1 所持レイヤ数による深さの決定

図 4.2に, 本プロトコルを利用したマルチキャストツリーの一例を示す.

Data Source Node と記されている A がソースノードである. ノード内に記されている値は各ノードが持つレイヤ数に基づく映像の品質である. 値が大きい程レイヤ数が多く, 高品質を求めるノードに映像を転送できる. ノード A の 10 レイヤの品質を持つ映像がソースノードの送信する最高品質の映像である. ソースノードである A は, 受信ノードの要求にしたがって映像配信を行なう.

マルチキャストツリーに参加を行うノードは, 自らの要求レイヤ数以上のレイヤ数を保持しているノードからの配信を受ける. 各ノードは親と子の関係を持ち, 親は子に, 子の要求する品

質を持つ映像を提供する. また, 各ノードは親となるノードが受信している品質までの映像を配信できる. この際, マルチキャストツリーに変更を加える必要は無い. 例として, E は 親となる B から 6 レイヤの品質を持つ映像を受信している. よって, 子となる H, I には 6 レイヤまでの品質で映像を送信できる. H や F の下にはさらに伸びるツリーがあると仮定する. G や I は, 2 や 1 レイヤの品質を持つ映像を要求するノードが存在しないためツリー上の末端のノードとなる. 無線端末等の狭帯域のネットワークしか持たないノードは, これらのノード G, I の様に末端のノードとなる.

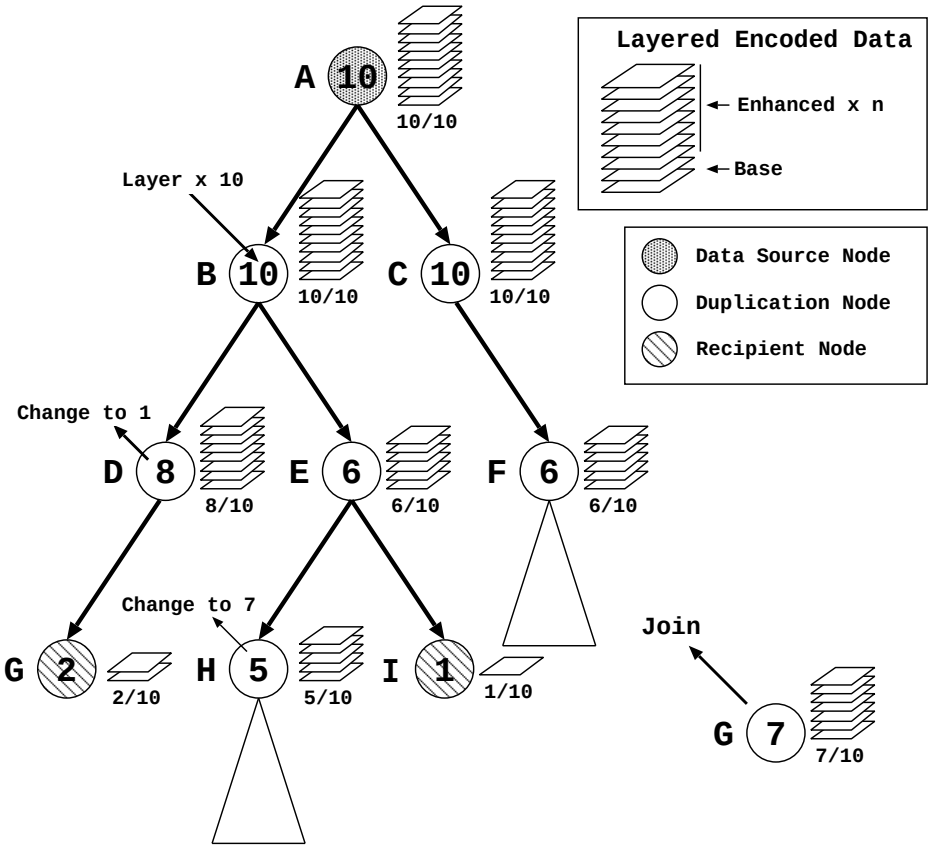

図 4.2: マルチキャストツリーの一例

4.4 実現する機能

LOLCAST により新たにレイヤという単位で配信が行える. これにより, 発信者は個々の受信者の資源環境を意識すること無く適切な配信を行える. この機能の他に, レイヤに分割したことによる新たな機能を提供できる.

4.4.1 ノード障害時におけるデータの保証

不安定な通信基盤上で動作するオーバーレイ・マルチキャストにおいて, 予期せぬノードの障害からマルチキャストツリーを復旧することが必要である. 本節では既存のオーバーレイ・マルチキャストプロトコルが用いる復旧手法を紹介する. また, 本プロトコルの階層符号化の

利点を活かした新たなノード障害時の効率的な復旧手法を提案する.

ノードの予期せぬ切断であっても, 切断の行なわれたノードの下位に位置するノードは, 継続してサービスを受けられる必要がある. ALMI[13] では, 特にノードの障害時における効果的な手法は存在せず, マルチキャストツリーから分断されたノード群は参加の手続を再度行ないマルチキャストツリーへの復旧を行なう. Narada[5] は Mesh-first 型のプロトコルであり, Contorl Topology 上でリンクを付け足すことによって復旧を行っている.

Host Cast のパス冗長化手法

これに対し, 予期しない切断に対する対処法として, 図 4.3 に示す Host Cast[10] のパス冗長化手法がある. この手法では, 親ノードの冗長化により, 複数の配信パスを事前に確保する.

図 4.3 において, B が予期せぬ切断によって, マルチキャストツリーから切断した際の E での処理の例を示す. 定常状態においては, E に対して $A-B-E$ というパスを利用して配信が行なわれている. この時, 冗長化を行なうパスとして, E の親である B と対等な位置関係を持つ C を経由する $A-C-E$, E の親である B のさらに上位ノードである A を経由する $A-E$ を Control Topology 上に確保している. E は B の切断時にこの冗長パスに切り替えることによって, 予期しない切断時における Data Topology の復旧までの時間を短縮する. これにより, ノードの予期しない切断の影響を受けるノードのマルチキャストツリーからの切断時間を抑えることができる. しかしながら, Host Cast の手法は, Control Topology 上で冗長化を行なっているため, マルチキャストツリーが収束するまでの時間が発生する.

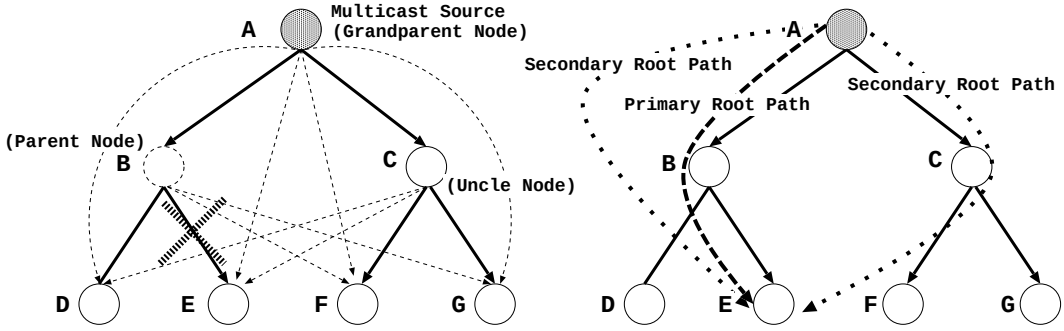

図 4.3: Host Cast における親ノードの冗長化手法

Base レイヤを利用したパス冗長化手法

本プロトコルでは, 階層符号化の利点を活かし, Base レイヤを利用したノード障害時における効率的な保証手法を提案する. 4.1 節で述べたように, 階層符号化において Base レイヤとは最低限の映像品質を保証するレイヤを指す. Host Cast の手法では Control Topology 上でパスの冗長化を行なっているが本手法では Data Topology 上でパスの冗長化を行なう.

図 4.4において, B が予期しない切断によって, マルチキャストツリーから切断した際の D での処理の例を示す. 定常状態では, $A-B-D$ というパスを利用し D に配信が行なわれている. この時, D は A, C から Data Topology 上で冗長的に Base レイヤのみを受け取っている. B の切断が起こると $A-B-D$ というパスが無効となり, D への配信が停止する. D は B の切断を検知すると即座に A もしくは C から最低限の品質を持つ Base レイヤへの映像に

切り替え, 情報の損失を防ぐ. 冗長化されたパスから Base レイヤの供給を受けつつ, 上述した HostCast 等の既存のオーバレイ・マルチキャストプロトコルの手法を利用しマルチキャストツリーの復旧を行なう. マルチキャストツリーの復旧を終えると, D は新たな親ノードとなる C から要求した品質での映像を受信する.

既存のオーバーレイ・マルチキャストプロトコルでは単一的なデータの配信しか考えられていない. そのため, Data Topology 上で冗長化を行なうにはフルレートの映像をメッシュ状に提供する必要がある. これは各ノードへの帯域的な負荷が非常に大きいため現実的でない. 本プロトコルでは Base レイヤのみを利用するため各ノードへの影響を抑え, 映像の復旧を効率的に行なえる.

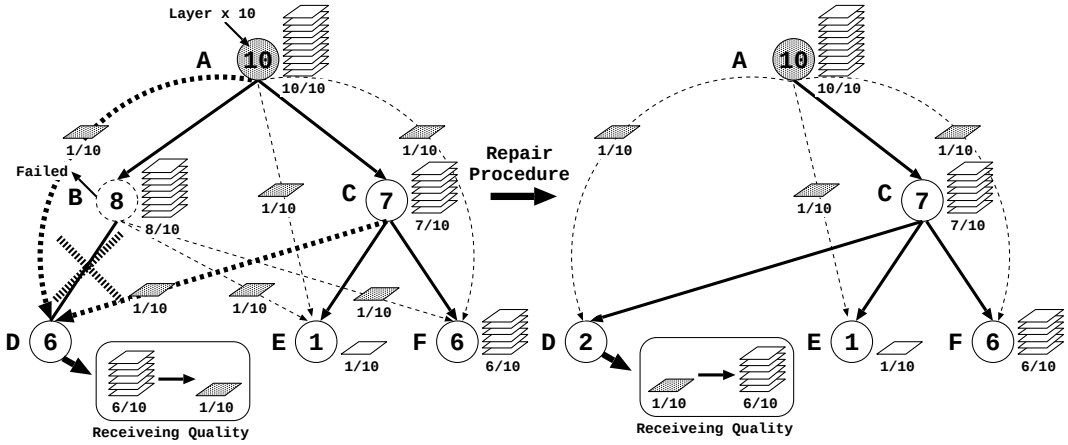

図 4.4: Base レイヤを利用した Data Topology での冗長化

4.4.2 要求品質の幅を利用した輻輳回避

LOLCAST においてデータの品質を自由に制御できる特徴を活かし, 輻輳回避を行なうことができる. 各ノードの要求する映像品質, つまりレイヤ数に幅を持たせ, 輻輳が発生した際に優先的に上位レイヤのデータを破棄することで, 輻輳の回避を行なう. この例を図 4.5 に示す.

B は A からのデータを E に中継している. B から E への通信路に輻輳が起きていない場合, E は要求した最高品質の映像が得られる. しかし, B が何らかの方法で輻輳を検知した場合, B は上位レイヤから優先的に中継を止め, 輻輳を避ける. 輻輳の検知手法に関しては, 既存研究 [20] を参考にする.

以上のように, レイヤ数の幅によって各ノードからの品質要求に柔軟性を持たせることができる. 要求レイヤ数の幅が狭い場合は要求レイヤ分のサービスを受けられるが, これを満たせない場合には配信が受けられない. 要求レイヤ数の幅が広い場合は, マルチキャストツリーへの変更は発生しにくく, 安定したサービスを受けられる. ただし, 受信品質が上下する可能性がある.

要求レイヤ数の幅を狭く取ることで,「要求品質以下のサービスでは見る意味が無い」と考える利用者を対象とし, 要求レイヤ数の幅を広く取ることで,「品質が落ちても継続してもサービスを受けたい」と考える利用者を対象とできる.

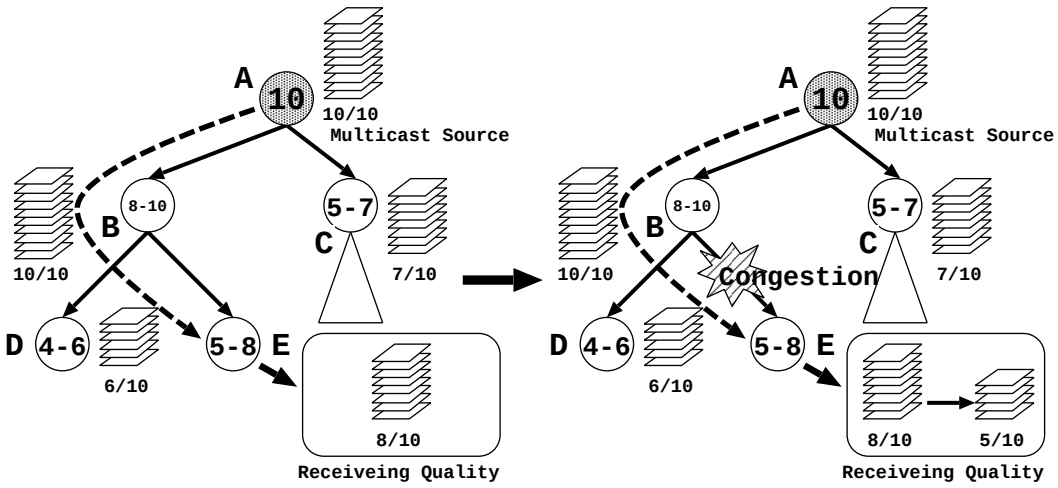

図 4.5: 要求品質の幅を利用した輻輳回避

第5章　LOLCASTプロトコルの設計

5.1　LOLCAST設計概要

LOLCASTではソースノードが全ノードの情報を保持し, マルチキャストツリーの管理を行う. 参加ノードはマルチキャストツリーの変更をソースノードにメッセージングし, その更新を行う. ノード間のすべての情報のやり取りはメッセージパッシングにより行われる. メッセージを受け取ったノードは自身の保持するデータ構造に更新を加え, 必要であれば返信となるメッセージを送る.

5.2　想定利用環境

LOLCASTの想定利用環境を述べる. 規模は数千人を対象とした. これは1.1述べたインターネット上で活動する人々の属するコミュニティの大きさを考えた際に, 個人レベルで運営されている最大のコミュニティでも数千人程度であると考え, この規模を想定する. この際それぞれの参加者の所持するネットワーク資源や, 計算機資源は一般的なものとする.

5.2.1　ノードの定義

本節ではLOLCASTで用いるノード同士の関係の定義を行う. 図5.1でノード同士の関係の一例を示す.

マルチキャストツリーに参加するノードは2種類に分けられる. Source NodeとRecipient Nodeである. Source Nodeとは図中のNode$A$にあたり, マルチキャストツリーの配信元となるノードを指す. この他のノード B, C, D はSource Nodeから配信されるデータを受け取るノードをRecipient Nodeとする.

図5.1においてノード D がマルチキャストツリー上で上位に位置するノード B からデータを受け取っていた際, ノード D はノード B のChild Nodeと定義される. これに対応して, ノード B はノード D のParent Nodeと定義される. マルチキャストツリーに新たに参加するノード E をNew Nodeと定義され, ノード E のParent Nodeとなる条件を満たしているノード A, B, C をPotential Parent Nodeと定義する. 最後に, マルチキャストツリーから離脱を行うノード B をLeave Nodeと定義する

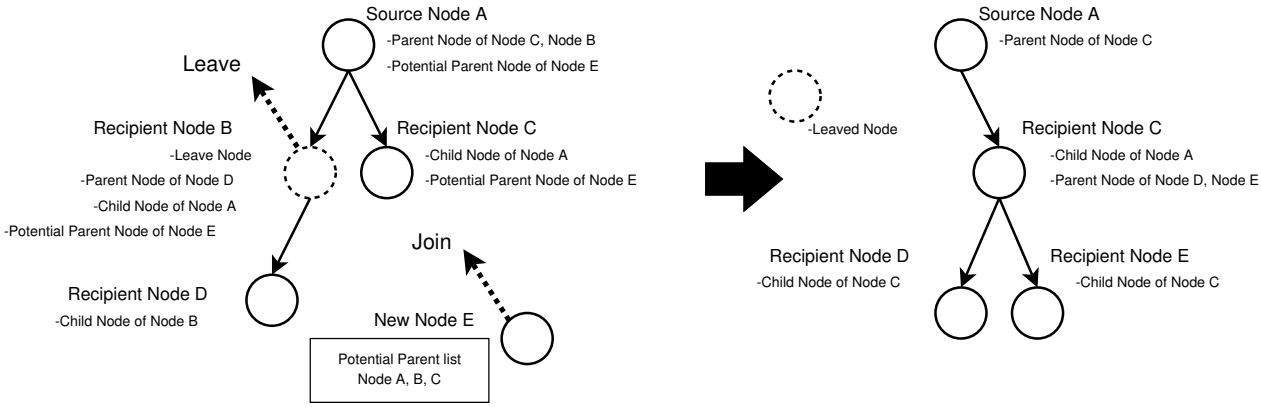

図 5.1: LOLCAST におけるノードの定義

5.2.2 各ノードの保持する情報

5.2.1節で述べたように, マルチキャストツリーに参加するノードは Source Node と Recipient Node の 2 種類に分けられる. 本節ではそれぞれのノードの保持している情報について述べる.

ノード情報

ノード情報は各ノードの LOLCAST プロトコルにおける基本情報を格納している. ノード情報は LOLCAST で行われるすべての通信においてノードを特定する情報として利用される.

1. ノード識別子
   特定のマルチキャストツリーに参加しているノードに対し割り振られる一意な識別子. ノードの検索の際にキーとして用いられ, 割り当ては Source Node によって行われる.
2. 要求保持レイヤ数
   ノードの要求, 又は保持しているレイヤ数を示す. Source Node の保持するレイヤ数が最大の値となり, 1 レイヤが最小値となる.
3. 対応するネットワーク層プロトコル
   ノードの利用可能なネットワーク層のプロトコルを示す.
4. アドレスリスト
   利用可能なネットワーク層プロトコル毎のアドレスが格納されているリスト.

ツリーノード情報

ツリー構造において管理されるノードの情報は基本情報であるノード情報の他にマルチキャストツリーの保持を行う際に必要な情報が含まれる.

1. ノード情報

2. マルチキャストツリーにおけるノードの深さ
3. Parent Node のツリーノード情報
4. Child Node のノード識別子のリスト

ツリー構造

マルチキャストツリーの状態を保持するツリー構造には以下の情報が含まれる.

1. Source Node のツリーノード情報
2. 全ノードのマップ
   全ノードの登録されているマップ. ノード識別子をキーとして持ち, ツリーノード情報をデータとして持つ. 主に検索に用いられる
3. Leave Node のノード識別子のリスト
   マルチキャストツリーからの離脱処理を行っているノードのノード識別子のリストである. Leave Node の状態をロックするために用いられる.

Source Node 固有情報

Source Node は上述したツリー構造によりマルチキャストツリーの維持を行っている.

1. Source Node 自身のノード情報
2. 全ノードのツリーノード情報を含むツリー構造
3. ノード識別子とツリーノード情報のマップ
   ノード識別子を利用し, その識別子を持つノード情報にアクセスするためのマップ. 主にノードの検索に利用される.
4. Child Node のリスト
   自身が Parent Node となっている Child Node のノード情報を含む.

Recipient Node 固有情報

Recipient Node の所持する情報は以下になる.

1. 自身のノード情報
2. Source Node のノード情報
3. Parent Node のノード情報
4. Old Parent Node のノード情報
5. Child Node のリスト
   自身が Parent Node となっている Child Node のノード情報を含む.

6. Potential Parent (PPL) のリスト
   Source Node より受信した PPL を一時的に格納しておくために存在する.

5.2.3 メッセージ群

LOLCAST はメッセージパッシングにより, ノード間の情報のやり取りを行う. メッセージはその機能別にいくつかのメッセージ群に分けられる. 本節では LOLCAST で利用するメッセージ群について述べる. メッセージは, Rendezvous Message 群, PPL Message 群, Join Message 群, Data Message 群, Leave Message 群, Prune Message 群, Notify Message 群の 7 つの群に分けられる.

Rendezvous Message

Rendezvous Message 群は New Node がマルチキャストツリーに参加するために必要となる情報をやり取りするためのメッセージ群である.

- **Rendezvous Request (Recipient Node to Source Node)**
  マルチキャストツリーに新たに参加するノードは Source Node に対し, Rendezvous Request を送る事によりマルチキャストツリーへの参加に必要な Source Node の情報を要求する. メッセージには, Recipient Node のノード情報が含まれる. メッセージを受け取った Source Node は対応するネットワーク層プロトコルを比較し, 満たしていれば Rendezvous Accept, 対応していなければ Rendezvous Reject を Recipient Node に返す.

- **Rendezvous Accept (Source Node to Recipient Node)**
  Source Node は新たに Recipient Node のノード識別子を生成し, 自身のノード情報と共に Recipient Node に送信する. メッセージを受け取った Recipient Node はノード識別子を自身のノード情報に格納し, Source Node のノード情報を保持する.

- **Rendezvous Reject (Source Node to Recipient Node)**
  Rendezvous Request を受けたノードが要求を受け入れられない際に返却されるメッセージである. Rendezvous Reject を受けた Recipient Node は, 新たな Source Node を探し Rendezvous Request を送る必要がある.

PPL Message

PPL Message 群は, Potential Parent のリスト構造である Potential Parent List (PPL) の送受信を行うためにある. PPL については, 5.2.5 節にて述べる.

- **PPL Request (Recipient Node to Source Node)**
  Recipient Node は PPL Request によって Source Node に対して PPL の要求を行う. メッセージには Recipient Node のノード情報が含まれる. PPL Request を受け取った Source Node は, PPL を PPL Data によって Recipient Node に対し送る.

- **PPL Data (Source Node to Recipient Node)**
  Source Node は PPL Request によって取得した Recipient Node のノード情報を用い

PPL を自身の保持するツリー構造を利用し生成され Recipient Node に送信する．この際 Recipient Node の要求レイヤ数, ツリーの深さ, Child Node の数を利用し優先度順に並べられた PPL が作られる．PPL を受信した Recipient Node は自身のノード情報に PPL を格納する．

Join Message

Join Message 群は PPL Message 群により PPL を入手したノードが Potential Parent に対し, 自身を子ノードとして受け入れるよう要求を行うメッセージ群である．

- **Join Request (Child Node to Potential Parent Node)**
  Child Node は PPL に格納されている最も優先度の高い Potential Parent Node に対し Join Request を送る事で自身を子ノードとして追加するよう要求する．メッセージには Child Node のノード情報が含まれる．送信後に PPL から Potential Parent Node の情報を取り除く．Join Request を受け取った Potential Parent Node は自身のノード情報とメッセージに含まれる Child Node のノード情報から保持レイヤ数が要求レイヤ数を満たしているか, 管理子ノード数に空きがあるかを検証する．要求を満たしていれば, Child Node のノード情報が Potential Parent Node の子ノード情報のリストに追加され, Join Accept を Child Node に送信する．参加が不可能であれば Join Reject を Child Node に送信する．

- **Join Accept (Potential Parent Node to Child Node)**
  Potential Parent Node は自身のノード情報と共に Join Accept を Child Node に送る．Join Accept を受信した Child Node は Potential Parent Node のノード情報を自身のノード情報の Parent Node のノード情報に格納する．

- **Join Reject (Potential Parent Node to Child Node)**
  Potential Parent Node は Join Reject によって Child Node に親ノードとなる要求を拒否した事を伝える．Join Reject を受信した Child Node は次に優先度の高い Potential Parent Node に対し Join Request を送信する．

Data Message

Data Message 群はデータの送信開始要求, 送信停止要求を行う．

- **Data StartRequest (Child Node to Parent Node)**
  Child Node は Data StartRequest により Parent Node に対しデータの送信開始要求を行う．ChildNode のノード情報の含まれたメッセージが Parent Node に送信される．Data StartRequest を受信した Parent Node は Child Node のノード情報のアドレスリストに含まれるアドレスに対しデータの送信を開始する．

- **Data StopRequest (Child Node to Parent Node)**
  Child Node は Data StopRequest により Parent Node に対しデータの送信停止要求を行う．ChildNode のノード情報の含まれたメッセージが Parent Node に送信される．Data

StopRequest を受信した Parent Node は Child Node のノード情報のアドレスリストに含まれるアドレスに対するデータの送信を停止する.

Leave Message

Leave Message はマルチキャストツリーから離脱を行う Parent Node が自身の Child Node に対し, 自身の離脱を広告するメッセージ群である.

- **Leave Request (Parent Node to Child Node)**
  マルチキャストツリーからの離脱を行う Parent Node は Leave Request により Child Node に対し, 他の Parent Node への移動を促す. Parent Node は自身の保持する子ノードのリストに含まれる全 Child Node に対してメッセージを送信する. メッセージを受信した Child Node は Parent Node のノード情報を Old Parent Node のノード情報に一時的に格納する.

- **Leave Complete (Child Node to Parent Node)**
  Leave Request を受けて他の Parent Node に移動した Child Node が, 以前の Parent Node に対し移動が完了した事を伝える. メッセージには Child Node のノード情報が含まれる. メッセージを受信した Parent Node は, 子ノードのリストから Child Node のノード情報を取り除く.

Prune Message

Prune Message は Child Node が Parent Node に対して, 自身を子ノードから取り除く要求するためのメッセージ群である.

- **Prune Request (Child Node to Parent Node)**
  子ノードを一つも保持していない Child Node が, Parent Node に対して Parent Node の子ノードのリストから取り除くように要求する. メッセージには Child Node のノード情報が含まれる. メッセージを受信した Parent Node は, 子ノードのリストから Child Node のノード情報を取り除き, Prune Complete を Child Node に送信する.

- **Prune Complete (Parent Node to Child Node)**
  Prune Complete は Parent Node が Child Node に対し, 子ノードのリストから Child Node の情報を取り除いた事を伝えるためのメッセージである. メッセージを受け取った Child Node は Source Node の保持するマルチキャストツリーの更新を行うため, Notify LeaveComplete を送る.

Notify Message

マルチキャストツリーの情報は Source Node が管理している. Join/Leave 等によってこのマルチキャストツリーに対して変更が成された場合に Source Node に対してマルチキャストツリーの更新を要求する必要がある.

- **Notify JoinComplete (Recipient Node to Source Node)**
  JoinCompleteは Parent Nodeからのデータの受信を開始した Recipient Nodeがマルチキャストツリーに参加したことを Source Nodeに伝えるためのメッセージである．メッセージには現在の Parent Node のノード情報と自身のノード情報が含まれる．Source Nodeは Recipient Node の Parent Nodeをツリー構造から検索し, その子ノードとして Recipient Nodeを追加する．

- **Notify LeaveProgress (Recipient Node to Source Node)**
  マルチキャストツリーからの離脱を行う Recipient Nodeが自身が離脱処理中である事を Source Nodeに伝えるためのメッセージである．メッセージには Recipient Node のノード情報が含まれる．メッセージを受信した Source Nodeは Recipient Node のノード情報を Leaveが進行中であるノードのリストに格納する．

- **Notify ParentChanged (Recipient Node to Source Node)**
  親ノードのマルチキャストツリーからの離脱等により, 親ノードの変更がされた Recipient Nodeが Source Nodeにマルチキャストツリーの変更を報告するためのメッセージである．メッセージには, 現在の Parent Node のノード情報と自身のノード情報が含まれる．メッセージを受信した Source Nodeは Parent Node のノード情報と Recipient Node のノード情報を利用し, 以前の Parent Nodeから現在の Parent Nodeへ Recipient Node の位置をマルチキャストツリー上で更新する．

5.2.4 メッセージフォーマット

LOLCASTで用いられる各メッセージは図 5.2に示すのフォーマットに格納される．

Message Type, **Message Subtype**により 5.2.3節で述べたメッセージの種類を規定する．**Message Length**, **Data Length**にはそれぞれメッセージのサイズとメッセージに含まれるデータのサイズが格納されている．次に送信元ノードのノード情報としてそのノード情報が格納される．**Message Data**はメッセージに含まれるデータを格納する可変長のフィールドである．以上のフィールドからメッセージが構成される．

5.2.5 マルチキャスト・ツリーの構成

本節では, LOLCASTの核となる, マルチキャストツリーの構成方法について述べる．

パラメータの定義

マルチキャストツリーの構成を行う上で利用されるパラメータを挙げる．

Source Node S, Recipient Node $R_1, R_2, R_3, ...R_n$, Number of layer L_n, Number of children node(s) C_n, Depth D_n

PPL Extraction

Source Nodeが全ノードの情報を保持し, マルチキャストツリーを管理する LOLCASTにおいて, Source Nodeによる Parent Nodeの決定が大きくマルチキャストツリーの品質を左右す

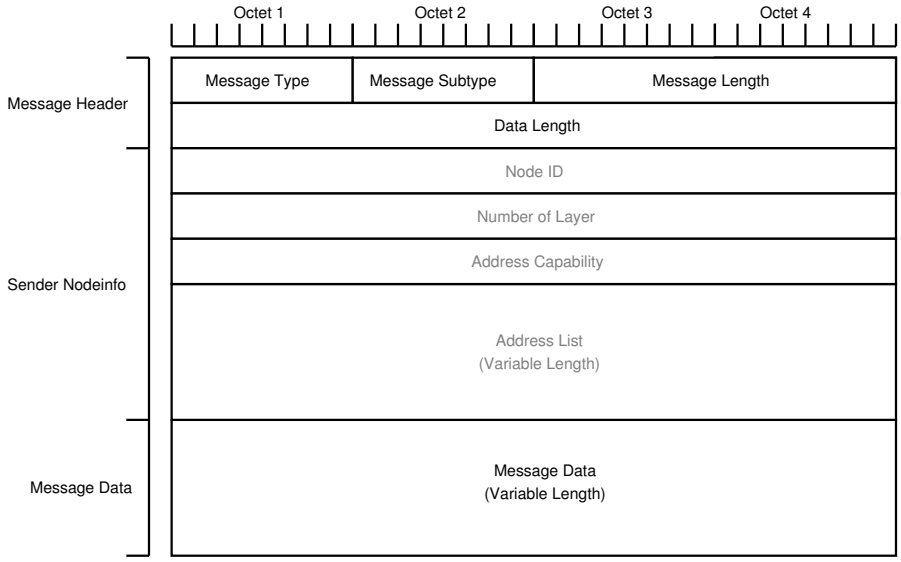

図 5.2: メッセージフォーマット

る. 本節では, 最適な Potential Parent Node の選択をし, Potential Parent List の抽出を行う PPL Extraction を図 5.3を用いて解説する.

マルチキャストツリーに参加する全ノードを比較し最適なノードを探索する手法では, 探索のオーバーヘッドが非常に高くなってしまう. よって, 一定量のノードの中から最適なノードを選択する手法を用いた. PPL Extraction は 2 つのフェーズにより構成される. Temporary PPL Extraction と PPL Extraction である. Temporary PPL Extraction により, 上述した一定量のノードを抜き出し, PPL Extraction により抜き出されたノードの中から最適なノードを選出する. 比較を行うノード数を示すパラメータを設け, その変更が可能である.

Potential Parent Node は以下の条件を満たす必要がある.

- Child Node の要求レイヤ数を満たしている
- Leave Node のリストに含まれていない
- Child Node 数に空きがある

以上の条件を満たすノードから, マルチキャストツリーにおける深さが最小となるノードから優先度が高い Potential Parent Node として, PPL に格納される.

Temporary PPL Extraction の流れを述べる. まず, ツリー構造内のノード識別子とツリーノード情報のマップよりツリーノード情報を抜き出す. 抜き出したツリーノード情報の所持レイヤと Child Node の要求レイヤ数の比較を行う. さらに, そのノードの Child Node 数に空きがあるかの照合を行う. 以上の条件を満たすノードのツリーノード情報を Temporary PPL に追加する. 以上の処理を上述した一定量を満たすまで続けられる. Temporary PPL が一定量を満たした時点で, PPL Extraction に引き渡す.

PPL Extraction では引き渡された Temporary PPL に含まれるノードより, 最適なノードの選択を行う. まず Temporary PPL に格納されているノードが Leave Node のリストに含まれていないか照合を行う. Child Node のリストに含まれていないノードは PPL に追加される. 以上の処理が Temporary PPL が空になるまで続けられる. Temporary PPL が空になった状態で

PPL のエントリが 1 つも存在しない場合は再度 Temporary PPL Extraction が行われる. PPL にエントリが存在する場合は PPL に含まれるノードは, マルチキャストツリーにおける深さが浅い順に並び替えられる. 以上で PPL Extraction は完了する.

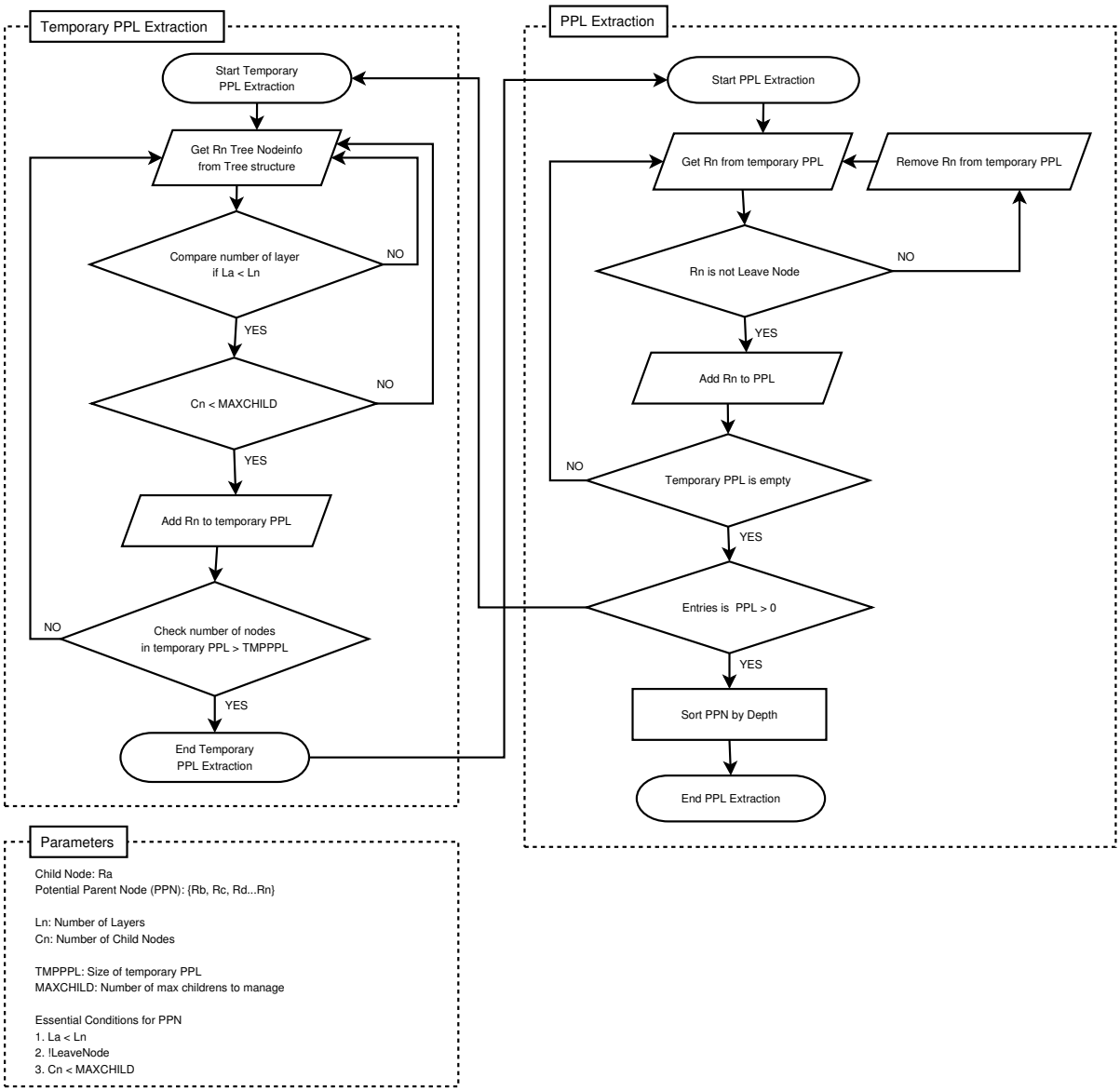

図 5.3: PPL Extraction の流れ

Rendezvouz Procedure

New Node はマルチキャストツリーに参加するために Source Node の情報が必要となる. Source Node の情報は図 5.4に示すように, Web ページ/メールや人づてに伝えられる. Source Node の情報の入手は以上のような処理によって行うことを想定するため, プロトコルの処理に

は含まれない.

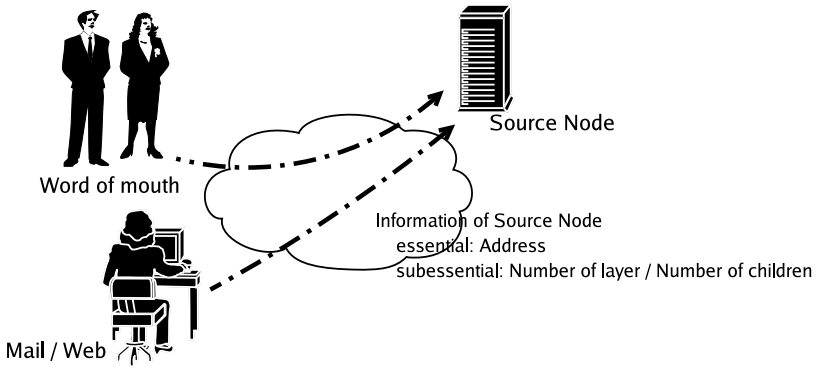

図 5.4: Rendezvous Procedure

Join Procedure

本節では, New Nodeがマルチキャストツリーに参加する際の手順を図 5.5, 5.6, 5.6, その際行われるメッセージパッシングの様子を図 5.8を用い解説する. 例で用いるマルチキャストツリーには S_a, R_b, R_c, R_d の 4 ノードが参加しており, S_a が Source Node となっている. このようなマルチキャストツリーがあった際に, New NodeR_e が新たにマルチキャストツリーに参加するすることを想定する. また, 各ノードの管理する Child Node の最大数を 2 として進める.

1. 新たにマルチキャストツリーに参加する R_e は 5.2.5節で述べた Rendezvous Procedure により取得した Source NodeS_a のノード情報を元に **Rendezvous Request** を S_a に対し送信する.

2. **Rendezvous Request** を受けた S_a は, R_e のノード識別子の割り当てを行う. 割り当てを行った R_e のノード識別子と S_a のノード情報をメッセージに格納し, **Rendezvous Accept** を R_e に対し送信する.

3. **Rendezvous Accept** を受けた R_e は, S_e のノード情報を格納し, さらにノード識別子を自身のノード情報に格納する.

4. 次に R_e は自身の親ノードとなるノードを探すために S_a に対して **PPL Request** を送信する. メッセージには R_e の要求レイヤ数である L_e が含まれる.

5. S_a は 5.3節で述べた手法を用い, PPL の生成を行う. この際, メッセージに含まれる R_e の要求レイヤ数がパラメータとして渡される. この例では R_c, R_d が優先度順に格納される.

6. 生成した PPL をメッセージに格納し, **PPL Data** を R_e に対し送信する.

7. メッセージを受信した R_e は PPL を格納する.

8. R_e は, PPL から最も優先度の高い R_c のノード情報を元に, **Join Request** を R_c に対し送信する. メッセージには R_e のノード情報が含まれる.

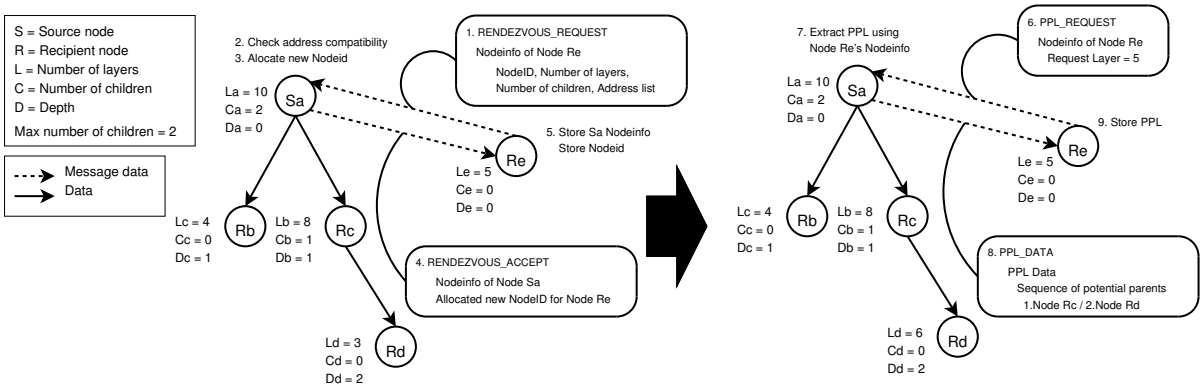

図 5.5: Join Procedure 1/3

9. **Join Request** を送信し終えた R_e は R_c のノード情報を PPL から削除する.
10. R_c は **Join Request** に含まれる R_e のノード情報を用い, Child Node として参加が可能であるかを判断する. まず R_e の要求レイヤ数が R_c の保持レイヤ数を満たしているか, 比較を行う (L_c ¿ L_c). この場合の R_e の要求レイヤ数は 5, R_c の保持レイヤ数は 8 であるため参加可能である.
11. 次に, R_c の管理子ノード数に空きがあるかの比較を行う (C_c ¡ MAXCHILD). この際の最大管理子ノード数は 2 であるため, 参加可能である.
12. R_e を子ノードとして受け入れ可能と判断した R_c は R_e のノード情報を子ノードのリストに格納する.
13. 子ノードとして受け入れ可能である事を R_e に伝えるため, R_c のノード情報を含んだ **Join Accept** を R_c に対し送信する.
14. メッセージを受信した R_e は, R_c のノード情報を親ノードのノード情報に格納する.
15. 子ノードとしての受け入れが完了した R_e はデータの要求を行うために, R_c に対して **Data StartRequest** を送信する.
16. **Data StartRequest** を受けた R_c は要求レイヤ分のデータの送信を R_e に対し開始する.
17. R_e はデータの受信を開始する.
18. R_e はマルチキャストツリーが更新された事を S_a に伝えるために **Notify JoinComplete** を S_a に対し送信する. このメッセージには, 親ノードと子ノードの対, R_c と R_e のノード情報が含まれる.
19. **Notify JoinComplete** を受けた S_a は, まず R_c のツリーノード情報を R_c のノード識別子を用い検索する. R_c のツリーノード情報に子ノードとして R_e を追加する. 以上で R_e のマルチキャストツリーへの参加が完了する.

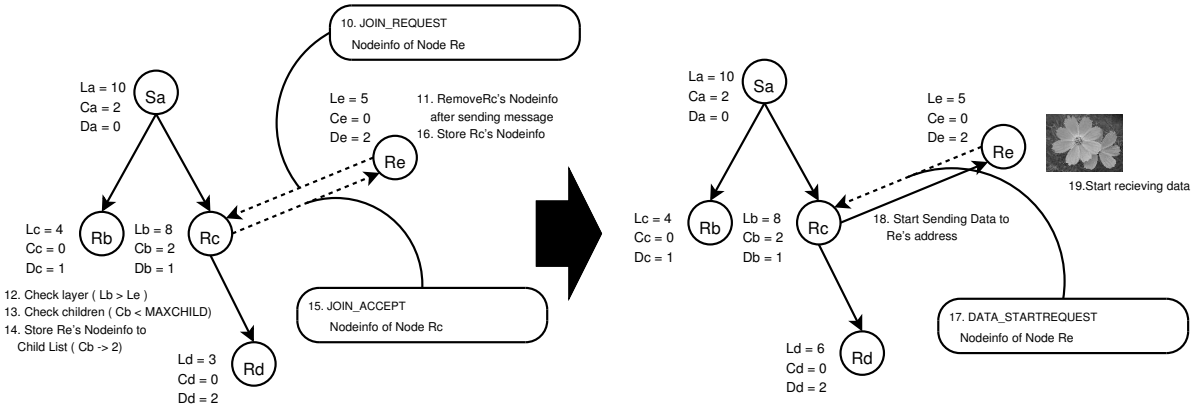

図 5.6: Join Procedure 2/3

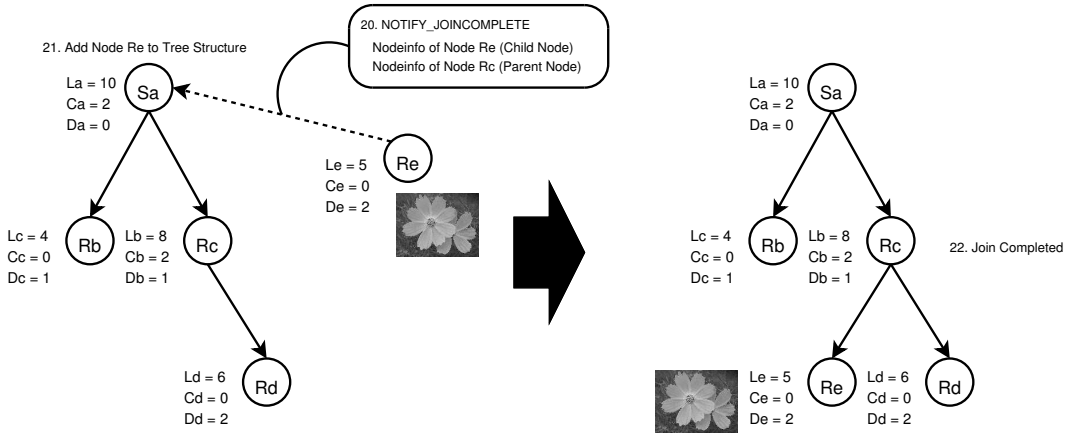

図 5.7: Join Procedure 3/3

Leave Procedure

本節では, Leave Nodeがマルチキャストツリーからの離脱を行う際の手順を図 5.9, 5.10を用い解説する. 例で用いるマルチキャストツリーには S_a, R_b, R_c, R_d, R_e, R_f の 6 ノードが参加しており, S_a が Source Node となっている. このようなマルチキャストツリーがあった際に, Child NodeR_f を管理している R_e がマルチキャストツリーからの離脱を行った場合を想定する. また, 各ノードの管理する Child Node の最大数は 2 とする. Leave Procedure において最も重要な点は, Leave Node の Child Node 以下のノードに影響を与えない様にデータを冗長化する.

1. マルチキャストツリーからの離脱を行う R_e は最初の手順として, S_a に対し, **Notify LeaveProgress** を送る. **Notify LeaveProgress** は自身が Leave Procedure を実行中であることを Source Node に伝え, PPL Extraction の際に PPL に自身を含めない様, 伝える役割がある.

2. **Notify LeaveProgress** を受けた S_a はツリー構造内の Leave Procedure 中ノードのリストに R_e を追加する.

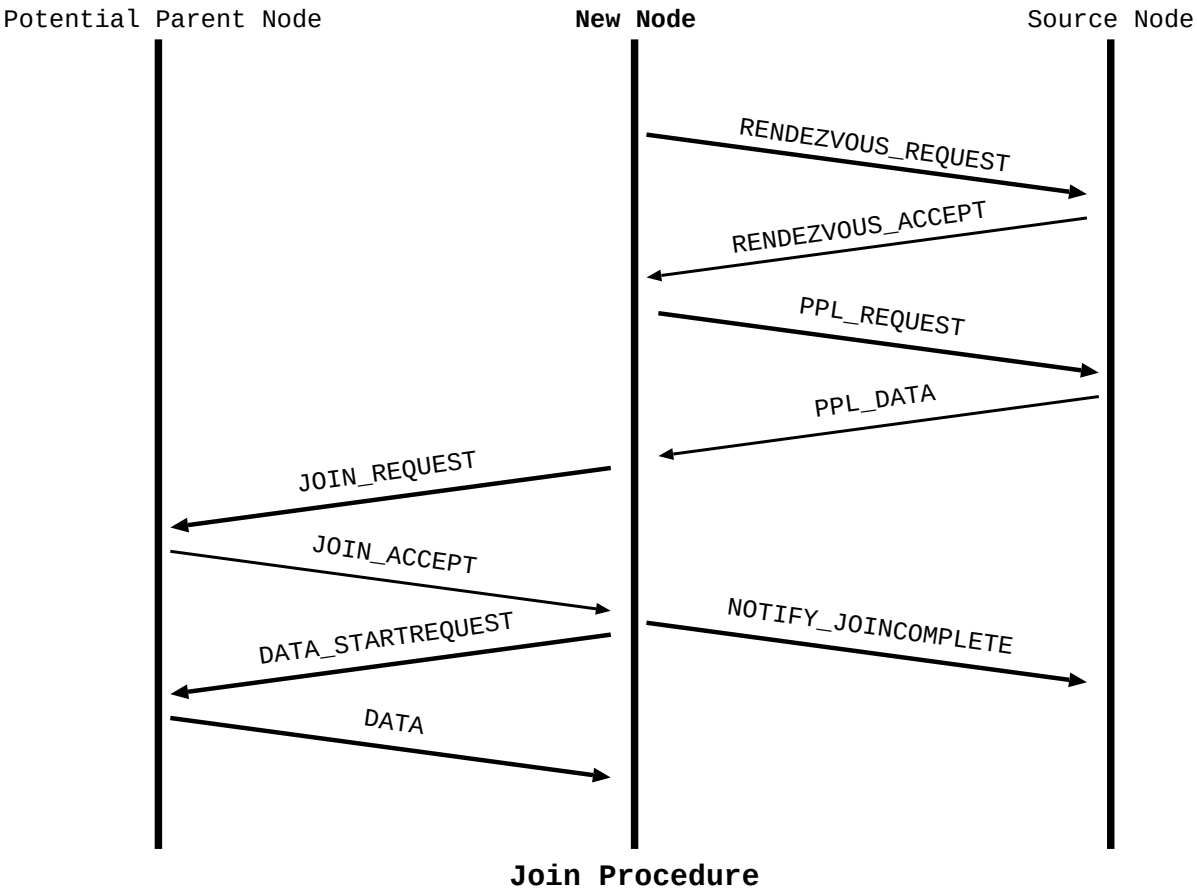

図 5.8: Join Procedure におけるメッセージパッシング

3. R_e は Child Node 以下のノードに影響を与えずに離脱を行うために, 自身からの離脱を促す **Leave Request** を自身の Child Node のリストに含まれる全 Child Node に対し送信する.

4. **Leave Request** を受けた, R_f は R_e のノード情報を Old Parent のノード情報に格納する.

5. R_f は再度 Join Procedure を行い, 新たな Parent Node に対し参加を行う. 新たに S_a から **PPL Request** により PPL を受け取った R_f は, R_b を新たな Parent Node として参加する. R_b から **Data StartReqeuest** によりデータを受け取り始める段階で, R_e, R_b の 2 つの Parent Node からデータを受けている.

6. R_f は新たに受け取り始めた R_b からのデータへの切り替えが完了すると同時に, データの停止要求を行う **Data StopRequest** を Old Parent である R_e に対し送信する.

7. **Data StopRequest** を受けた R_e は R_f に対する要求レイヤ分の配信を停止する.

8. Old Parent である R_e からのデータの配信が停止し, 単一の Parent Node からの配信を受けている状態になった R_f は, **Leave Request** に対する, 返信として **Leave Complete** を R_e に対し送信する. メッセージには R_f のノード情報が含まれる.

9. **Leave Complete** を受けた R_e は R_b は管理 Child Node のマップから R_f のノード情報を削除する．これにより, R_e は Child Node を一切管理していない状態となる．

10. R_f は Parent Node が R_e から R_b に変更され, マルチキャストツリーが更新されたことを Source Node に伝える必要がある．R_f は Old Parent のノード s 情報に格納されている R_e のノード情報を削除する．さらに新たな Parent Node である R_b と自身のノード情報を含んだ **Notify ParentChanged** を S_a に送信する．

11. **Notify ParentChanged** を受けた S_a は, ツリー構造内のノード識別子とツリーノード情報のマップを利用し, R_f のツリーノード情報を検索する．R_f のツリーノード情報を用い, R_f の Parent Node を R_e から R_b に繋ぎ変え, ツリー構造を最新の状態に更新する．

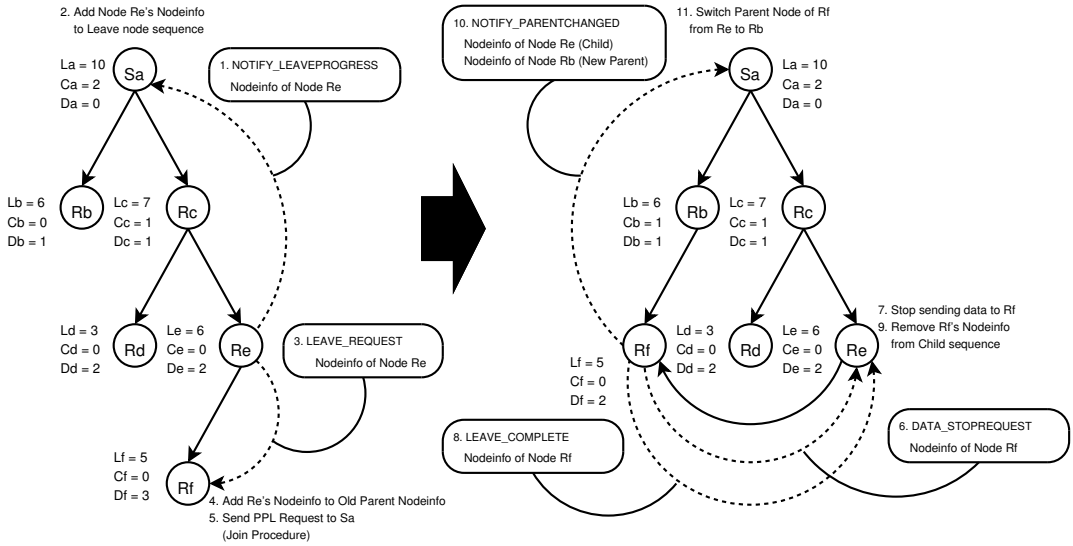

図 5.9: Leave Procedure 1/2

12. Leave Procedure の 9. において Child Node 一切管理しない状態になった R_e は Parent Node に対し自身を切り離す要求を行う **Prune Request** を送信する．メッセージには R_e のノード情報が含まれる．

13. **Prune Request** を受けた R_c は管理子ノードのリストから R_e のノード情報を削除する．削除を終えた, R_c は **Prune Reuqest** への返信として R_e に **Prune Compelte** を送信する．

14. **Prune Complete** を受けた R_e は, Parent Node も管理する Child Node も一切無く, マルチキャストツリーから離脱した状態となる．最後に R_e は Source Node に対し **Notify LeaveCompelte** を送り, 自身をツリー構造から削除するように要求する．

15. **Notify LeaveComplete** を受けた, S_a はツリー構造から R_e のツリーノード情報を削除し, Leave Procedure が完了する.

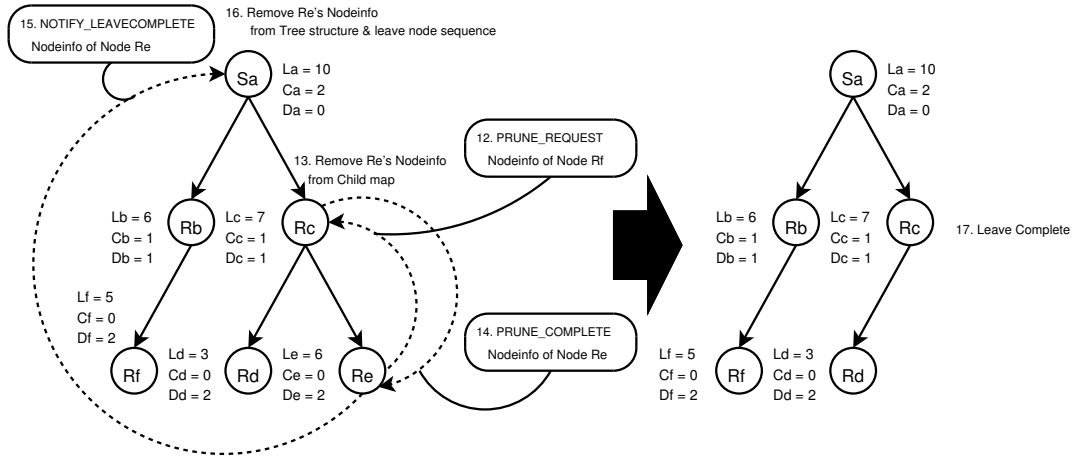

図 5.10: Leave Procedure 2/2

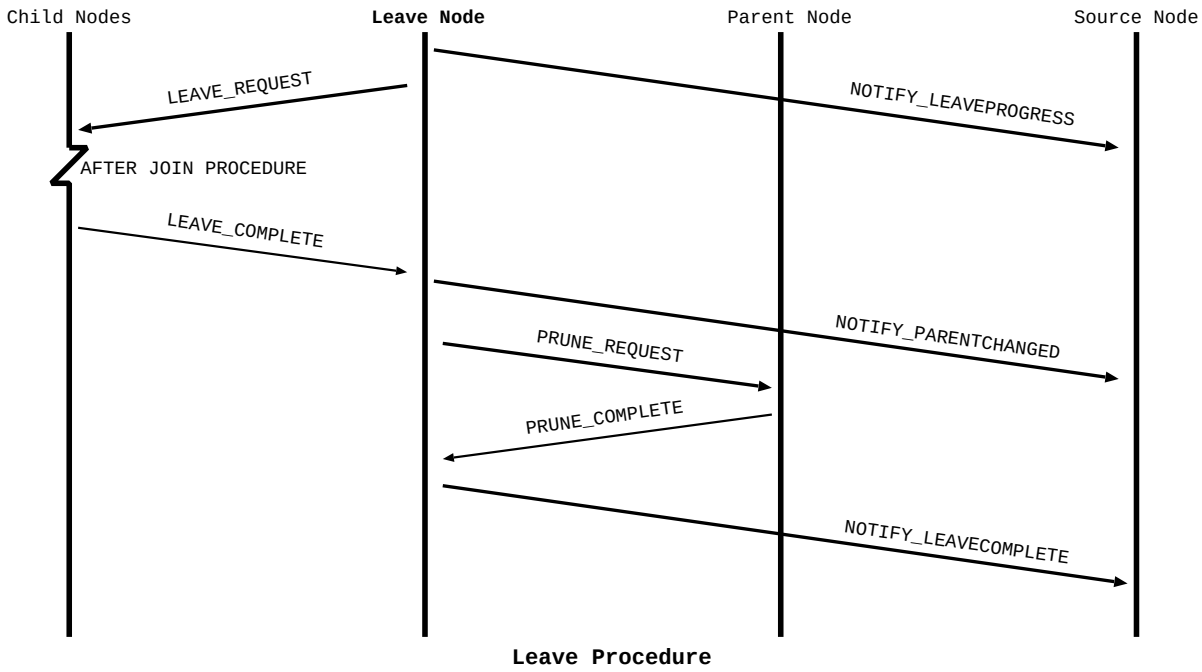

図 5.11: Leave Procedure におけるメッセージパッシング

第6章　実装

6.1　実装概要

LOLCASTプロトコルのシミュレータによる評価, アプリケーションによる評価を単一のプロトコル処理部分で行うために図 6.1のようなモジュールに分割されたシステムを実装する. LOLCASTは 4つのモジュールより構成される. LOLCASTプロトコル処理モジュール, シミュレーションモジュール, アプリケーションモジュール, ネットワークモジュールの 4つである. 各モジュールの持つ機能と関係を述べる.

- **LOLCASTプロトコル処理モジュール**
  LOLCASTプロトコル処理モジュールは本システムの核となるプロトコルに関連する処理を行うモジュールである. 本モジュールにより, 到着メッセージによるデータ構造への処理や, それに対する返信となるメッセージを下部に位置するモジュール接合部に引き渡す役割を担っている. メッセージパッシングよ用いたプロトコルの処理により, マルチキャストツリーの構築を行う.

- **モジュール接合部**
  モジュール接合部ではアプリケーションモジュールやシミュレータモジュールから引き渡されたメッセージをプロトコル処理モジュールに対しメッセージにあった処理への振分けをおこなう. また, プロトコル処理モジュールにおいて処理を終えた返信のメッセージをアプリケーションモジュールやシミュレータモジュールに引き渡す役割を担っている.

- **アプリケーションモジュール**
  アプリケーションモジュールはデータをプロトコル処理モジュールより指定されたフォーマットに従いレイヤ分割結合し, ネットワークモジュールにその送信要求を行う. 受信したデータを表示する機能もここで提供される. モジュール接合部からはレイヤ数, 送信先の情報のみが与えられるため, プロトコル処理モジュールはデータフォーマットに依存すること無く処理ができる.

- **ネットワークモジュール**
  ネットワークモジュールはデータの中身は判断せず, 指定されたアドレスに対し渡されたデータの送信, 受信したデータのアプリケーションモジュールへの引き渡しの機能を担う. 最後にシミュレータモジュールでは, 仮想的に複数のノードを用意しプロトコル処理モジュールからのメッセージに従った処理を指定された仮想的なノードに対して行う.

6.2　実装環境

本実装を行った環境を表 6.1に示す.

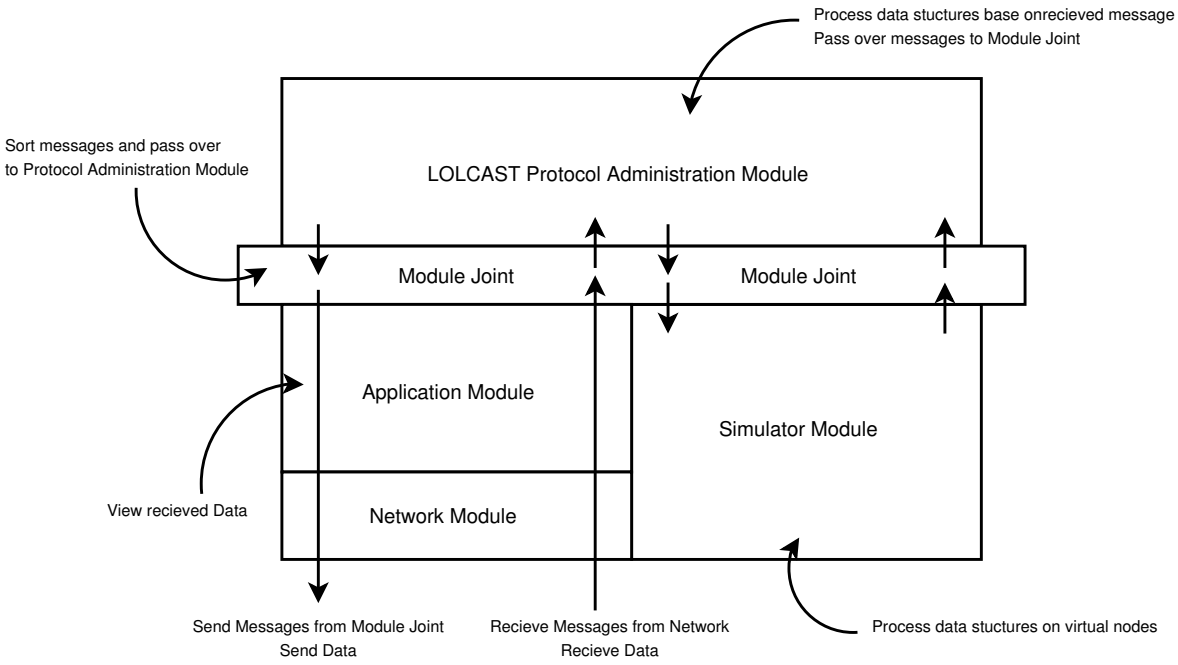

図 6.1: LOLCAST システム図

表 6.1: 実装環境

プロセッサ	PowerPC G4 1.25Ghz / Pentium III 1Ghz
メモリ	768MB / 512MB
OS	MacOSX 10.3.7 / NetBSD 1.6.2
開発言語	C 言語
利用ライブラリ	ctmpl

6.2.1 CTmpLib

C 言語においてテンプレートを扱う CTmpLib を本実装で利用した. CTmpLib は抽象化されたデータ構造の指定により複数の実装方法を選択できる. シーケンス, セット, マップ, マルチセット, マルチマップに対応しており, それぞれの構造に対し複数の実装方法が選択できる. また, すべてのデータ構造に対する操作は単一なインターフェイスにより利用できる. 本実装では, リストとマップの作成に CTmpLib を利用した.

6.3 データ構造

本節では 5節で述べた各ノードの保持するデータ構造の実装方法を解説する.

Source Node

図 6.2に Source Node の保持するデータ構造とそれぞれの関係を述べる．Source Node は全ノードを含むツリー構造の保持を行う **disttree** 型の構造体と, 自身のノード情報を保持する **nodeinfo** 型の構造体をもつ． **disttree** 内には, 自身のツリーノード情報の保持を行う **disttree_node** 型の **dt_source**, Leave Node を保持する **distnodeseq** 型のリスト, 全ノードのツリーノード情報の検索と保持を行うマップである **distnodemap** 型の **dtmap_nodes** が保持される．**distnodemap** の実装方法としてスキップリストを採用した．

Recipient Node

Recipient Node の保持するデータ構造とその関係を, 図 6.3に示す．Recipient Node は自身のノード情報, Parent Node のノード情報, Old Parent Node のノード情報, Source Node のノード情報を **nodeinfo** 型で保持している．さらに **nodeinfo** 型のリスト構造である, **nodeseq** 型で管理 Child Node のリストと PPL を保持している．

6.4 LOLCASTプロトコル処理モジュール

本節では LOLCAST プロトコル処理モジュールの実装において特徴的な処理を行う箇所の解説を行う．

ツリー操作関数群

ツリー構造関数群は Source Node の保持するツリー構造に対しノードを追加, 削除, 入れ替えを行う．本節ではツリー構造関数群の中から重要な役割を持つ関数を取り上げ, それぞれについて解説を行う．

- **disttree_addnode()**
  disttree_addnode() はツリー構造にノードを追加する際に利用する．引数として新規参加ノードと Parent Node の **disttree_node** を取る．Parent Node の検索を **dtmap_nodes** を用いて行い, その **dtmap_children** に新規参加ノードを追加する．

- **disttree_removenode()**
  disttree_removenode() はツリー構造からのノードの削除に利用する．引数として削除を行うノードとその Parent Node の **disttree_node** を取る．**disttree_addnode()** と同様に, **dtmap_nodes** を用いて Parent Node の検索を行う．見つかった Parent Node の **disttree_node** の **dtmap_children** から離脱ノードのエントリを消去する．最後に離脱ノードの領域を解放する．

- **disttree_setdepth()**
  disttree_changeparent() はノードの深さを設定する関数である．引数として Parent Node と Child Node の **disttree_node** を取り, Child Node 以下に存在するすべての子ノードの深さを更新を再起的に行う．

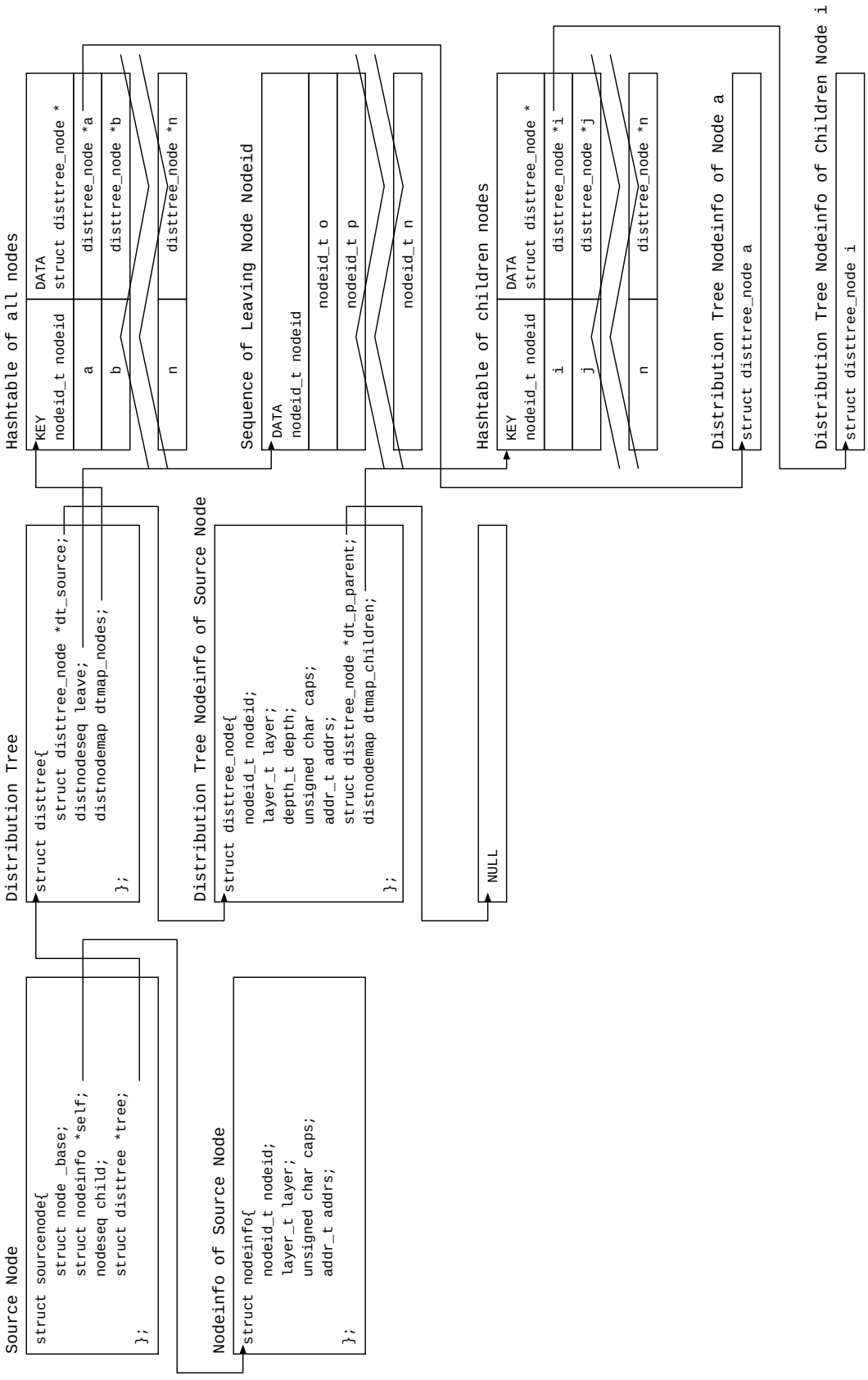

図 6.2: Source Node の保持するデータ構造

図 6.3: Recipient Node の保持するデータ構造

```
struct disttree *disttree_new(void);
void disttree_delete(struct disttree *disttree_p);
struct disttree_node *disttree_node_new(void);
void disttree_node_delete(struct disttree_node *disttree_node_p);
struct disttree_node *disttree_node_find(struct disttree *disttree_p, nodeid_t nodeid);
void disttree_switchnode(struct disttree *tp, struct disttree_node *parent,
                         struct disttree_node *child);
void disttree_addnode(struct disttree *disttree_p, struct disttree_node *parent,
                      struct disttree_node *child);
void disttree_removenode(struct disttree *disttree_p, struct disttree_node *child);
void disttree_changeparent(struct disttree *disttree_p, struct disttree_node *newparent,
                           struct disttree_node *child);
void disttree_setdepth(struct disttree_node *parent, struct disttree_node *child);
distnodeseq disttree_getparentlist(struct disttree *tp, layer_t layer);
```

図 6.4: ツリー構造関数一覧

```
void disttree_changeparent(struct disttree *tp, struct disttree_node *parent,
                           struct disttree_node *child){
    distnodemap_iterator itr;

    /* check for layer, parent > child */
    if(parent->layer < child->layer){
        abort();
    }

    /* remove child data from current parent child map */
    itr = iterator_first(child->dt_p_parent->dtmap_children);
    itr = iterator_find(child->dt_p_parent->dtmap_children, child->nodeid);
    if(!iterator_is_end(itr)){
        dprintf(("\t---node %d found\n", child->nodeid));
        itr = iterator_erase(itr);
    } else {
        dprintf(("\t---node %d not found\n", child->nodeid));
        abort();
    }

    /* add child data to new parent child map */
    iterator_insert_pair(parent->dtmap_children, child->nodeid, child);
    /* change parent entry of child */
    child->dt_p_parent = parent;
    /* set depth */
    disttree_setdepth(parent, child);
}
```

図 6.5: disttree_changeparent()

- **disttree_changeparent()**
  disttree_changeparent() は Parent Node の繋ぎ変えを行い, **Notify ParentChanged** のメッセージを受け取った際に呼び出される. 引数として, Child Node の **disttree_node**, 新たな Parent Node の **disttree_node** をとり, ノードの繋ぎ変えを行う. まず, Child Node の検索を **dtmap_nodes** で行い, Old Parent Node の **dtmap_children** から Child Node のエントリを削除する. 新たな Parent Node の **dtmap_children** に Child Node のエントリを追加する. Child Node の **p_parent** に Parent Node の Nodeinfo を入れる. 最後に **disttree_setdepth()** が呼ばれ, 深さの更新が行われる.

```
distnodeseq disttree_getparentlist(struct disttree *tp, layer_t layer){
    distnodeseq tmpppl;
    distnodeseq ppl;
    distnodemap_iterator itr;
    int pplnum;

    /* 略 */

    tmpppl = distnodeseq_new(PPL_MAX, PPL_MAX, NULL, NULL);
    ppl = distnodeseq_new(PPL_MAX, PPL_MAX, NULL, NULL);
    itr = iterator_first(tp->dtmap_nodes);

    while(1){
        /* extract temporary ppl */
        itr = ppl_layercheck(tp->dtmap_nodes, tmpppl, layer, itr);

        if(iterator_count(tmpppl) > 0){
            /* remove leave node */
            pplnum = ppl_removedeny(tp->leave, tmpppl);
            /* check number of childs  */
            pplnum = ppl_childcheck(tp->dtmap_nodes, tmpppl);
            if(iterator_count(tmpppl) > 0){
                ppl = ppl_extractbydepth(tp->dtmap_nodes, tmpppl);
                /* sort ppl with depth */
                dprintf(("ppl with out switch\n"));
                break;
            }
        }
    /* 略 */
    }
    return ppl;
}
```

図 6.6: disttree_getparentlist()

- **disttree_getparentlist()**
  disttree_getparentlist() は PPL Extraction に用いられる関数である．5.3節で述べたように, Temporary PPL Extraction と PPL Extraction の二つの処理により PPL が生成される．図 6.6に **disttree_getparentlist()** 関数を示す.

  ppl_layercheck() により, レイヤ比較が行われ, Temporary PPL が生成される．次に, **ppl_removedeny()** により Leave Node が削除され, **ppl_childcheck()** により, Child Node 数の空きがチェックされる．最後に生成された PPL を **ppl_extractbydepth()** により, 深さが浅いノードから順に並べかえられ, PPL の生成が完了する.

メッセージ処理関数群

メッセージの処理を行う関数は, メッセージ毎に受信, 送信を行う recv 関数, send 関数が用意される．到着したメッセージはモジュール結合部により, **Message Type**, **Message Subtype** フィールドを利用して各 recv 関数に渡される．recv 関数はメッセージの種類に従った処理をデータ構造に対し行い, 必要であれば send 関数を呼び出す．呼び出された send 関数はメッセージの作成を行い, **Message Data** にデータを格納する.

メッセージの送受信を行うノードが Source Node か Recipient Node で処理の変わるメッセージがある．C 言語では多様性を仕様で表現できないため, 上述した **node** 型の構造体でノードの

```
void send_rendezvous_request(struct node *self);
void recv_rendezvous_request(struct node *self, struct msg *msgdata);
void send_rendezvous_accept(struct node *self, struct nodeinfo *peer);
void recv_rendezvous_accept(struct node *self, struct msg *msgdata);
void send_rendezvous_reject(struct node *self, struct nodeinfo *peer);
void recv_rendezvous_reject(struct node *self, struct msg *msgdata);
void send_ppl_request(struct node *self);
void recv_ppl_request(struct node *self, struct msg *msgdata);
void send_ppl_data(struct node *self, struct nodeinfo *peer);
void recv_ppl_data(struct node *self, struct msg *msgdata);
void send_join_request(struct node *self);
void recv_join_request(struct node *self, struct msg *msgdata);
void send_join_accept(struct node *self, struct nodeinfo *peer);
void recv_join_accept(struct node *self, struct msg *msgdata);
void send_join_reject(struct node *self, struct nodeinfo *peer);
void recv_join_reject(struct node *self, struct msg *msgdata);
void send_data_startrequest(struct node *self);
void recv_data_startrequest(struct node *self, struct msg *msgdata);
void send_data_stoprequest(struct node *self);
void recv_data_stoprequest(struct node *self, struct msg *msgdata);
void send_leave_request(struct node *self);
void recv_leave_request(struct node *self, struct msg *msgdata);
void send_leave_complete(struct node *self);
void recv_leave_complete(struct node *self, struct msg *msgdata);
void send_prune_request(struct node *self);
void recv_prune_request(struct node *self, struct msg *msgdata);
void send_prune_complete(struct node *self, struct nodeinfo *peer);
void recv_prune_complete(struct node *self, struct msg *msgdata);
void send_notify_joincomplete(struct node *self);
void recv_notify_joincomplete(struct node *self, struct msg *msgdata);
void send_notify_parentchanged(struct node *self);
void recv_notify_parentchanged(struct node *self, struct msg *msgdata);
void send_notify_leaveprogress(struct node *self);
void recv_notify_leaveprogress(struct node *self, struct msg *msgdata);
void send_notify_leavecomplete(struct node *self);
void recv_notify_leavecomplete(struct node *self, struct msg *msgdata);
```

図 6.7: メッセージ処理関数一覧

保持するデータ構造を引き渡し, **nodetype** を利用し処理を分けた. 図 6.7にメッセージ処理で用いられる関数の一覧を示す.

6.4.1 モジュール結合部

モジュール結合部は **lc_recv()**, **lc_send()** の 2 つの関数から構成される. **lc_recv()** により, 受信メッセージの振分けが行われプロトコル処理モジュールにメッセージが引き渡される. **lc_send()** はプロトコル処理モジュールより受け取ったメッセージをシミュレータモジュールやアプリケーションモジュールに引き渡す.

6.4.2 シミュレータモジュール

シミュレータモジュールでは, ノードが新たにマルチキャストツリーに参加する際に Join Procedure にかかる時間と, その際に発生する PPL Extraction にかかる時間の計測を行う機能を実装した. シミュレータモジュールは図 6.8に示す関数群から構成される.

simu_init_sourcenode(), **simu_init_recipientnode()** により, Source Node, Recipient

```
nodeid_t simu_nodeid_alloc(void);
struct sourcenode *simu_init_sourcenode(void);
struct recipientnode *simu_init_recipientnode(struct sourcenode *snp, layer_t layer);
void simu_recv(struct msg *msgdata, struct nodeinfo *peer);
struct timeval evaluate_joinprocedure(struct sourcenode *snp, struct msg *msgdata, layer_t layer);
```

図 6.8: シミュレータ関数群

```
nodeid: 1 (0 depth, 10 layer, 3 children)
         nodeid: 2 (1 depth, 8 layer, 3 children)
                 nodeid: 5 (2 depth, 8 layer, 3 children)
                         nodeid: 13 (3 depth, 3 layer, 0 children)
                         nodeid: 14 (3 depth, 6 layer, 0 children)
                         nodeid: 16 (3 depth, 3 layer, 0 children)
                 nodeid: 6 (2 depth, 7 layer, 0 children)
                 nodeid: 7 (2 depth, 7 layer, 0 children)
         nodeid: 10 (1 depth, 9 layer, 1 children)
                 nodeid: 4 (2 depth, 1 layer, 0 children)
         nodeid: 11 (1 depth, 10 layer, 2 children)
                 nodeid: 3 (2 depth, 6 layer, 3 children)
                         nodeid: 8 (2 depth, 2 layer, 0 children)
                         nodeid: 9 (2 depth, 5 layer, 0 children)
                         nodeid: 12 (3 depth, 2 layer, 0 children)
                 nodeid: 15 (2 depth, 10 layer, 0 children)
```

図 6.9: マルチキャストツリー出力の例

Node のデータ構造が初期化される. **simu_recv()** は **lc_send()** に引き渡されたメッセージの受信に利用される. **evaluate_joinprocedure()** により, ノードが Join Procedure により参加するまでの処理にかかる時間を計測する. **evaluate_joinprocedure()** を複数回実行することにより, ツリーに参加するノードの増加による Join Procedure にかかる時間の増加を測定できる. **evaluate_joinprocedure()** は引数として新規ノードの要求レイヤ数を取り, 0 を指定することによりランダムなレイヤ数を指定できる. 乱数の生成には random 関数を用いた. また, 時間の計測には gettimeofday 関数を用い, 返り値として経過時間を返す. **evaluate_joinprocedure()** をランダムなレイヤ数を指定し実行した際に生成されるマルチキャストツリーの一例を図 6.9に示す.

第7章　評価

7.1　評価の目的

本節では, LOLCASTが 1.2節で述べた目標環境実現のための必要要件を満たしているか考察を行う. 必要要件を満たすことにより,「発信者が一般利用者にとって現実的な資源で, 多くの受信者に対し, 個々の受信者が要求する品質でサービスを提供できる環境」が実現される. また, LOLCASTにおいて提案した階層的な構造を持つデータの配信に伴うオーバーヘッドの計測を行い, プロトコルの有効性を述べる.

7.2　定性評価

定性評価では, 1.2節で述べた目標環境実現のための必要要件を LOLCASTが満たしているか考察を行う.

図 7.1に示すように, LOLCASTは既存オーバーレイマルチキャスト技術と比べ, 単一の配信網の管理により, 複数の受信者の資源環境に適した配信が行える. 配信者の帯域的資源の負担も, 品質毎のデータを送ること無いため, 少なく, 多様な受信者に対応できる.

7.3　定量評価

定量評価ではシミュレータを用い, LOLCASTで新たに提案を行った階層的な構造を持つデータの配信に伴うオーバーヘッドの増加を検証した. オーバーヘッドとは Source Nodeにおける処理時間の増加を指す. Source Nodeに対する Join Procedureとその処理に含まれる PPL Extractionの処理時間を利用しオーバーヘッドの検証を行う.

前述した Join Procedureと PPL Extractionの処理時間を計測するシミュレータは 6章で実

表 7.1: 定性評価

評価項目/配信技術	LOLCAST	既存オーバーレイマルチキャスト技術
配信元の帯域的負荷	単一なデータ	単一なデータ
提供可能な品質	複数の品質	単一な品質
配信網の維持コスト	単一なマルチキャストツリー	品質毎のマルチキャストツリー
品質保証	最低品質のデータ保証, 輻輳回避手法	冗長化された配信パスを保証 (HostCast)

```
% ./lolcast -r 10
#nodenum joinprocedure pplextraction
1 93 20
2 63 12
3 29 12
4 23 9
5 48 9
6 25 10
7 26 9
8 24 9
9 48 10
10 35 18
```

図 7.1: シミュレータ実行例

表 7.2: 評価環境

プロセッサ	Pentium III 1Ghz
メモリ	512MB
OS	NetBSD 1.6.2
コンパイラ	gcc 2.95.3 (20010315)
コンパイラオプション	-O3

装を行った LOLCAST プロトコル処理モジュールとシミュレータモジュールを用いた.

次にシミュレータの実行方法と出力されるデータについて述べる. 実装を行ったシミュレータは引数としてマルチキャストツリーに参加する総ノード数を指定する. 実行すると, 現在の参加ノード数, 参加ノードが増加した際の Join Procedure にかかる時間, 参加ノードが増加した際の PPL Extraction にかかる時間を出力する. シミュレータの実行例を図 7.1に示す. この際, **-r** オプションをつけることにより, ランダムなレイヤ数を要求するノードの参加時間を出力できる. **-r** オプションを指定しない場合は同一のレイヤ数を要求するノードの参加時間を出力する. それぞれの時間の単位はマイクロ秒で表される.

本評価では, 新規参加ノードが均質な値を要求レイヤ数として指定した場合と要求レイヤ数をランダムとした場合の処理時間を比較することにより, 階層的な構造を持つデータの配信に伴うオーバーヘッドを測定した.

7.3.1 実験環境

以下の環境においてシミュレータを実行した. この際, プロセススイッチによるデータのブレを防ぐため, 他プロセスの影響を受ける可能性の低い Single User Mode で計測を行った.

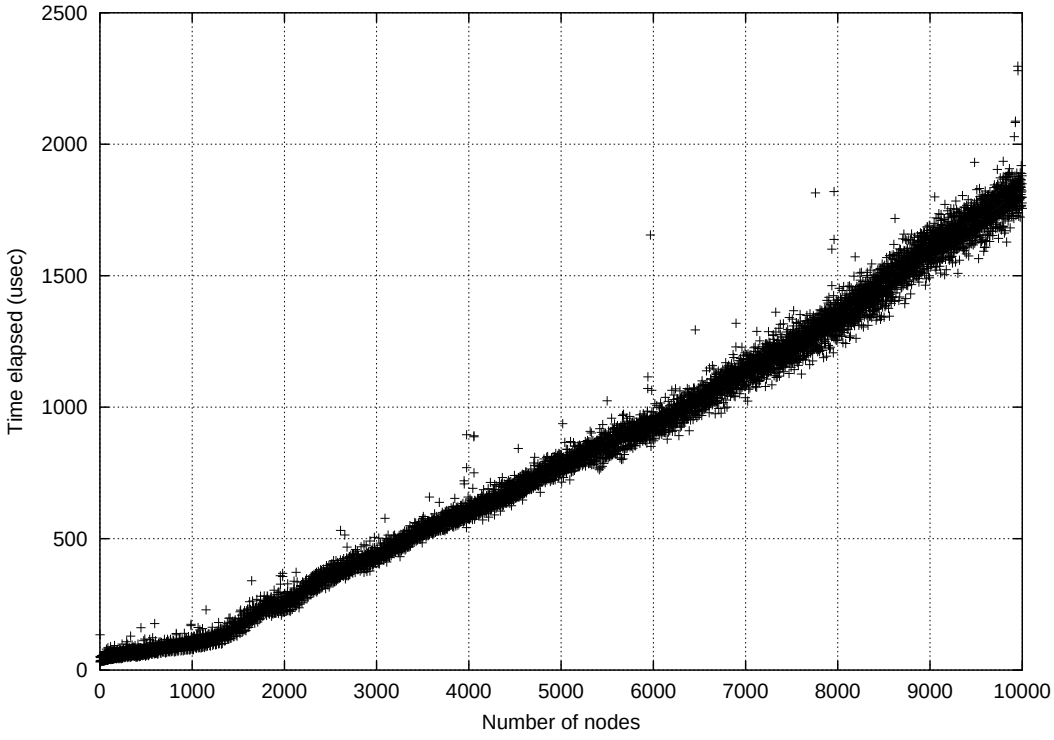

図 7.2: Join Procedure 所要時間 (10,000 nodes / Fixed layer / Skip List)

7.3.2 実験結果

固定レイヤ数による測定

図 7.2に計 10,000 ノードをマルチキャストツリーに 1 ノードづつ追加を行った際の Join Procedure に伴う処理時間を示す. 図 7.3に計 10,000 ノードをマルチキャストツリーに 1 ノードづつ追加を行った際の PPL Extraction に伴う処理時間を示す. 共に, 新規参加ノードの要求レイヤ数は固定値を取る. 図 7.3に 1000 ノード毎の Join Procedure, PPL Extraction の平均処理時間を示す.

ランダムレイヤ数

図 7.5に計 10,000 ノードをマルチキャストツリーに 1 ノードづつ追加を行った際の Join Procedure に伴う処理時間を示す. 図 7.6に計 10,000 ノードをマルチキャストツリーに 1 ノードづつ追加を行った際の PPL Extraction に伴う処理時間を示す. 共に, 新規参加ノードの要求レイヤ数はランダムに決定される. 図 7.4に 1000 ノード毎の Join Procedure, PPL Extraction の平均処理時間を示す.

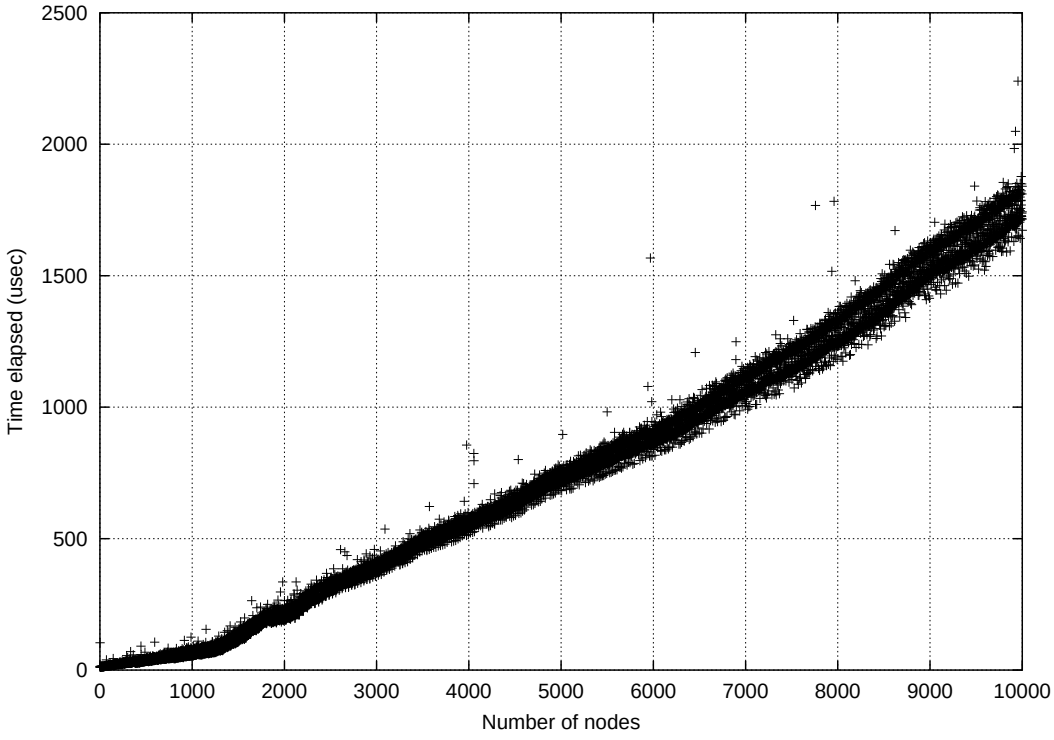

図 7.3: PPL Extraction 所要時間 (10,000 nodes / Fixed layer / Skip List)

表 7.3: 同一のレイヤ数の値を指定した際の処理時間平均

ノード数	Join Procedure (usec)	PPL Extraction (usec)
1000	75.211	39.284
2000	174.944	134.929
3000	355.41	311.035
4000	527.597	480.662
5000	692.782	643.399
6000	860.21	809.592
7000	1032.33	980.978
8000	1234.35	1180.21
9000	1468.23	1411.35
10000	1714.89	1654.12

7.4 評価結果

LOLCAST の想定規模である数千人を対象とした配信を考える. ランダムなレイヤ数を要求する 5,000 ノードが参加を行った際には, 図 7.4に示す様に 600usec 程度で Join Procedure が完了している. また, 10,000 ノードが参加を行っている場合でも 2,000usec 程度で Join Procedure が完了している点から, ランダムなレイヤ数を用いても大きなオーバーヘッドは発生しないと

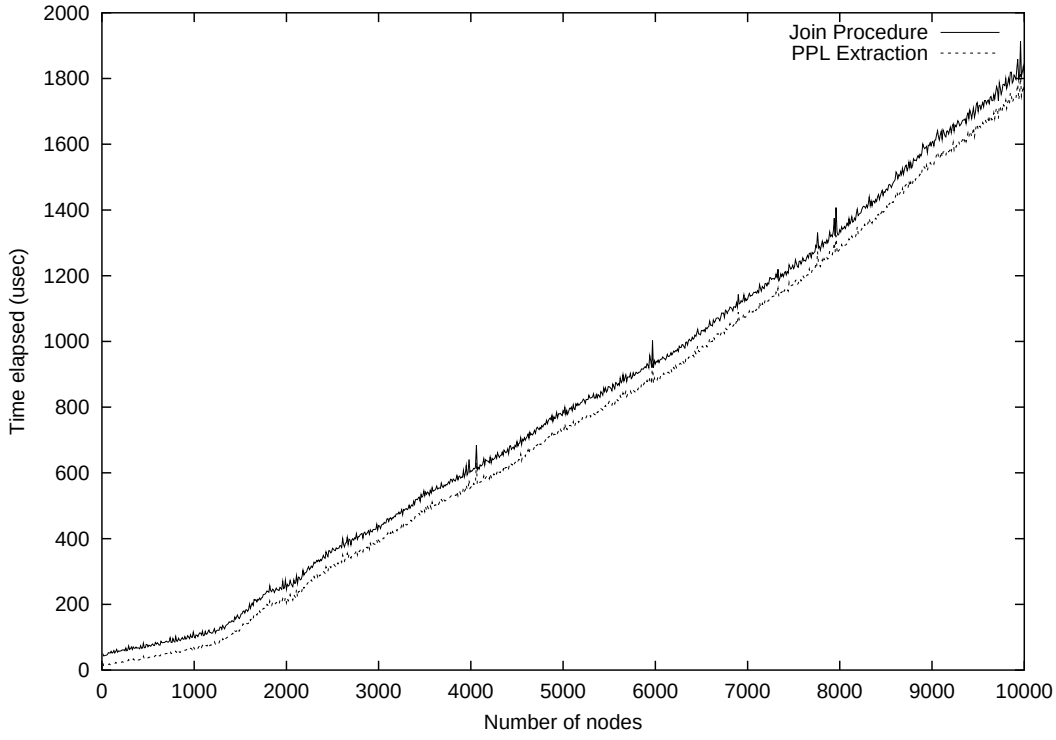

図 7.4: Join Procedure における PPL Extraction 所要時間 (10,000 nodes / Fixed layer / Skip List)

表 7.4: ランダムにレイヤ数を指定した際の処理時間平均

ノード数	Join Procedure (usec)	PPL Extraction (usec)
1000	73.134	37.76
2000	146.678	108.842
3000	304.653	262.234
4000	475.288	430.582
5000	618.879	573.492
6000	770.426	722.807
7000	920.903	870.902
8000	1078.18	1026.74
9000	1254.48	1200.84
10000	1443.25	1386.22

言える. また PPL の処理が Join Procedure において非常に大きなウェイトを占めていることがわかる. 以上の結果により, LOLCAST を用いることにより, 既存技術と比較しより効率的な配信手法を提供できた.

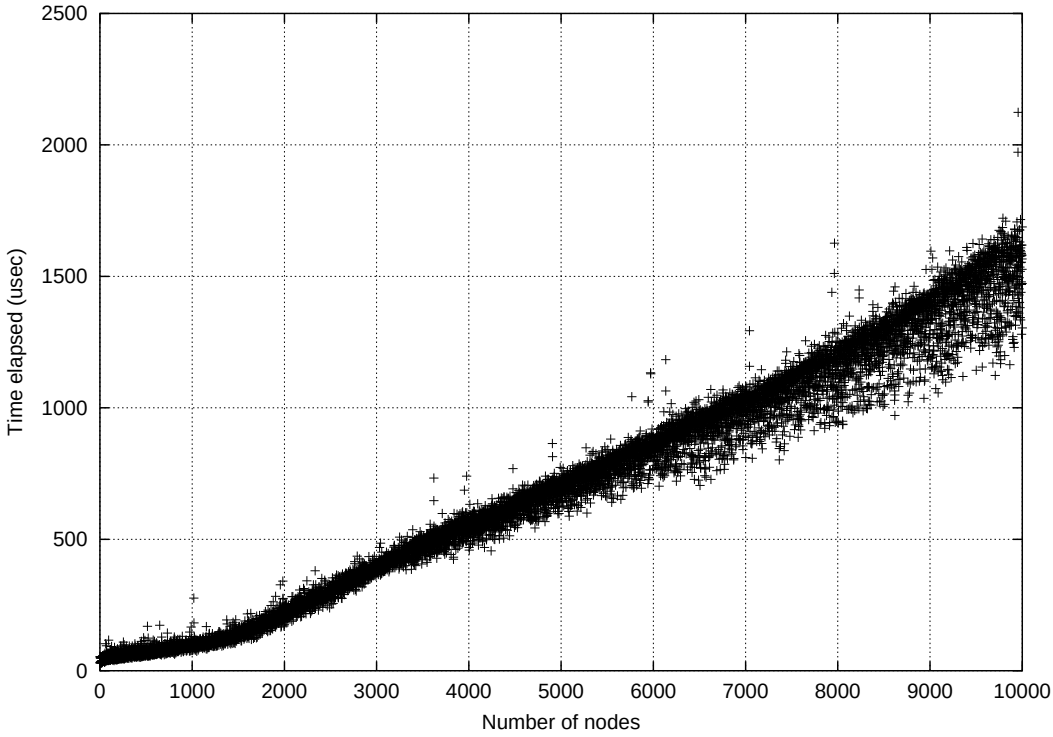

図 7.5: Join Procedure 処理時間 (10,000 nodes / Random layer / Skip List)

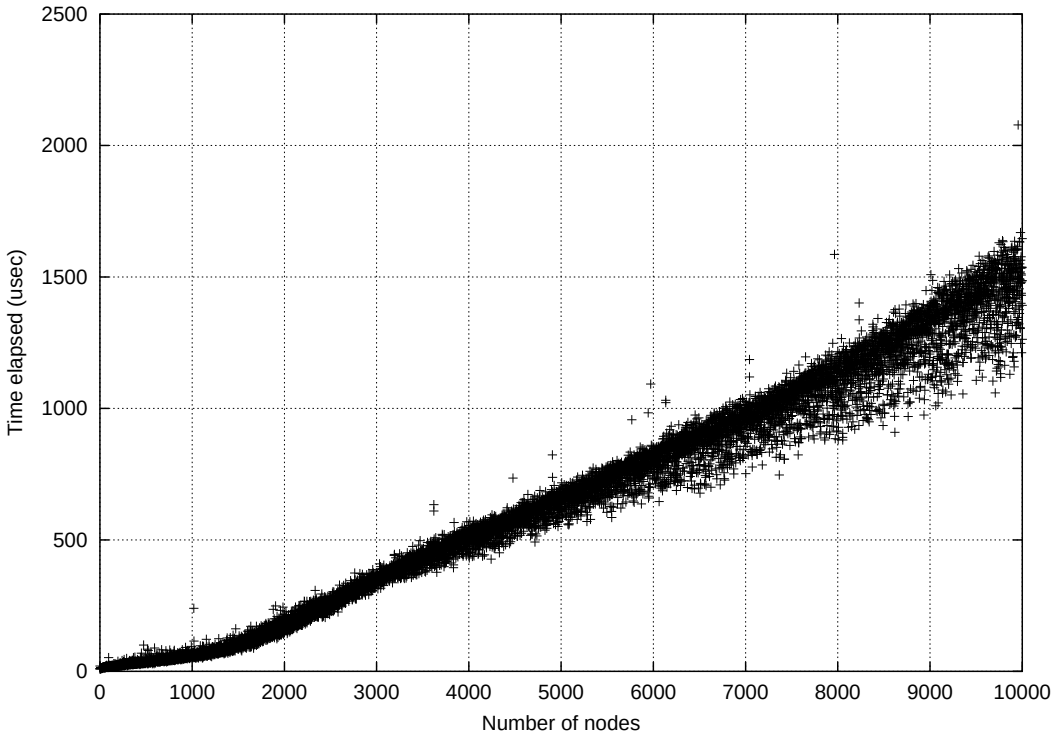

図 7.6: PPL Extraction 処理時間 (10,000 nodes / Random layer / Skip List)

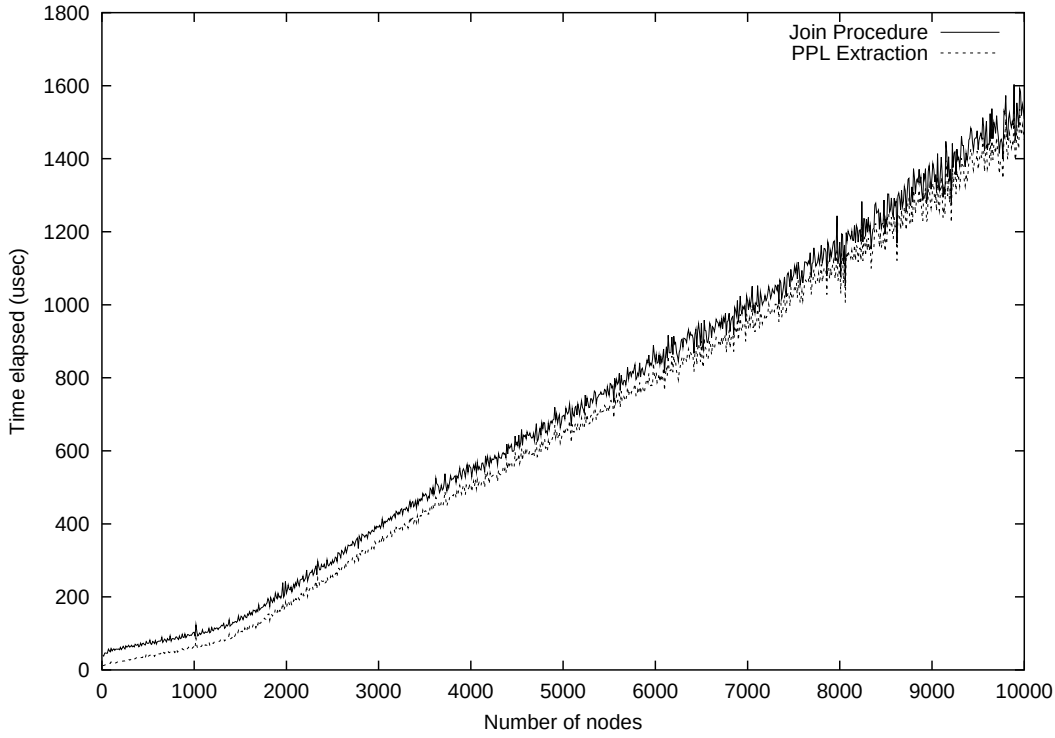

図 7.7: Join Procedure における PPL Extraction 所要時間 (10,000 nodes / Random layer / Skip List)

第8章　結論

8.1　まとめ

本研究では, 既存配信手法の問題点を解決するため LOLCAST の提案, 及び設計と実装を行い, 評価を行った. LOLCAST により, 階層構造を持つデータの配信が可能となり, レイヤという単位での配信が可能となった. 発信者は単一のデータを配信することで受信者の資源環境に合わせた配信が行われる. 発信者の資源環境が限られ, さらに受信者の資源環境が多様である特徴をもつ一般利用者を対象とする配信手法を提供できた.

8.2　今後の課題

8.2.1　PPL Extraction の性能改善

7節で述べたように, Join Procedure における処理時間の大部分は PPL Extraction の処理に裂かれており, この処理を改善することにより全体のパフォーマンスを向上できる. 現状では Temporary PPL Extraction 時にレイヤ数を満たすノードを線形にサーチを行っているため, 負荷が大きい. ノードの検索を検索木に変更することにより計算量で比較すると $O(n)$ から $Ologn$ に改善するため, 処理時間自体の向上も期待できる.

8.2.2　フレームワークの提供

現状の設計では, 個々のデータフォーマット毎にデータの加工、表示を行うアプリケーションモジュールが必要となってしまう. アプリケーションモジュール内の機能を分割し, データフォーマット依存の処理を行うモジュールをアプリケーションモジュールに組み込める構造を持たせる. データフォーマットモジュールでは, そのフォーマットにおけるレイヤの定義とデータの加工部分の実装を行う. これにより, アプリケーションモジュールはデータフォーマット依存の処理を隠蔽することが可能となり, レイヤ数という統一されたインターフェイスで複数の異なるフォーマットに対応できる.

8.2.3　アプリケーションによる評価

今回は, シミュレータの実装とそれを用いた評価を行った. オーバーレイ・マルチキャスト技術は対象アプリケーションに沿った設計を行うことが重要とされているため, アプリケーションによる評価も必要となる. 本研究の目標で挙げた, リアルタイム性の高い映像配信手法として LOLCAST が利用可能であるかを映像配信アプリケーションを利用し評価する.

謝辞

本研究を進めるにあたり，御指導を頂きました，慶應義塾大学環境情報学部教授徳田英幸博士，村井純博士，同学部助教授の楠本博之博士，中村修博士，同学部専任講師の南政樹氏，重近範行博士に感謝致します.

研究において絶えず御指導と御助言を頂きました慶応義塾大学政策メディア研究科 今泉英明氏, 石原知洋氏に深く感謝します. 最後の最後で心強いサポートをくれた後輩, 空閑洋平氏に感謝します. また同じ境遇に立ち, 良き話し相手, 相談相手となってくれた, 慶応義塾大学政策メディア研究科 入野仁志氏, 慶応義塾大学環境情報学部, 谷隆三郎氏, 千代佑氏, 松園和久氏, 同大学総合政策学部 吉田雅史氏に感謝します.

最後に論文執筆中, 良質の音楽を提供してくれた以下のアーティスト達に感謝します. Tidy Trax, Untidy Trax, ACO, Clammbon, Honda Lady, YMCK, Rainstick Orchestra, Laurent Garnier, X-Dream, Hixxy, Kurli, Four Tet, London Elektricity, Massive Attack, Parasence, Infected Mushroom, Radio Head, Bjork, Royksopp, David Lewston, Scion, Shpongle, Suitei Shyoujyo, DJ Q'Hey, DJ Shinkawa, DJ Hoekzema, DJ Uraken, Every Little Thing, Mr.Children, DJ Die, DJ Marix, Lalgudi G. Jayaraman, Ustad Amjadali Khan, Untidy DJ's, Tidy Boys, Perpetual Motion, 1200mics, Sorma, Kindzadza, Psyside, DJ Now!, Aoa, Kojima Mayumi, Eminem, Scott Brown, Juno Reactor, UA, etc.

参考文献

[1] HITACHI. 360 度どこからでも見ることができる立体映像ディスプレイ技術. *http://www.hitachi.co.jp/New/cnews/040224a.html*, 2003.

[2] Christophe Diot and Brian Neil Levine and Bryan Lyles and Hassan Kassem and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. In *IEEE Network Vol.14, num 1*, pages 78–88, 2000.

[3] P. Francis. Yoid : Extending the internet multicast architecture. In *Technical report, AT&T Center for Internet Research at ICSI (ACIRI)*, April 2000.

[4] S. Banerjee and B. Bhattacharjee. A comparative study of application layer multicast protocols. 2002.

[5] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast (keynote address). In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12. ACM Press, 2000.

[6] Yang Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 55–67. ACM Press, 2001.

[7] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[8] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217. ACM Press, 2002.

[9] B. Zhang, S. Jamin, and L. Zhang. Host multicast: A framework for delivering multicast to end users. In *IEEE Infocom*, 2002.

[10] Zhi Li and Prasant Mohapatra. Hostcast: A new overlay multicasting protocol. In *IEEE International Communications Conference (ICC)*, 2003.

[11] Anne-Marie Kermarrec Miguel Castro, Michal B. Jones and Antony Rowstron. An evaluation of scalable application-level multicast built using peer-to-peer overlay. In *Infocom 2003, San Francisco, CA*, 2003.

[12] 村田正幸 CaoLe Thanh Man, 長谷川 剛. サービスオーバーレイネットワークのためのインラインネットワーク計測に関する一検討. In **電子情報通信学会技術研究報告** *(IN03-176)*, pages 53–58, 2003.

[13] Dimitris Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, San Francisco, CA, USA, March 2001.

[14] Yatin Chawathe. Scattercast: an adaptable broadcast distribution framework. *Multimedia Syst.*, 9(1):104–118, 2003.

[15] John Jannotti, David K. Gifford, M. Frans Kaashoek, and James W. O'Toole Jr. Overcast: Reliable multicasting with an overlay network. In *5th Symposium on Operating System Design and Implementation (OSDI)*, December 2000.

[16] Yan Zhu, Min-You Wu, and Wei Shu. Comparison study and evaluation of overlay multicast networks. In *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference*, July 2003.

[17] MPEG-2 Generic coding of moving pictures and associated audio information. http://www.chiariglione.org/mpeg/standards/mpeg-2/mpeg-2.htm.

[18] David Taubman. High performance scalable image compression with ebcot. In *IEEE Transactions on Image Processing Vol.9 No.7*, pages 1158–1170, 2000.

[19] David Taubman, Erik Ordentlich, Marcelo Weinberger, Gadiel Seroussi, Ikuro Ueno, and Fumitaka Ono. Embedded block coding in jpeg2000. In *IEEE International Conference on Image Processing (ICIP)*, pages 33–36, 2000.

[20] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 43–56. ACM Press, 2000.