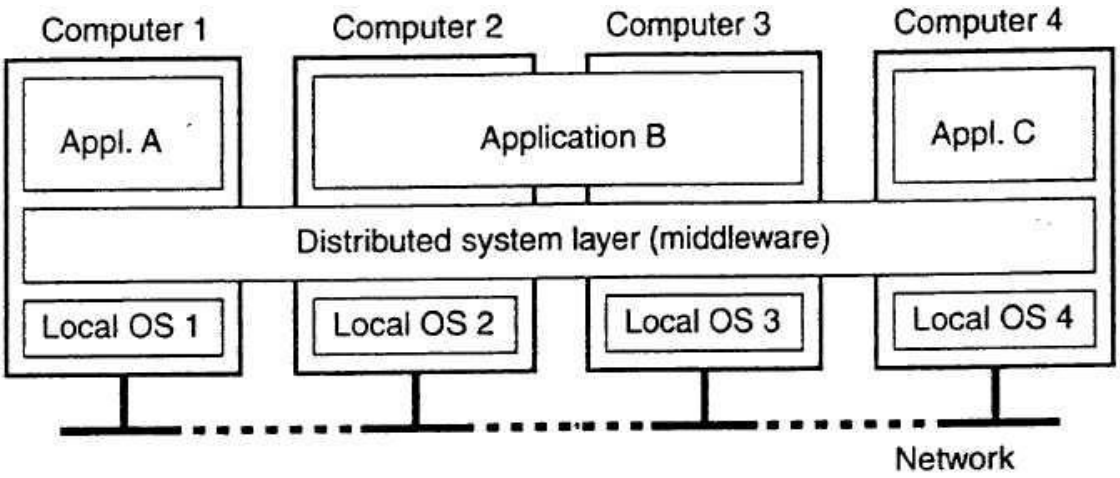
سیستم‌های توزیعی
دکتر پدرام

سيستم هاي توزيعي
اصلاحات درس

فصل 1: تعاریف

تعریف سیستم توزیعی: مجموعه‌ای است از کامپیوترهای مستقل و خودمختار که از دید کاربرانش یک سیستم منسجم (Coherent) و منفرد به نظر می رسد. یعنی سیستم هایی که در یک شبکه توزیع شده است ولی این توزیع و گستردگی از دید کاربرانش مخفی است و کاربر فکر می کند بایک سیستم محلی سروکار دارد. هر سیستم سیستم عامل و Platform خودش را دارد.

شکل سیستم توزیعی



Middleware: لایه است نرم افزاری بین سیستم عامل و برنامه کاربردی که مفهوم توزیع شدگی را پیاده سازی می کند. استقلال Application از Platform. ارائه شفافیت.

اهداف سیستم توزیعي (دسترسی به منابع, Openness, Distribution Transparency و Scalability)

- **Making Resources Accessible:** منابع در اختیار همه قرار گیرد.
- **Distribution Transparency، شفافیت توزیع و به سمت سیستم واحد رفتن**
  - **Access Transparency:** مشخص نشود در سیستم هاي مختلف هستيم مثلاً File system، يا Notation (نمايش داده) روي سيستم يعني Beg Endean يا Little Endean

- **Location Transparency:** يعني مكان سرويس مشخص نباشد اگر باشد كه اين توزيع شدگی را داريم می‌بينيم و اين مغاير تعريف سيستم توزيعی است مثلاً www.yahoo.com كه نوعی Location Transparency دارد و محل سرويس مشخص نيست.
- **Migration Transparency:** يعني اگر سرويس جابجا شد و به محل ديگري رفت باز اين موضوع از كاربر مخفي باشد. اين قويتر از Location Transparency است.
- **Relocation Transparency:** اين قويتر از Migration Transparency است. يعني سرويس در حين استفاده جابجا مي شود.
- **Replication Transparency:** از يك سرويس جاهاي مختلف داشته باشيم.
  - مسئله مهم در اينجا Consistency است.
- **Concurrency Transparency:** پراسس های همزمان با توجه به اينكه از منابع مشترک استفاده می‌كنند, مزاحم هم مي شوند و هميشه به نظر می‌رسد فقط يک پراسس در سيستم است.
- **Failure Transparency:** اگر يک پراسس از كار افتاد سيستم به كار خود ادامه دهد يعنی وظيفه آن پراسس را به ماشين ديگری می‌دهيم.
- **Persistence Transparency:** مشخص نباشد اصلاعات روی حافظه اصلی ذخيره شده است يا روی هارد ديسک.

- **درجه شفافیت**: شفافیت با Performance در مغایرت دارد (Trade off). در برخی از کاربردها اصلاً ایده شفافیت خوب نیست مانند Mobile Application. شخصی وسیله موبایل دارد و می خواهد فایلی را چاپ کند. اگر سیستم Transparent باشد یک چاپگر خلوت را انتخاب می‌کند ولی چاپگر نزدیک از نظر فیزیکی مناسب‌تر است. گاهی شفافیت اگر سیستم خوب طراحی نشده باشد باعث می‌شود رفتار غیرقابل انتظار از خودش نشان دهد. مثلاً فرایند C می‌خواهد از سرویس S استفاده کند. اگر سرویس دهنده از کار بیفتد و برای این موضوع فکری نشده باشد سیستم دچار سردرگمی خواهد شد. برعکس هم ممکن است S به کسی پاسخ می دهد که اصلاً منتظرش نیست.
- **Openness:** به راحتي به سيستم متصل شويم و به راحتي از آن جدا شويم.
  - **Interface:** از طريق يک Interface می‌توان سرويسی را از يک Server درخواست كرد. اين Interface شامل اسم Procedure و پارامتر هاي آن است.
  - **IDL (Interface Definition Language)** زبانی است كه Interface را تعريف می كند. اين Interface شامل Syntax و Semantic را تعريف می كند.
    - **Syntax** نوع پارامترها و ...
    - **Semantic:** چه كاري انجام مي دهد.
  - **تعريف Interface بايد Complete و Neutral باشد.**
  - **Complete:** يعني همه پارامترها تعريف شده باشد.
  - **Neutral يا خنثي:** هيچ قيدي در باره نوع پياده سازي يا Implementation نبايد باشد. مثلاً زبان خاص يا پياده سازي بصورت Object Oriented باشد.

- اگر تعریف Interface بصورت Complete و Neutral باشد می‌توان خاصیت‌های Interoperability و Portability را داشت.
- **Interoperability:** مي توان از كامپانت هاي كمپاني هاي مختلف استفاده كرد.
- **Portability:** اگر component را از يك سيستم توزيعي به سيستم توزيعي ديگر ببريم باز كار كند.
- **Flexibility:** چرا می‌خواهیم Processی را جایگزین کنیم؟ چون flexibility لازم را ندارد. جدا کردن مکانیزم و Policy ما را به Flexibility می‌رساند.
- **مکانیزم:** مثلاً Web caching بايد بتوانيم ذخيره سازي انجام دهيم.
- **Policy:** مثلاً زمان Refresh كردن يا اظهار نظر روي Content برخي را Cache كنيم برخي نه.

- **Scalability يا گسترش پذيري**
  - **Scalability Problems يا موانع گسترش پذيري** (Size, Geography, Administration)
    - **Size:** مي‌خواهيم تعداد نودهاي يك سيستم را افزايش دهيم.
      - **Centralize Component** باید اجتناب شود نه تنها این بلکه از الگوریتم مرکزی هم باید اجتناب شود.
      - **Single Point of Failure**
      - **Centralized Algorithm:** مثلاً مسيريابي در شبكه. مسيريابي بر اساس اطلاعات محلي انجام مي‌شود. پس مرکزی نیست اگر مرکزی بود خیلی بد می‌شد زیرا Scalable نبود ولی احتمالاً بهترین مسیر را پیدا می‌کرد.
      - اگر این موارد را رعایت کنیم به Scalability در اندازه می رسیم:
        1) هيچ ماشيني اطلاعاتي از كل سيستم ندارد. (الگوريتم مركزي)
        2) ماشين ها بر اساس اطلاعات محلي تصميم مي گيرند.
        3) خرابي يك ماشين باعث خرابي الگوريم نمي شود. از اول هم اطلاعات این ماشین کامل نبود.
        4) نمي توان فرض كرد يك ساعت جهاني وجود دارد. یعنی الگوریتم‌ها نباید بر اساس زمان کار کنند.
    - **Geography:** مي‌خواهيم فاصله‌هاي فيزيكي نودها زياد باشد. يعني در فواصل طولاني افزايش پيدا كند.
      - مشكل اصلي تاخير يا Delay است.
      - ارتباط Synchronous
      - ارتباط Asynchronous، Interrupt و Interrupt Handler لازم است.
    - **Administration:** يعني بخشي تحت يك مديريت باشد و بخشي تحت مديريت ديگر. از همه مشكل‌تر است. ممكن است سياست ها با هم در تضاد باشد.
      - Peer to Peer يعني همه Process ها داراي يك ارزش هستند.
  - **تكنيك هاي Scaling**
    - **Hiding Communication Latency:**

- يـك روش همـان Asynchronous Communication مـی‌باشـد کـه مطلـوب نیست.
- انتقال بخشي از کار از سرور به کلاینت یا Code Migration
- **Platform:** شامل سخت افزار و سیستم عامل

▪ **Distribution:** تمام اطلاعات يك نقطه نباشد و فشار به يك نقطه نيايد.

- يك مثال DNS (Domain Name System) است که ساختار سلسـله مراتبـي درختی دارد. مثال دیگر اسناد در WWW است.

▪ **Replication:** كپي سازي هم باعث مي‌شود فشار به يك نقطه نيايد.

- این بجز اینکه در سایز موثر است در جغرافیا هم موثر است.
- Caching حالـت خـاص از Replication اسـت ولـي بـا هـم فـرق دارد. Replication توسط صاحب Resource انجام مي شـود ولـي cachingتوسـط Client انجام مي‌شود.
- **Consistency:** حالـت كپـي هـا بايـد يكسـان باشـد. مـثلاً دو Database اطلاعات بانکی نمی‌تواند متفاوت باشد.
  - **Strong:** مثل بازار سهام يا حسابهاي بانكي و کاربردهای مالی
  - **Week:** اکثر کاربردها را شامل مي‌شود.
  - دسترسي همزمان به منابع بصورت Same Order باشد.

- **دامها يا Pitfalls سر راه سيستم توزيعي كه بايد به آنها توجه شود:**
  1) The network is reliable
  2) The network is secure
  3) The network is homogenous
  4) The topology doesn't change
  5) Latency is zero
  6) Bandwidth is infinite
  7) Transport cost is zero
  8) There is one administrator
- **مثالهـايي از سيسـتم هـاي تـوزيعي** (Distributed Computing Systems, Distributed Information systems, Distributed Embedded Systems)
  - **Distributed Computing Systems**

▪ **Cluster Computing:** گـره مشـابه يـا Homogenous و دارای يـک Master Node است.

- با توجه به اینکه فاکتور Price / Performance بهبـود یافتـه یعنـی بـه سـمت کاهش میرود می‌توان Super Computer با کمک این ابرار ساخت. شکل زیر کلاستر Beowulf را نشان می‌دهد.

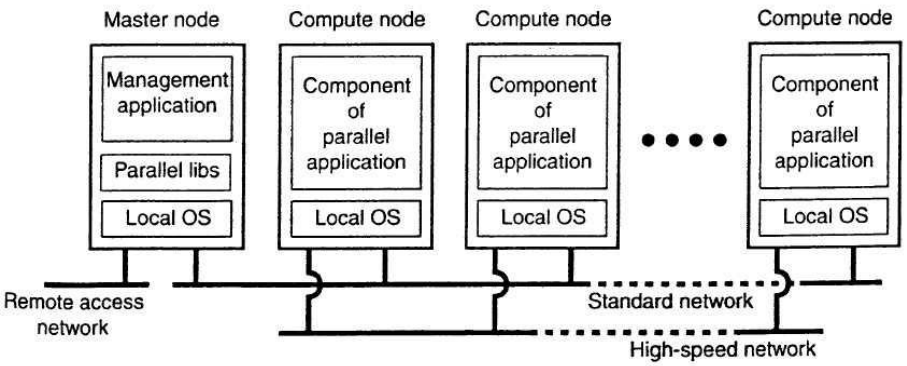


Figure 1-6. An example of a cluster computing system.

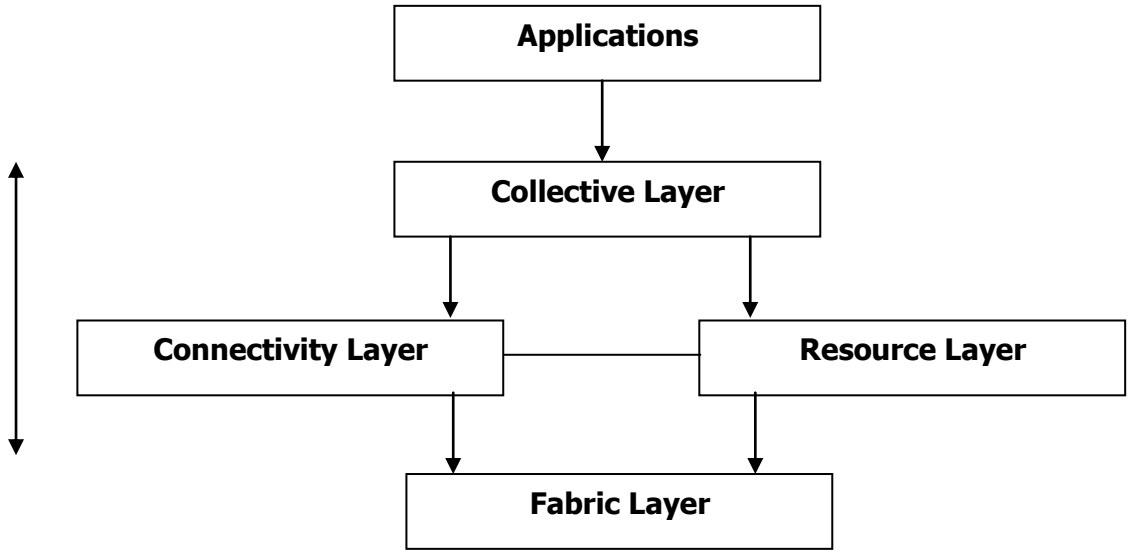
- هدف گيري Cluster Computing براي Parallel Computing يعني يك Task داريم كه به اجزاي كوچكتر تقسيم مي كنيم. نه اينكه Multiple Task داشته باشيم.
- کارهای Master Node: یکی اختصاص نود برای یک برنامه کاربردی یا تشکیل صف برای Job ها. قصد این سیستم Multi Task نیست بلکه Single Task است با سرعت بالا و همینطور ارائه Interface برای کاربران. کاربر با گره اصلی در تماس است. Middle ware شامل امکان ارتباطی پیشرفته است.
- **مدل دیگر سیستم کلاستر MOSIX است:** بر اساس Process Migration كار مي كند. Process به محل Resource می رود و با سرعت بالاتری کار می‌کند.

▪ **Grid Computing:** گره‌ها غیر مشابه یا Heterogeneous هستند. اصطلاحاً می گویند Federation of Computing Systems.

- **Resources from different Organizations** هستند كه تحت Administrator های مختلف هستند و پراکنده. مجموع این Resource ها می خواهند یک task که مربوط به یک VO است را انجام دهند کل قضیه این است.
- **Virtual Organization:** در واقع اين گروهی از کاربران یا برنامه‌ها هستند كه از Resource هاي مختلف براي انجام Task استفاده مي‌كند.
- **مدل چهار لايه براي Grid**
  - **Fabric Layer:** مسئول تنها يك Resource است. Query مي كند حالت يك Resource را. نزديكترين لايه به خود Resource.
  - **Connectivity Layer:** Communication Protocol برای فعالیت های Grid لازم است که اینجا انجام می‌شود. مثلاً براي انتقال داده بين منابع يا پروتكل‌های ديگر مانند پروتكل دسترسی راه دور يا پروتكل امنيتی تائيد هويت كاربران يا برنامه‌های كه از طرف آنها اجرا می‌شود.
  - **Resource Layer:** مكمل Fabric Layer است كه براي Management منابع بكار مي رود. Function هايی كه در

Connectivity Layer برای ارتباط مشخص شده است در اینجا پیاده سازي شده است. و اینترفیس لایه فابریك را Call مي كند.

- **Collective Layer:** اداره مجموعه Resourceها چون برای انجام یک Task به Resource های متقاوت احتیاج داریم. کشف منابع. تصمیم اینکه کدام سرویس کارایی بهتر یا هزینه کمتر دارد. تخصیص و زمانبندي Taskها. اداره Data Replication برای Performance و Fault Tolerance.
- **Applications:** این همانی است که در Virtual Organization تعریف شده است. و از محیط Grid براي اجرا استفاده مي كند. نیاز به منابعي دارد.
- **Middleware:** کار Access و Management منابع را انجام می‌دهد.



- **Distributed Information Systems:** (Enterprise Computing) براي شركت ها يا سازمانهاي بزرگ – دو مورد در این گروه بررسی می‌شود:
  - **Distributed Transaction:** Transaction فعاليتي كه داراي خاصيت همه يا هيچ است. فعاليت مي تواند يك، چند و يا بخشي از Process باشد. مثال از Transaction برداشت از يك حساب و واريز به حساب ديگر نمی‌شود بخشی از آن انجام شود. يا رزرو بليط براي سه مسير تا رسيدن به مقصد. Process يک Transaction نيست زيرا اگر بخشی از آن انجام شود و به هر دليلی متوقف شود كارهايی كه تا آن لحظه انجام داده است باقی می‌ماند. نمونه هايی از عمل‌های پايه در Transaction به اين شرح است: Begin Transaction, End Transaction, Abort, Commit, Write, Read.
    - **Transaction اصطلاحاً دارای خاصیت ACID است.**
    - **Atomic:** یا همه یا هیچ. قابل تجزیه نیست.

- **Consistent:** يعني عمل Transaction نبايد چيزهايي را كه در سيستم ثابت هستند به هم بزند. مثلاً در يك بانك از يك حساب برداشت و واريز به حساب ديگر نبايد موجودي بانك را به هم بزند.
- **Isolated:** اگر چند Transaction همزمان اجرا مي شود بايد مجزا از هم به نظر بيايد و مزاحم هم نشوند.
- **Durable:** اگر Transaction به پايان رسيد نتيجه باقي است (Permanent).
- **Nested Transactions**
  - مثال: Transaction مسافرت: رزرو هتل و رزرو بليط هواپيما
  - از ديدگاه Middleware : يك Application داريم.
  - يک مدلی هست برای Transaction بصورت شکل زير. درخواست‌ها از طرف Client همه به يک موجوديت داده می‌شود بنام Transaction processing Monitor. در شکل زير اگر همه Transactionها انجام شد کل کار تمام است اگر نشد آنهايی که انجام شده است بايد برگردد سر جايش.

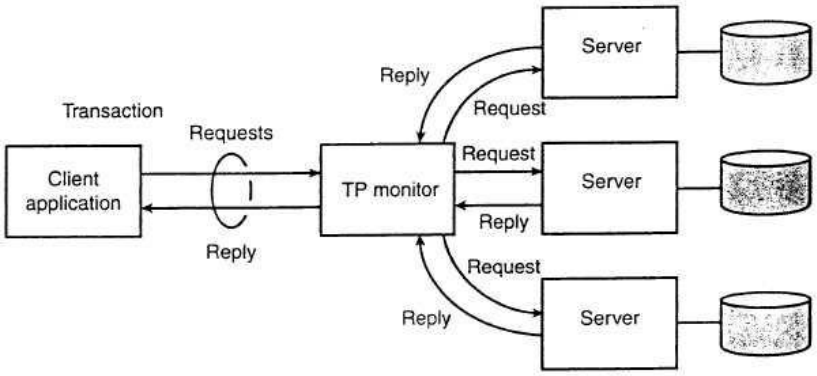
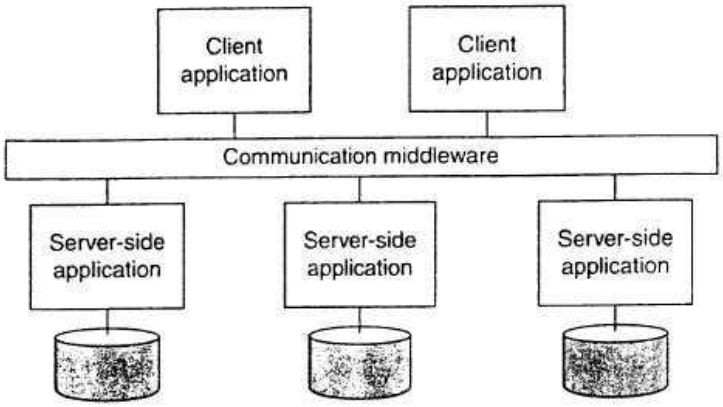


Figure 1-10. The role of a TP monitor in distributed systems.

- **Enterprise Application Integration (EAI):** Transaction Processing منحصراً از مدل Client/Server استفاده می‌کرد اما براي حالاتي كه Application ما به بخش‌های مختلف تقسيم شده است. متدهايي لازم است تا خود Application ها با هم ارتباط برقرار كنند و ارتباطات فقط بصورت Client/Server نيست.



- یکی از روش‌های ارتباط Applicationها RPC (Remote Procedure Call) است.
- روش دیگر RMI (Remote Method invocation) براي سيستم هاي Object Oriented می‌باشد.
- روش سوم Message Oriented Middleware (MOM) است که از طریق Message با هم در ارتباط هستند.
- مدل دیگر Publish Subscribe است که براي يك گروه از Process ها مشترک می‌شوند در فعالیت و آنها که مشترک هستند می‌توانند ببینند.

- **Distributed Embedded Systems:** سيستم هاي خيلي كوچك كه به هم وصل مي شود مانند تجهيزات موبايل
- **مثال بعدی سیستم های توزیعي فراگير يا Distributed Pervasive Systems:** سيستم هاي قبلي نودها Stable هستند و يك شبكه با كارايي خوب داريم كه اينها را به وصل كرده‌اند. ولي كلاسي از كاربردها هستند كه اينگونه نيستند و با اجزاي ساده تر و كوچكتري سرو كار دارد. كه مي توان در Embedded Systems يا Sensor Network ديد. كلاس ديگري از كاربردهاي Distributed هستند.
  - مشخصه اصلی این است که نودها می‌توانند متحرک باشند یا Battery Powered باشند که برای همین می‌گوییم Stable نیستند. زیرا بعد از مدتی باطری تمام می‌شود. نودها به راحتی Fail می‌کنند. دارای مشخصات زیر است:
    1) سيستم بايد تغييرات محيط را تحمل كند. گره ها زياد و كم مي شود.
    2) بصورت موردي خودش را Configure كند مثلاً بسته به Application.
    3) اشتراك در اطلاعات وجود دارد.
  - Transparency خيلي وجود ندارد.
  - مثال: Home System, Electronic Health Care Systems, Sensor Network
  - **مثال Home System**
    - يك Home Network داريم كه يک سيستم روی آن ساخته شده است شامل PC و PDA و Smart Phone و تلوزيون و ...
    - بايد Self configuring باشند چون دائم محيطشان تغيير می‌كند.
    - Self-managing بايد باشند يعنی ورود و خروج از سيستم را اداره كند.(Universe of Plug and Play)
  - **يكي ديگر از كاربردها Electronic Health Care Systems**
    - سنسورهاي مختلف در بدن قرار داده شده است كه سلامت بدن را كنترل می كند با شبكه بيسيم.
    - دو نوع است يكی هاب دارد و هاب چند وقت به چند وقت اطلاعات را به كامپيوتر می دهد هاب می‌تواند مديريت شبكه را هم انجام دهد. يكي هاب

ندارد فقط فرستنده دارد و اطلاعات از طريق فرستنده ارسال مي شود. نودها اطلاعات را به فرستنده می‌دهند.

- اين شبكه را BAN (Body Area Network) مي گويند.
- سوالات: داده ها كجا بايد ذخيره شود؟ چگونه جلوي از دست دادن اطلاعات را بگيريم؟ چه زير ساختي لازم است تا هشدارهايي صورت گيرد؟ پزشك چگونه بازخورد به اين شخص بدهد؟ سيستم چگونه مي شود مقاوم باشد و از كار نيفتد زيرا شبكه ضعيف است؟ Security را چكار كنيم؟
- In-network Data Processing زمانی است که شبکه می‌تواند پردازش مختصری انجام دهد.

- **يكي ديگر از مثالها كه كاربرد زياد دارد Sensor Networks**
  - شامل صدها و هزارها نود كوچك هستند.
  - داراي سنسور هستند كه درجه حرارت، عبور چيزي يا ... را حس مي كنند.
  - داراي قدرت باطري هستند. پس بايد بسيار كارا باشند.
  - دو چهار چوب دارند. يكي نودها با هم همكاري ندارند و پردازش ندارند و اطلاعات را به يك كامپيوتر مركزي.
  - ديگري برعكس هر نود توان پردازشي و ذخيره دارند اگرچه محدود و بعد به يك سيستم مركزي متصل است. بهتر است روش تركيبی استفاده شود.
  - Aggregate بسته اطلاعات ارسال كنند.
  - مي تواند چيزي بين اين دو باشد.
  - Dynamic Tree (In-network Data Processing)
  - مثلاً سوال می کنیم ترافیک بخش شمالی بزرگراه فلان چیست؟ در اولی اطلاعات نودها دائم دارند اطلاعات می‌دهند و در کامپیوتر مرکزی اطلاعات آن بخش جدا می‌شوند ولی در دومی نودها خودشان متوجه می‌شوند کدام یک باید اطلاعات ارسال کنند و بقیه اصلاً اطلاعات ارسال نمی‌کنند. اطلاعات را هم بصورت پردازش شده می‌دهند یعنی اطلاعات را Aggregate می‌کنند و یک بسته اطلاعاتی ارسال می‌کنند. این دو حالت حدی هست ولی عموماً چیزی بین این دو است.

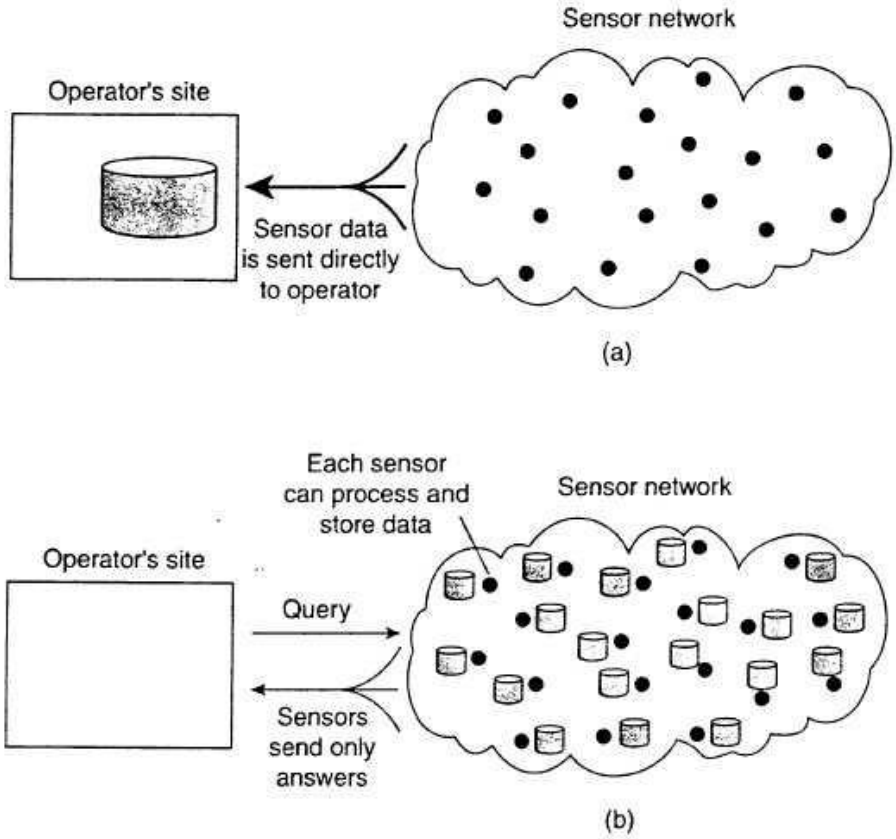


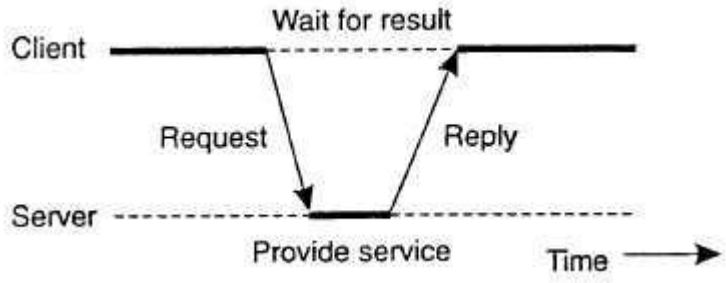
Figure 1-13. Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.

فصل 2: معماري هاي نرم افزار

مطالب فصل 2:

- شیوه‌های معماری یا Architectural Style
  - Layered
  - Object Base
  - Data Centered
  - Event Based
- معماری های سیستم یا System Architecture
  - متمرکز یا Centralized
    - معماری چند طبقه
  - غیر متمرکز یا Decentralized
    - معماری نظیر به نظیر ساخت یافته
      - Cord
      - Can
    - معماری نظیر به نظیر غیرساخت یافته
      - مدلی که هر گره c عنصر همسایه را دارد
      - شبکه‌های Overlay یا روی همگذاری
      - Super Peer
  - ترکیبی (متمرکز و غیرمتمرکز)
    - Edge Server
    - Collaborating Distributed Systems
    - Bit Torrent
- معماری یا میان افزار
  - رهگیرها یا Interceptors
  - نرم‌افزار تطبیقی
- خودمدیریتی یا Self Management
  - کنترل بازخورد

- سیستم توزیعي به لحاظ نرمافزاری قطعات پیچیده اي از نرم افزار است که روي ماشین هاي مختلف توزیع شده اند. روش هاي مختلف براي سازماندهي این قطعات وجود دارد. این سازماندهي معماري را مشخص مي کند. هم سازماندهي منطقي و هم فیزیکي داریم. مثلاً بصورت منطقي همه Component ها را در یك رینگ قرار دهیم. Software Architecture چگونگي سازماندهي Component ها و ارتباط آنهاست؟ در این بخش می‌خواهیم ببینیم هدف سیستم-های توزیعی چه بوده است. هدف این بود که لایه ای بنام Middleware بدست بیاوریم که Application را از Platform جدا کند و از طریق Middleware به Distribution Transparency برسیم.
- **Adaptability:** تطبیق پذیری یعنی این سیستم توزیعی به گونه‌ای طراحی شده باشد که خود را با تغییرات محیطی وفق دهد. گره‌ای اضافه می‌شود, کم می‌شود, پروتکل تغییر می‌کند و ... . برای اینکار نرم‌افزار می‌تواند خودش را مانیتور کند و بر اساس آن واکنش مناسب نشان دهد. علمی است که از بازخورد استفاده می‌کند.
- **System Architecture:** یك مفهوم کلي است برای هر سیستم می‌شود بکار برد. (روشها: مرکزی, غیرمرکزی و ترکیبی).
  - **مرکزي Centralized Architecture**
  - براي توصیف این نوع معماري از مدل Client / Server استفاده مي کنیم که مدلي است قدیمي. Client یك موجودیت Active است و سرور یك موجودیت Passive. زیرا Client آغاز کننده ارتباط است و سرور صبر می کند تا کسی از او چیزی بخواهد. اگر شبکه مطمئن و سریع بود می‌توان از ارتباط بدون اتصال برای Client / Server استفاده کرد. در جاهایي که شبکه مطمئن نیست مي توان از سرویس اتصال گرا استفاده کرد به جای Repeat Request که سربارش زیاد است.



  - **Application Layering:** چگونه یك Application را می‌توان به Client و Server تقسیم کرد یعنی چه بخش Application در Server است و چه بخش در Client. خیلي از این مدل C/S براي دسترسي به بانك اطلاعات ساخته شدند و داراي سه سطح مي باشند:
    - **سطح User Interface:** بخشي که بطور مستقیم با کاربر تماس است. اطلاعات را از کاربر می‌گیرد و اطلاعات لازم را به او می‌دهد.
    - **Process Level:** بخشي که اصل برنامه در اینجاست و هسته برنامه است.
    - **Data Level:** مدیریت داده واقعي. خاصیت Persistence بودن را دارد.

- **مثال:** Internet Search Engine

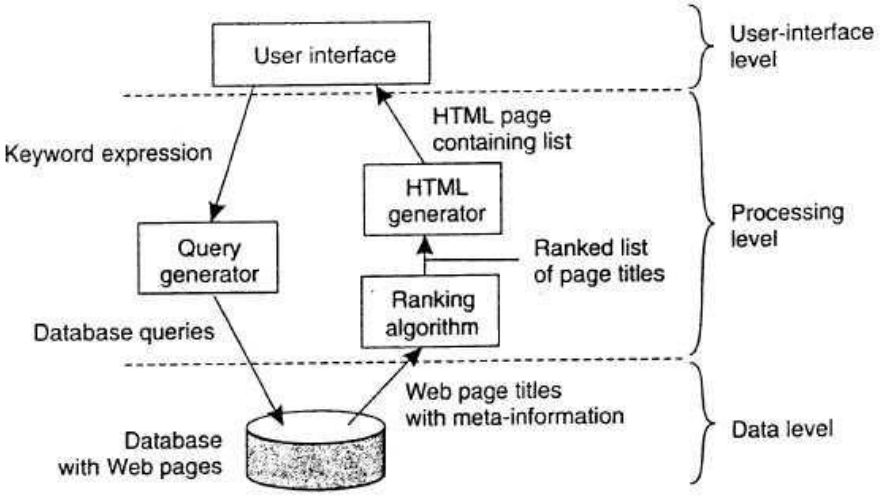
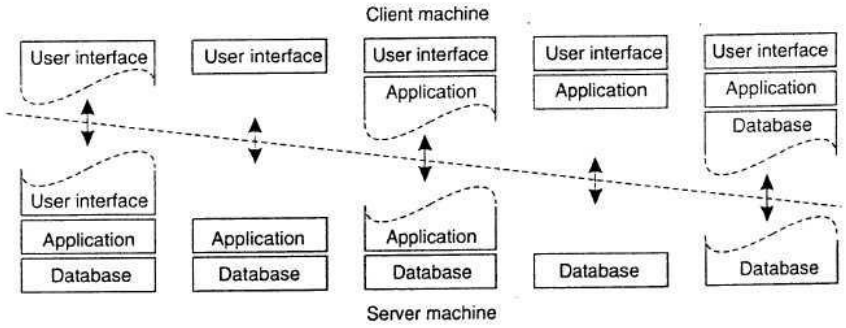


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

- مثال دیگر سیستم تصمیم یار برای کارگذار سهام که یک Front End دارد که همان User Interface است و یک Back End که اطلاعات مالی است. که بین اینها Processing Level وجود دارد که داده مالی را آنالیز می‌کند.
- مشخصه سیستم هاي رابطه اي این است که داده و برنامه کاربردي از هم مستقل هستند. مدل رابطه‌ای ممکن است همه جا خوب نباشد و Object Oriented Database بهتر باشد
- معماري هاي چند طبقه، Thin Client و Fat Client آخری از طریق Caching امکان پذیر است در غیراینصورت Clientهای دیگر به آن دسترسی ندارند.



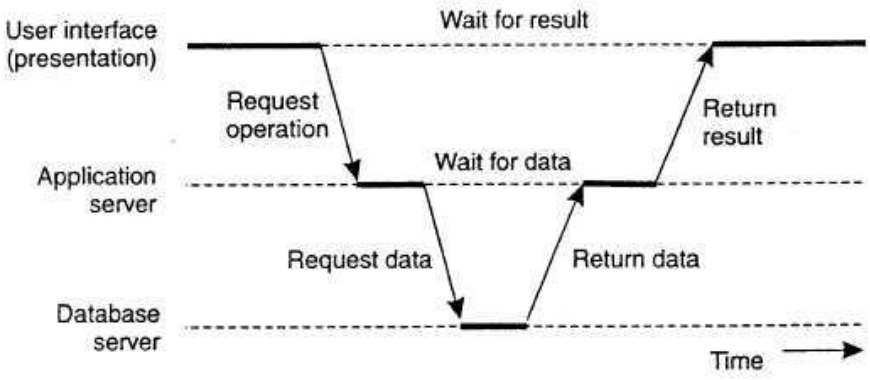


Figure 2-6. An example of a server acting as client.

- مدلهاي Two-Tiered و Three-Tiered نسبت به مدل اول اين شكل بالا سه لايه دارد. بيشتر هم مي‌توان داشت كه به آن Multi-tiered مـی‌گوينـد. ايـن يک تقسيم بندی منطقی است.
- توزيع شدگی بالا را Vertical Distribution يا توزيع شدگي عمودي می‌گويند. توزيع افقی مانند توزيع يـک database در ماشـينهای مختلـف. در واقـع توزيـع عمودی هر لايه در يک ماشين انجام می‌شود ولی در توزيع افقی يک لايه در چند ماشين انجام می‌شود.

- **غير مركزي يا غير متمركز Decentralized Architecture**
  - مشكل سيستم‌های مركزی اين است كه نقطـه مركـزی مـی‌توانـد Single Point of Failure شود.
  - در اين معماري Process ها يكسان هستند و مي توانند گاهي Client و گاهي سرور باشند و هر كدام بخشي از كار را انجام دهند. برای همين به آن Servent می‌گوينـد. اصل و اساس آن Channel ها و Process ها هستند.
  - Horizontal Distribution بحث بعدي است.
  - يك نمونه و مهمترين آنها مدل **Peer to Peer** است. همه مدلها بصورت Symmetric يا متقارن هستند. نودها Process ها هستند و لينكها كانال ارتباطی فرايندها.
  - Overlay Network ارتباط Process ها را نشان مي دهد. شبكه فوقاني اسـت يعنـي ديدگاه منطقي است شبكه اي روي شبكه اصلي. ممكن است دو نود مجـاور از نظـر فيزيكی خيلی از هم دور باشند. اساس Overlay Networks عبارت است از كانالهـا و نودها.

  - **دو نوع Overlay Network مي توان داشت: يك Structured و ديگـري Non Structured.**
  - **Structured Peer to Peer Systems**: برمبناي Distributed Hash Table هستند (DHT). يك سري Process داريم كه نودهای ما هسـتند و روي Data Item هـا عمـل

مي كنند. به هر Data Item يك عدد تصادفي بنام Random Key مي دهيم. براي نودها هم همينطور. مي تواند هر يك Data Item به يك نود يا چندين Data Item به يك نود نسبت داده شود. اين نسبت دادن يك روش كاملا مشخص است يعنی كاملا مشخص است يک Data Item به چه نودی داده می‌شود. اين اعداد بايد منحصر به فرد باشند. Look up هم همينطور مشخص است. اگر يك Request براي يك Data Item وجود داشته باشد مي توانيم Route كنيم به نود مورد نظر. اين سيستم از Hash Table برای دادن id به نودها استفاده می‌کند.

- مثالي از سيستم Peer to Peer: سيستم Chord كه يك Over Lay network دارد و نودها در يك حلقه سازماندهي شده اند. حال ببينيم assign كردن data item ها به نودها چگونه است. مثلاً به يک data item عدد ده می دهيم نگاه می كنيم ببينيم بعد از 10 اولين نودی كه وجود دارد چيست, 12 است پس 10 به نود 12 assign می‌شود. اگر مثلاً نود 12 را در نظر بگيريم data itemهای 8 و 9 و 10 و 11 و 12 اگر باشند به آن assign می‌شوند. پس در واقع زمانی كه يک كليد K را به ما می‌دهند كوچكتريت گره ای انتخاب می شود كه id < k باشد. برای Lookup يعنی ببينيم مثلاً 10 كجاست بايد Successor(10) را پيدا كنيم. حالا ببينيم اين نودها چگونه در يک Overlay Network سازمان دهی می‌شوند. اين را Membership Management می‌گويند يا مديريت عضويت. اين Membership Management شامل دو عمل Join و Leave است.

درج يا Join:

1) اول عدد تصادفي id ايجاد مي كنيم.
2) بعد Successor(id) رارا پيدا مي كنيم. و از او Predecessor را می پرسد.
3) با Successor و Predecessor تماس بگير تا به Successor بگويی كه Predecessor تو من شدم و به Predecessor بگويی Successor تو الان من هستم.
4) بين اينها درج مي شود.
5) بعد اطلاعات نودها اصلاح شود و data itemهای بعدی تحويل گره جديد شود.

ترک يا Leave:

1) نود id ميخواهد ترک كند. هم به Successor(id) و هم به Predecessor(id) اطلاع می‌دهد كه می‌خواهد ترک كند.
2) Data item هايی كه داشته را به Successor (id) تحويل می‌دهد.

در اين شكل 4 بيت استفاده شده است تا 16 نود داشته باشيم.

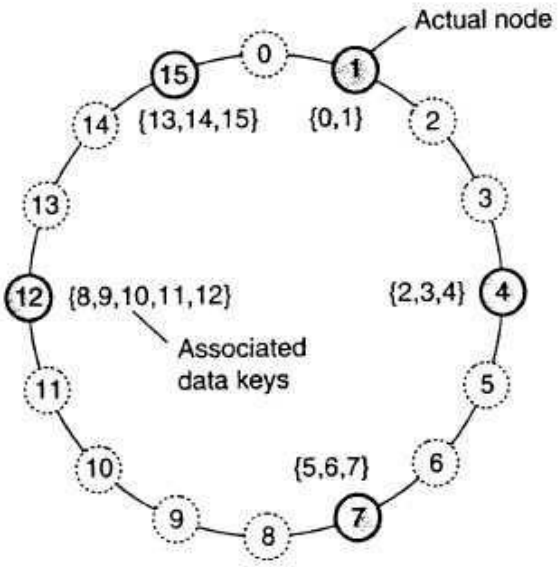


Figure 2-7. The mapping of data items onto nodes in Chord.

- مثال دیگر سیستم CAN (Content Addressable Network) است. CAN یك فضاي D بعدي كارتزين است. این فضا D بعدی به تعداد گره هایی که وجود دارد تقسیم شده است. در اینجا هم data item ها یک random key می‌گیرند. برای مثال از فضای دو بعدی استفاده می‌کنیم. هر نود یا data item در فضای دوبعدی دو عدد دارد. در هر ناحیه در شکل زیر نقطه مرکز را بعنوان آن نود گرفته است. هر data item که درون یک مستطیل بیفتد به آن مستطیل اختصاص داده می‌شود. مثلاً اگر data item بصورت (0.15, 0.25) باشد به گره (0.2, 0.3), assign می‌شود.

:Join

1) به آن یک عدد تصادفی بنام P می‌دهیم. که عدد دو بعدی است.
2) بعد Lookup(p) = Q و منطقه پیدا می‌شود.
3) آن ناحیه را به دو قسمت تقسیم می‌کنیم.
4) P, data item ها خود را از Q می‌پرسد.

:Leave

فرض کنیم نود (0.9, 0.6) می‌خواهد Leave کند پس data item هایی که در آن ناحیه هستند باید منتقل شود.

1) ناحیه به یکی از همسایه‌یش داده می‌شود. مثلاً به (0.9, 0.9). می بینیم خاصیت مستطیل بودن به هم می‌خورد. برای حل مشکل باید سازماندهی تغییر کند. و یک Processی بطور مرتب این سازماندهی را انجام دهد.

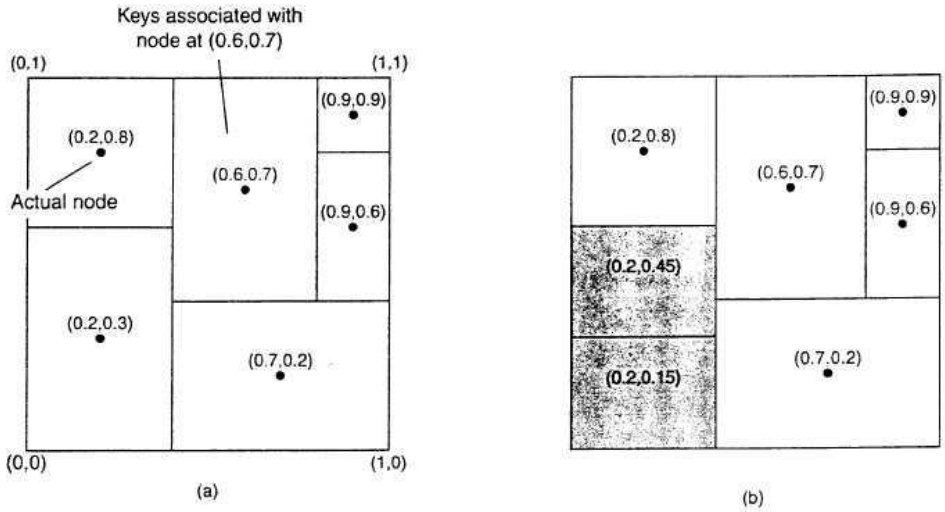


Figure 2-8. (a) The mapping of data items onto nodes in CAN. (b) Splitting a region when a node joins.

- **Unstructured Peer to Peer Architecture:** از الگوریتم تصادفي براي توليد Overlay Network استفاده مي شود. هر نود ليستي از همسايه ها به تعداد C تا را نگه مي دارد. Look up چگونه است؟ از روش Flood استفاده مي شود يعني سيل آسا. اين Overlay Network شبيه Random Graph مي شود. نودها بطور مرتب با همسايه ها تبادل اطلاعات مي كنند تا اطلاعات ليست آنها به روز باشد. هر عنصر ليست شامل يک Id همسايه و يک age است. هر چه age بيشتر باشد يعنی آخرين مراجعه با آن نود قديم بوده است و اهميت نودهايی در ليست كه دارای age بالا هستند كم می‌شود. پس هر ليست بر اساس age مرتب شده است. هر نود تعداد C/2 + 1 قلم اول ليستش را به همسايه می‌دهد. آن يک بخاطر اين است كه اطلاعات خودش را هم می‌دهد. وقتی كه اطلاعات را از همسايه گرفت بر حسب age مرتب می‌كند و زيادی‌ها را Discard می‌كند. هر بار كه اطلاعات را تبادل می-كنند ageها يک واحد افزايش پيدا می‌كند. اما اگر Referenceی به هر يک از data item های ليست شود age آن صفر می‌شود. چه موقعی Exchange انجام می-شود؟ نودی كه در حالت Push است شروع می‌كند. نود ديگر بايد در حالت Pull باشد. دفعه بعد نود بايد تغيير وضيت دهد. اگر غير اين باشد Overlay network بصورت **جزيره‌های مجزا** درمی‌آيند.

درج:

يک سری نود well known داريم با يکی از آنها بصورت اختياری تماس می‌گيرد. خودش را به ليست او اضافه می‌كند و ليست همسايگان او را می‌گيرد.

Leave:

هيچ كاری لازم نيست انجام شود بدون اينكه به همسايه‌ها اطلاع داده شود. زيرا ديگر وجود ندارد كه كسی به او Reference كند و به تدريج از ليست‌ها خارج می-شود.

- **ترکیبي Hybrid از Overlay Networkها**
- **two layered approach to peer to peer overlay network**

می‌توان بصورت ساخت‌یافته و غیرساخت‌یافته در نظر گرفت. لایه بالا ساخت یافته است و لایه پایین غیرساخت یافته. می‌توان به جای age از نزدیکی فیزیکی استفاده کرد که در بالایی به عنوان پروتکل استفاده می‌شود. این ساخت یافته است زیرا معیاری انتخاب کردیم.

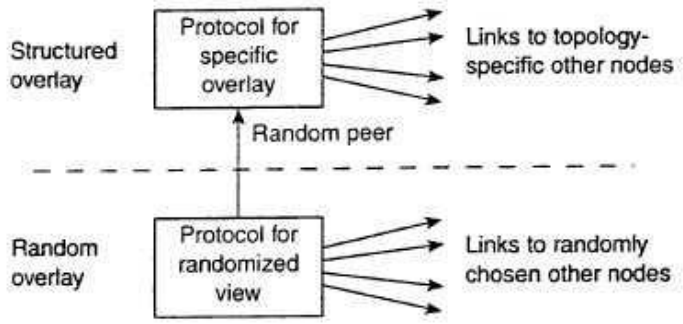


Figure 2-10. A two-layered approach for constructing and maintaining specific overlay topologies using techniques from unstructured peer-to-peer systems.

- مثال: شبکه‌اي توری از نودهاي N*N شبکه دو لایه که Unstructured هست در لایه پایین و Structured است در لایه بالاتر بر اساس فاصله فیزیکي. فاصله دو نود (a_1, a_2) و (b_1, b_2) عبارت است از $d_i = \min(N - |a_i - b_i|, |a_i - b_i|)$ این شبکه را تروس می‌گویند. لزومی ندارد فاصله فیزیکی معیار باشد هر معیار دیگر می‌تواند باشد.

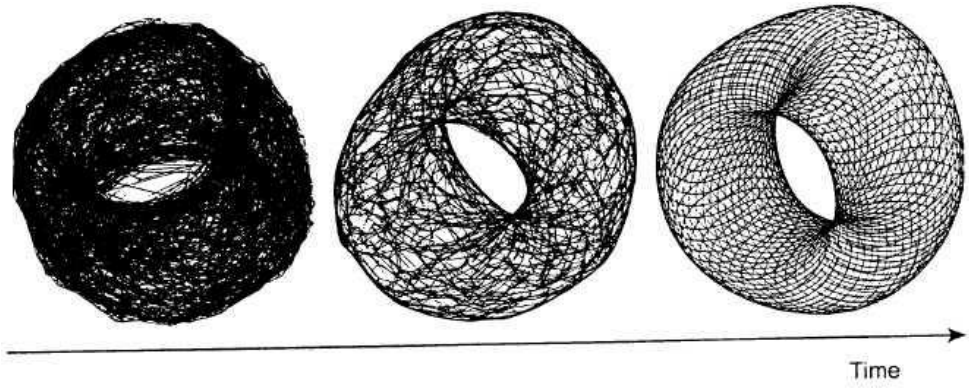
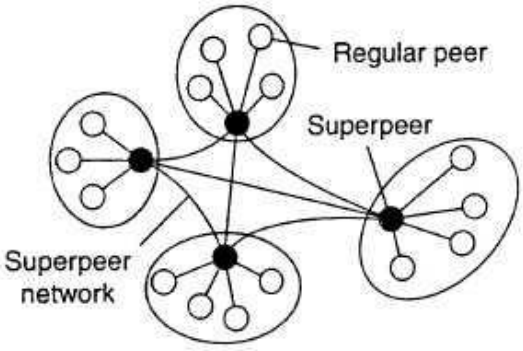


Figure 2-11. Generating a specific overlay network using a two-layered unstructured peer-to-peer system [adapted with permission from Jelasity and Babaoglu (2005)].

- **ساختار Super Peers (در موضوع Unstructured Peer to Peer Architecture)**
  - در شبکه Non Structured تنها راه رساندن اطلاعات به نودهاي مورد نظر Flooding است. در این حالت Scalability سخت مي شود زیرا Flooding سرباز زیاد دارد. در این ساختار برخي نودها را Supper Peer مي گویم و براي هر يك نودهاي عادي را به آن وصل هستند. برای ارسال اطلاعات یک

نود اطلاعات را به Super peer خودش می‌دهد و Super peer به بقیه Super peer ها می‌دهد که تعدادشان خیلی کم است و بعد بدست Peer می‌رسد. خود Supper Peer ها با هم تشكيل يك Overlay Network را مي دهند. اين ساختار سلسله مراتبي است. در لايه اول Super Peer ها هستند. اين ساختار سربار كمتري دارد. اين ساختار مي تواند Fix Super Peer باشد يعني نودهاي Supper Peer ثابت باشند كه پايدار هستند. اين ساختار مشكل قابليت اطمينان دارد. براي حل مي توان از Pairng استفاده كرد يعنى دو نود Supper كنار هم قرار داد و Peer هاي ديگر به هر دو وصل شود. راه حل ديگر اين است كه از election استفاده كنيم.



A hierarchical organization of nodes into a superpeer network.

- Leave كافي است به Supper Peer خود اطلاع دهد.
- Join كردن بايد يك Supper Peer انتخاب كند.
- Supper Peer براي اين است كه مشكل Peer to Peer كه Flooding بود را حل كند.

- **ساختارهاي تركيبي يا Hybrid Architecture**
  - يعني تركيب Centralize و Decentralize تا از مزاياي آنها استفاده كنيم.
  - دو نمونه مي گوييم يكي Edge Server Systems و Collaborating Distributed Systems
  - در Edge Server در Overlay Network نودهايي را لبه مي گوييم. لبه مرز بين شبكه Enterprise (شبكه يك سازمان) و اينترنت واقعي. ارتباط نود كاربران با نود لبه بصورت Client/Server است و لبه ها با هم ساختار Peer to Peer دارند. نودهاي لبه اطلاعات نود كاربران را دارند.

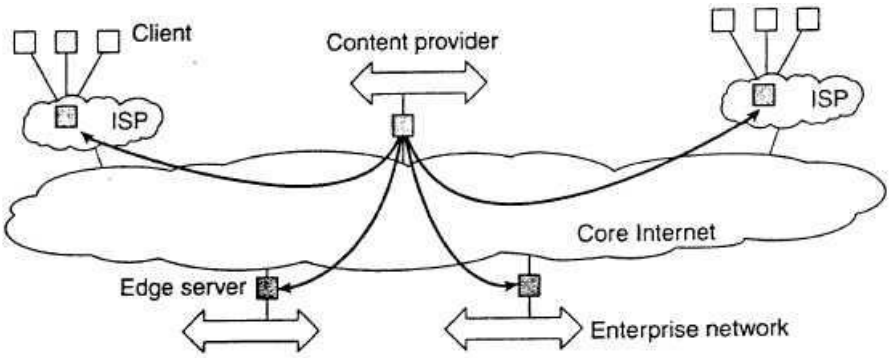


Figure 2-13. Viewing the Internet as consisting of a collection of edge servers.

- **نـوع دیگـر Collaborating Distributed Systems:** تعـدادي نـود داريـم بصـورت Peer. مثلاً براي دانلود فايل بخشي را از هر نود مي گيريم ولي در حال دانلود اگر نود ديگري خواست موظف هستيم آپلود هم بكنيم. يعني همكاري مـي كنـيم هـيچكس هميشه دانلود يا آپلـود نيسـت. ايـن سيسـتم كـاملاً Peer to Peer و Decentralized است. اين تركيب سيستم مركزي و غير مركزي است.

- **مثـال سيسـتم Bit torrent - Collaborate:** بـراى جسـتجو از يـک سـاختار مرکزی استفاده شد ولی برای دانلود از یک سیستم نظیر به نظیر. در ابتـدا یـک Lookup(F) انجام میدهد که روی Bit Torrent Web Page انجام میدهـد ایـن یـک Reference میدهد به یک File Server. داخلی ایـن فایـل سـرور یـک تورنـت فایـل اسـت بـرای فایـل F کـه شـامل اطلاعـات فایـل F اسـت. از جملـه اطلاعـات Reference بـه Tracker اسـت کـه دارای لیسـتی اسـت از N نـود کـه فایـل F را ذخیره کرده‌اند. Tracker یک موجودیـت مرکـزی اسـت. پـس از اینکـه فایلهـا پیـدا شـدند سـاختار می‌شـود Peer to Peer. بعد از اینکه نود ما دانلـود کـرد آن هـم بـه لیسـت Tracker اضافه می شود. اگر یک نود همکـاری نکـرد نودهـای دیگـر بـه او سرویس مناسب نمی دهند و جریمه اش می کنند.

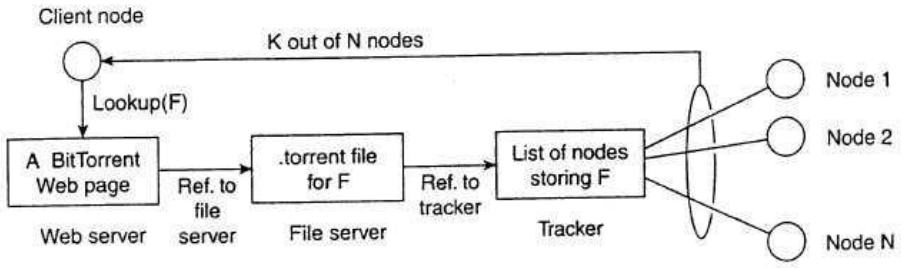


Figure 2-14. The principal working of BitTorrent [adapted with permission from Pouwelse et al. (2004)].

- **Architectural Styles:** چگونگي قرار گرفتن و ارتباط Component ها را بيان مي كند.
  - **از Logical Organization ناشي مي شود.**
  - معماري نرم افزار عبارت است:

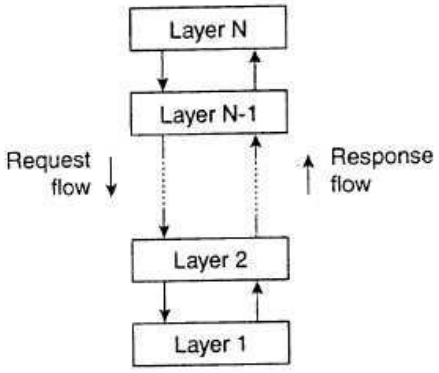
    1- تعدادي كامپانت نرم افزاري

2- چگونگي ارتباط آنها

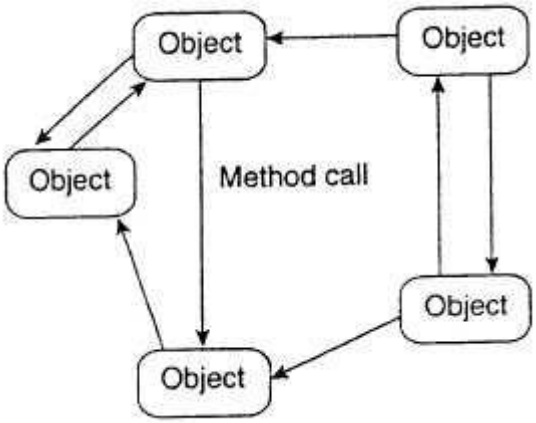
3- چه نوع داده اي را رد و بدل مي كنند

4- چگونه تشكيل يك سيستم را مي دهند

- **Component که همان Process است وقتی به اجرا در می‌آید:** يك واحد پيمانه اي (Modular Unit) است داراي Interface هاي تعريف شده. و Replicable مي باشد يعنی در صورت نياز می‌توان آن را جايگزين كرد كه همان Interface را دارد ولی مثلاً به شكل ديگری پياده سازی شده است.
- **Connection:** اينها چگونه اينها متصل شده اند. مثلاً يك Procedure call يك Connector است يا Message passing براي ارتباط دو مولفه نرم افزاري و Streaming Data كه زمان در آن تداخل دارد و براي صوت و تصوير است. Connector مكانيزمی است كه اينها را به هم متصل می‌كند.
- تركيب هاي مختلف مولفه ها و Connection ها **Architectural Styles** را مشخص مي كند. عبارتند از لايه‌ای, Object-Based, Data Centered و Event-Based
  - **Layered Architecture يامعماري لايه اي**
    - ساختار لايه اي ساده دارد. همواره لايه بالاتر از سرويس لايه پايين تر استفاده مي كند. برعكس امكان ندارد. Request و Result.
    - اين ساختار بسيار كاربرد دارد در Computer Networking



  - **Object Based Architecture يا شيئي محور**
    - كامپانت ها از Object ها تشكيل شده اند كه به هر گونه اي به هم متصل شده اند. و هر Object داراي چند Method است و برای ارتباط متد هاي يكديگر را Invoke مي كنند.

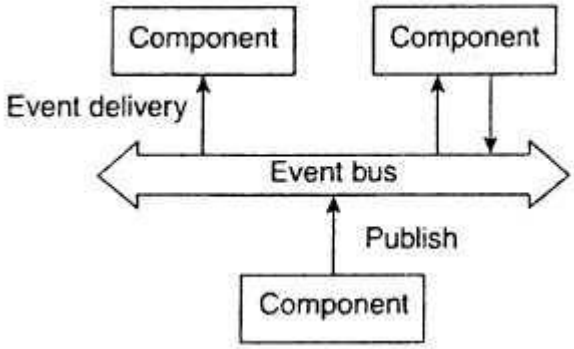


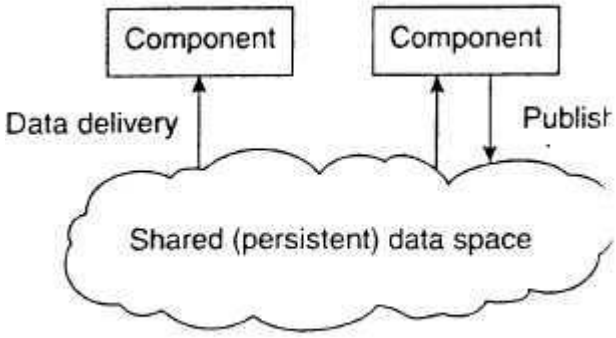
- **Data Centered Architecture یا داده مرکز**

- ساختار ساده اي است. يك ذخيره مشترك بين همه مولفه ها هست مثلاً يك فايل مشترك يا حافظه مشترك يا Web Based Distributed System. يكي آنجا مي نويسد و يكي مي خواند پس ارتباط بين اينها برقرار مي شود. اين ارتباط فقط بر اساس Read و Write است.

- **Event Based Architecture یا رویداد محور**

- ارتباط Process ها بر اساس انتشار يك Event است. يكي از روش ها Published / Subscribed Systems است. مي توان يك Bus داشت كه مفهوم منطقي است. مثلاً اطلاعي هست كه دماي منطقه اي را مي دهند كه به هر كس كه Subscribe است مي رسد. قرار دادن Event را روي Bus مي گويند Publish و گرفتن را مي گويند Deliver. ارتباط بر اساس Subscription است. هر كدام مي تواند Publish يا Deliver كند. هر Component براى اينكه اطلاعات را بگيرد بايد حتما در آن موقع در حال اجرا باشد نمى‌تواند بعداً به اجرا در بيايند و آن Event را دريافت كنند. در حاليكه در مدل قبل (Common Repository) اينگونه نيست. تركيب اين يعنى Data Centered و Event-Based را Shared Data Spaces مى گويند.





(b)

- **Shared Data Spaces ترکیب دو مورد بالاست.**

- هدف در سيستم توزيعي اين بود كه لايه اي بنام Middleware بوجود بياوريم كه Application را از Platform جدا كند و جزئيات Platform را از ديد Application مخفي كند.
- Adaptability يا تطبيق پذيري يعني باتغييرات محيطي در شبكه و پروتكل ها خودش را وفق دهد و مجبور نشويم Component ها را تغيير بدهيم.
  - نرم افزار خودش را مانيتور كند. و واكنش مناسب نشان دهد. اين كار از طريق Feedback Control Loop انجام مي شود.
- **مرور**
  - Architecture عبارتست از تعدادي Component و ارتباط بين آنها.
  - بر اساس تعريف بالا Architectural Styles را مشخص كرديم.
    - Layered
    - Object Based
    - Data Centered
    - Event Based
  - از لحاظ سيستمي اگر به معماري نگاه كنيم چگونه است؟ دو روش وجود دارد. (System Architecture)
    - Centralized Architectures: براي توضيح از مدل Client / Server استفاده شد. مشكلي كه دارند نقطه اي كه مركز است Single Point of Failure است.
    - Decentralized Architectures: مهمترين آنها سيستم هاي Peer-to-Peer است.
    - Component ها همه Process ها هستند.
    - Symmetric يا متقارن در مقابل Client/Server است.
    - در مدل Peer to Peer پردازش ها بصورت Overlay Network سازماندهي شدند. نودها Process ها هستند و لينك‌ها كانال ارتباطي اينها هستند.
    - براي ساختارهاي Peer to Peer دو نوع Overlay network مي توان در نظر گرفت. Structured و Unstructured.
    - دو مثال از Structured، يكي Chord و يكي CAN
    - Two Layered Approach لايه پايين Non Structured و ...

- **معماری در مقابل Middleware**
  - **هدف از میان افزار:** مهمترین هدف شفافیت بود. میان افزار هم از یک Architectural Style پیروی می کنند. مانند Object Base یا Event base یا ... . اما اینکار محدود کننده است لذا دنبال Middlewareهای تطبیق‌پذیر می‌رویم.
  - **رهگیر یا Interceptor:** یک ساختار نرم افزاری است که میتواند اجرای برنامه را قطع کند و اجازه میدهد کد بخصوصی اجرا شود. بعنوان نمونه از Interceptorها جهت تطبیق‌پذیر کردن Middlewareها استفاده می‌کنیم. یک مثال می‌زنیم. یک Middleware بصورت Object-based را در نظر بگیریم. یک Application یک سرویسی می‌خواهد که روی ماشین دیگری است ما می‌خواهیم همه چیز برای این Application محلی جلوه کند, حال زمانی که شرایط تغییر کند و نسخه‌های متعدد از Server Application داشته باشیم. پیغام باید به همه اینها برود در اینجا از Interceptor استفاده کنیم و Middleware را با شرایط جدید انطباق دهیم. و Call را مثلاً 5 تا می‌کند. یا ممکن است یک Interceptor در پایین بخواهیم زیرا شرایط تغییر کرده است. در واقع Interceptorها روشی است برای اینکه بتوان Middleware را تطبیق‌پذیر کرد.

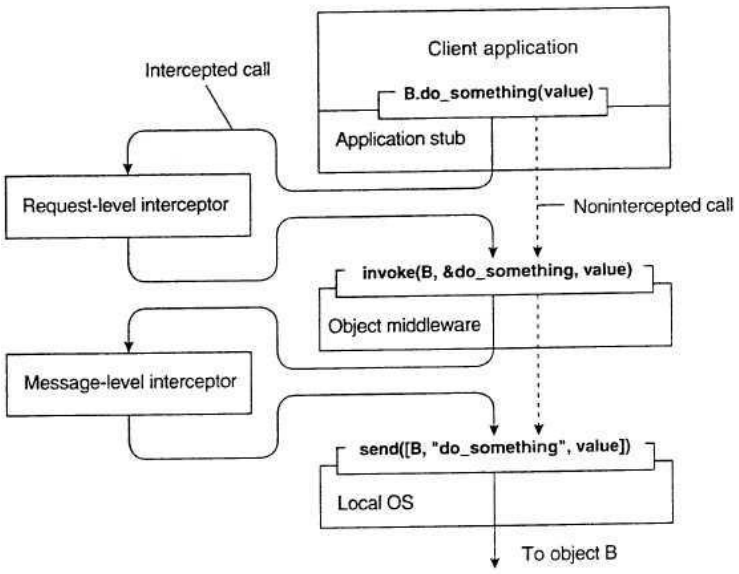


Figure 2-15. Using interceptors to handle remote-object invocations.

  - **رویکرد عمومي به نرم افزار تطبیقي**

اصولاً چه چیزی تغییر میکند؟ یکی می‌تواند Mobility باشد, QoS, Hardware Failing و ...

سه تکنیک اساسي در مورد تطبیق نرم افزار:

1) **تفکیک موارد:** یعنی چیزهایی که اهمیت دارد را جدا کنیم زیر Adaptive Software کار سختی است مثلاً Functionality را از Extra Functionality جدا کنیم. مثلاً هیمنکه فایل سرور سرویس می‌دهد Functionality است اما افزودن کارایی می‌شود Extra Functionality. یا در نظر گرفتن Security و FT بودن.
2) **انعکاس محاسباتي:** یعنی این نرم‌افزار یا Middleware خودش را چک کند و بسته به شرایط خودش را تغییر و تطبیق دهد.

3) **طراحي مولفه-محور**: اگر اینطور باشد می‌توان تطبیق‌پذیری را در انتخاب صحیح Componentها و ارتباط آنها داشت.

- **چرا Adaptive Middleware لازم است؟**

1) سیستم های توزیعی نمي توانند Shutdown شوند.
2) باید راه حلهایي داشته باشیم که Component ها را همینطور که سیستم کار می کند تغییر بدهیم.
3) سیستم های توزیعی باید توانایي ایجاد تغییر در محیط را داشته باشد.

- **خود مدیریتی:** یکی از موادی که گفتیم این بود که Middleware خودش را چک کند. این را Self Management می‌گویند.خوددرمانی, Self Configuring, Self Optimizing نیز شامل هست. چگونه می‌توان این کار را کرد؟ با استفاده از feedback control model که در صنعت, هواپیما و ... استفاده می‌شود. حالا در یک سیستم نرم‌افزاری هم از آن استفاده می‌شود.
- **مدل کنترل بازخورد** ابتدا یک مدلی از سیستم بدست می‌آوریم. هر سیستم یک ورودی دارد یک خروجی. علاوه بر ورودی عوامل ناخواسته هم به سیستم وارد می‌شود مثلاً noise و باعث می‌شود نتوان خروجی صحیح را گرفت. چجوری اینکار را می‌کنند جروجی را می-گیرند و آنالیز می‌کنند بر اساس یک الگوریتم کنترل و یک ورودی که ورودی اولیه را تصحیح می‌کند میدهد. به این feedback می‌گویند زیرا از خروجی می‌گیرد و به ورودی می‌دهد.

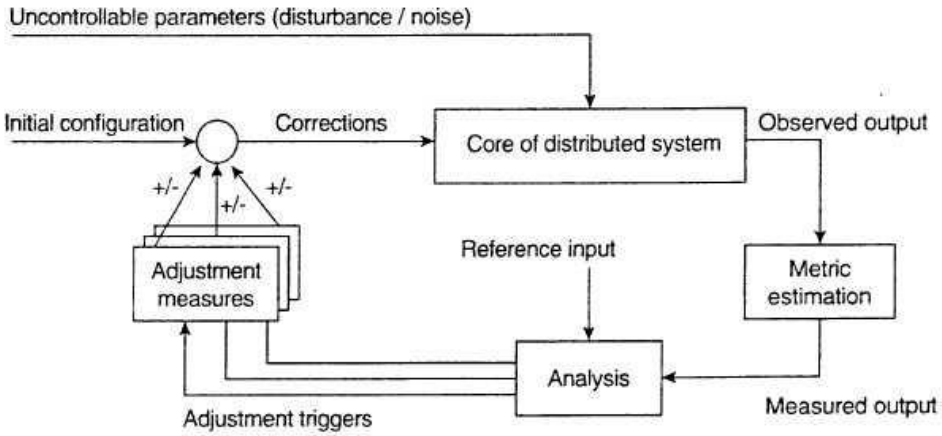


Figure 2-16. The logical organization of a feedback control system.

فصل 3: فرایندها

مطالب

- Threads
- Virtualization
- Client
- Server
- مهاجرت کد

روش‌های مهاجرت کد

مهاجرت کد و منابع محلی

مهاجرت در سیستم‌های ناهنگن

خلاصه: چیزی که توزیع شده است در واقع Process ها هستند و اینها هستند که در ماشین‌های مختلف توزیع شده است. در واقع واحد کار Process است.
وظیفه سیستم عامل زمانبندی و مدیریت Process هاست. از دیدگاه سیستم توزیعی مسائل دیگر مانند Multi Threading مطرح می‌شود تا بطور موثرتری Client/Server را سازماندهی کرد. همچنین از طریق Multi Threading عملیات Communication و Processing را همزمان انجام داد تا کارایی بهتری بدست آید. بعد در باره Virtualization صحبت می‌شود. نرم‌افزارهای مهم طول عمرشان بیش از Platform است برای اینکه بشود از این نرم‌افزار در Platformهای جدید استفاده کرد از مفهوم Virtualization کمک می‌گیریم. بعد در باره سازمانهای Client/Server بررسی می‌شود. و درباره طراحی آنها صحبت می‌شود. مطلب دیگر Moving Process است یعنی انتقال Process به ماشین دیگر به دلیل یک Performance و دیگری Fault Tolerance. بحث دیگر Code Migration است که قبل از اجرا مهاجرت انجام می‌شود تا Scalability و Dynamic Configuration را بدست آورد.

- **Threads**
  - هر Process شامل یک مسیر اجرایی است. این مسیر اجرایی را Thread می‌نامند که بطور سنتی هر Process دارای یک Thread است که سیستم عامل بین Processها سوییچ می‌کند که در هر بار سوییچ این اطلاعات مربوط به یک Process ذخیره شود و اطلاعات Process دیگر Load شود:
    1) CPU State شامل محتویات Registerها, Program Counter, Stack Pointer و ...
    2) اطلاعات MMU Map یا Memory Management Unit Map
    3) TLB را باید تخلیه کنیم یعنی Translation Look aside Buffer Flush

  - آیا می‌توان در یک process بیش از یک Thread داشت؟ بله به آن می‌گویند Multi Thread که می‌دانیم در هر لحظه فقط یک Thread درون یک Process در حال اجراست اما اگر آن Thread متوقف شد Thread دیگر درون همان Process می‌تواند کار کند. مثلاً Thread اول در حال کار است و تقاضای Port می‌دهد و متوقف می‌شود در اینجا برای کارایی بهتر و بیکار نبودن

CPU, Thread دوم را که عملیات Processing دارد را به اجرا در می‌آوریم. و بعد که دومی متوقف شد اولی را ادامه می‌دهیم. پس داخل یک Process می‌تواند فعالیت‌های بیشتری انجام داد و کارایی بهتری بدست آورد. ضمنا باید توجه کنیم Thread Switching ها ارزان‌تر است نسبت به Process Switching که تنها چیزی که باید Store و Load شود CPU State است. زیرا اینها داخل یک Process هستند و Shared Memory دارند. ضمناً دقت کنیم تنها راه ارتباط بین Processها IPC یا Inter Process Communication است. یعنی System Call داشته باشیم. در حالیکه برای Thread بجز این می‌توان از Shared Memory هم استفاده کرد. اگر پردازنده‌های بیشتری داشته باشیم می‌توان هر نخ را به یک پردازنده داد و کارایی بالاتری بدست آورد. یعنی انجام پردازش موازی.

نخ‌ها حتی در سیستم‌های غیر توزیعی هم می‌توانند مفید باشند. مثلاً یک برنامه کاربردی صفحه گسترده جایی منتظر ورود داده هستیم و جایی محاسبات بسته به سلولها انجام می‌شود اینها را بصورت چند نخی می‌توان انجام داد که یکی منتظر عملیات I/O است و نخ دیگر محاسبات انجام می‌دهد.

- **Thread Implementation**

دو روش پیاده سازی دارد:

1) User Lever یعنی بدون اطلاع سیستم عامل
2) System Level یعنی سیستم عامل اینکار را انجام دهد.

- معمولاً پیاده سازی در User Lever کارایی بالاتری دارد زیرا وقتی در سیستم عامل پیاده شود باید خیلی عمومی پیاده سازی شود چون باید هر کاربردی را پشتیبانی کند اما در System Level برای کاربرد خاصی طراحی می‌شود. مشکل User Level این است: زمانی که یک Blocking System Call انجام شود باید Thread متوقف شود تا مثلاً سیستم عامل اطلاعاتی را از دیسک بیاورد در این حالت سیستم عامل این Process را قطع می‌کند و Process Switch انجام می‌دهد و می‌رود سراغ Process بعدی چون نمی‌داند نخ دیگری در همان Process اولیه وجود دارد. استفاده از Non blocking system call دشوار است.

پس کدام را انتخاب کنیم User Lever کاراست ولی این اشکال را دارد از طرفی System Level اصلاً Efficient نیست ولی این اشکال را ندارد و می‌تواند به جای process switch عمل Thread Switch انجام دهد. شاید بهترین راه حل ترکیب اینها باشد. دو روش ترکیبی وجود دارد:

1) LWP یا Light Weight Process
2) Scheduler Activation

- **LWP (Light Weight Process)**

قرار است ترکیبی باشد هم از User Level و هم از System Level استفاده کند. یعنی سیستم عامل همانگونه که می‌تواند Process ایجاد کند می‌تواند LWP هم ایجاد کند که داخل یک Process است و سوییچ کردن بین اینها فقط شامل CPU State است. در واقع یک نخ ایجاد می‌کند. سیستم عامل برا ی Process ها عمل زمانبندی هم انجام می‌دهد یعنی می‌گوید الان نوبت کیست بعد کی و ... . اما می‌شود **زمانبندی را به User lever داد.**

دیگر اینکار در سیستم عامل انجام نمی‌شود. داخل Process توسط سیستم عامل LWP ایجاد می‌شود و Thread ها را به اینها Assign کنیم. ما به LWP می‌گوییم چه کدی را اجرا کن؟ مسئله زمانبندی چه می‌شود؟ کدی که داخل LWP است همان Scheduler است. زمانبندی نگاه می‌کند ببیند نوبت کدام نخ است و اجرا می‌شود. برای اینکه Schedulerها کار کنند باید به اطلاعات مشترک یعنی Shared Data دسترسی داشته باشند. چون حافظه مشترک دارند به راحتی می‌توان به Shared Data دسترسی داشت. برای اینکار باید Mutual Exclusion را به کار گرفت یعنی وقتی یکی جدول را می‌خواند تا ببیند نوبت کیست دیگری همزمان جدول را نخواند. زیرا وقتی جدول را می‌خواند تغییری هم در جدول می‌دهد. در اینجا می‌توان blocking system call هم انجام داد در این حالت چون سیستم عامل از وجود LWPها مطلع است نوبت را به LWP بعدی می‌دهد تا او نخ بعدی را انتخاب کند.
در این روش ساختن LWP ها توسط سیستم عامل است و زمانبندی از طریق User Lever است.

- **Scheduler Activation**

داخل هر Process کدی داریم به اسم Scheduler. یک نخ در یک process در حال اجراست و یک blocking system call انجام می‌دهد. سیستم عامل هیچ اطلاعی از نخ ندارد. پس سیستم عامل وقتی این blocking system call را دید شروع به انجام سرویس می‌کند اما به جای اینکه کنترل را به یک Process دیگر بدهد کنترل را برمی‌گرداند به همان Process و یک کدی بنام Scheduler را که داخل همین process است Call می‌کند زیرا زمان این Process تمام نشده است. و آن Scheduler نخ بعدی را به کار می‌اندازد. برای همین به این روش می‌گویند Scheduler Activation. البته این Call چون از پایین به بالا است می‌گویند Up Call. این روش هم مشکل blocking system call ندارد ولی بخاطر Up call روش مناسبی نیست زیرا در سیستم‌های لایه‌ای Call همیشه از لایه بالا به پایین است.

- **کاربرد Thread در Distributed Systems**

هم از دیدگاه Client و هم از دیدگاه Server اینکار را با مثال انجام می‌دهیم و فرض می‌شود Client یک Web Browser است.

از دیدگاه Client:

یک تقاضایی برای یک Web Server می‌دهد و او Reply می‌دهد و سپس Client اطلاعات را نمایش می‌دهد. یک Web page شامل تصویر, متن, جدول و ... است. همه اینها در یک مرحله انجام نمی‌شود و تعدای TCP Connection برقرار می‌شود و کم‌کم و به تدریج نمایش می‌دهد. آیا امکان دارد Client چند نخ داشته باشد و Connection های موازی بفرستد؟ و بعد Reply بگیرد؟ بله در اینصورت سریعتر هم خواهد بود حتی اگر از Web Server چند کپی داشته باشد می‌تواند Connection با

Web Serverهای مختلف داشته باشد و سرعت بالاتر برود. اشکال این است که اگر همه مشتری‌ها بخواهند اینکار را بکنند Load سرور زیاد می‌شود.

از دید Server:

برای مثال یک File Server را در نظر می‌گیریم. Clientهای مختلف تقاضای ارسال فایل می‌کند. اگر فایل سرور فقط یک نخ اجرایی داشته باشد تقاضا را می‌گیرد بعد می‌رود جلو یک تقاضا به دیسک می‌دهد منتظر پاسخ می‌ماند و بعد نتیجه را به مشتری بازمی‌گرداند و بعد دوباره منتظر تقاضا می‌ماند و این کار را تکرار می‌کند. زمانی که فایل سرور دارد به مشتری اول سرویس می‌دهد اگر Request دیگری بیاید باید wait کند. یعنی اگر سرور بصورت Single Thread باشد به تقاضاها بصورت **سریال** پاسخ داده می‌شود.

آیا می‌شود فایل سرور را بصورت Multi Thread انجام داد تا چندین Request را بصورت همزمان پاسخ دهد؟ جواب مثبت است. یک نخ را بنام Dispatcher می‌گذاریم که تمام Request ها به آن می‌رسد. بعد این Dispatcher تقاضای اولی را می‌گیرد Thread خالی پیدا می‌کند و به آن می‌دهد تا اجرا کند و پاسخ را باز گرداند و بلافاصله Request بعدی را می‌گیرد. به همین ترتیب برای بقیه تقاضاها. این نخها را Worker Thread می‌نامند. اگر Worker Threadها خیلی به دیسک مراجعه کنند این سیستم خیلی بهتر از Single Thread نخواهد بود. پس در صورتی خوب است که دسترسی موازی به دیسک ممکن باشد. مانند ساختارهای Raid.

سوال دیگر این است که آیا ممکن است من سرور Single Thread داشته باشم ولی کارایی Multi Thread داشته باشم؟ پاسخ مثبت است. به اینصورت که call هایی که توسط آن نخ انجام می‌شود بصورت Non blocking باشد. یعنی Thread تقاضای اول را بگیرد به دیسک بدهد ولی منتظر پاسخ Disk نماند و برود سراغ Request بعدی. یعنی این نخ مانند ماشین حالت محدود عمل می‌کند. یعنی وقتی تقاضایی به دیسک می‌دهد در یک جدول درج می‌کند ولی منتظر پاسخ نمی‌ماند. از طرف دیگر هر موقع دیسک پاسخ را آماده می‌کند به نخ اطلاع می‌دهد و نخ می‌رود سراغ جدول و مشاهده می‌کند پاسخ مربوط به کیست و به مشتری پاسخ می‌دهد.

سوال دیگر این است: آیا امکان دارد در یک سرور چند process داشت که یکی Dispatcher Process است؟ بله باز در این سیستم داریم موازی سازی انجام می‌دهیم. و هر Process می‌تواند دسترسی به دیسک داشته باشد و باید بتوان به دیسک دسترسی موازی داشت. آیا این معادل سیستم Multiple Thread است یعنی می‌توان همان Performance را داشته باشد؟ جواب منفی است. زیرا Multiple Thread حافظه مشترک دارد و داخل یک Process هستند. اما اینجا هر کس Cache خودش را دارد. این باعث می‌شود نتوان کارایی Cache مشترک را داشت پس کارایی کمتر است. جدول زیر مقایسه این روشها است. استفاده از blocking system call و Parallelism جنبه مثبت است.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Figure 3-4. Three ways to construct a server.

تا کنون گفتیم با داشتن Multiple Thread و یا Multi Process می‌توان موازی‌سازی را یا بصورت واقعی یا غیر واقعی نشان داد. اگر یک پردازنده داشته باشیم تصور موازی سازی را بوجود می‌آوریم از طریق Resource Virtualization.

- **Virtualization**
  - برنامه‌های کاربردی معمولاً عمر بیشتری از Platform خودشان یعنی یعنی سخت افزار و سیستم عامل دارند و باید بتواند روی Platformهای جدید قابل اجرا باشد. شکل زیر سازمان عمومی یک برنامه, واسط و سیستم و سیستم مجازی را نشان می‌دهد. Interface می‌تواند شامل API و System call باشد.

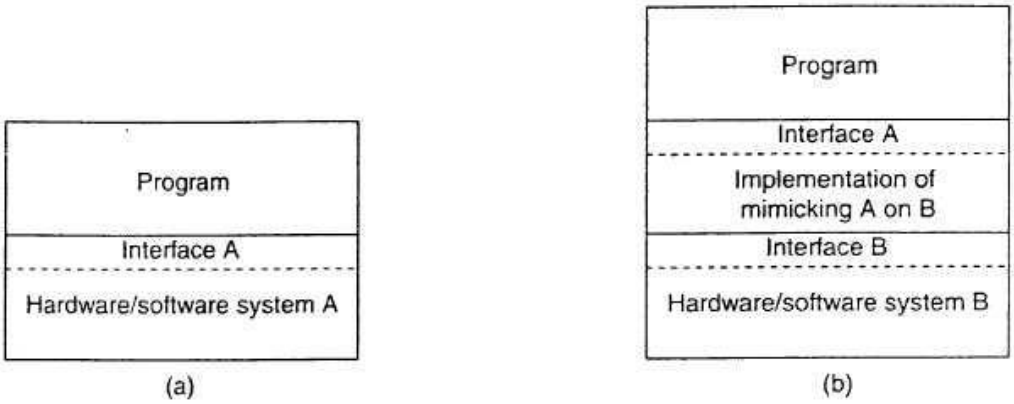


Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

Interface شامل چهار مورد است:

1. دستورالعمل‌های ماشین یا Machine Instructions
2. دستورالعمل‌های ممتاز یا Privileged Instructions که سیستم عامل یا برنامه‌های ممتاز می‌توانند استفاده کنند. برنامه‌های معمولی نمی‌توانند.
3. System Callها
4. Application Programming Interface (API)

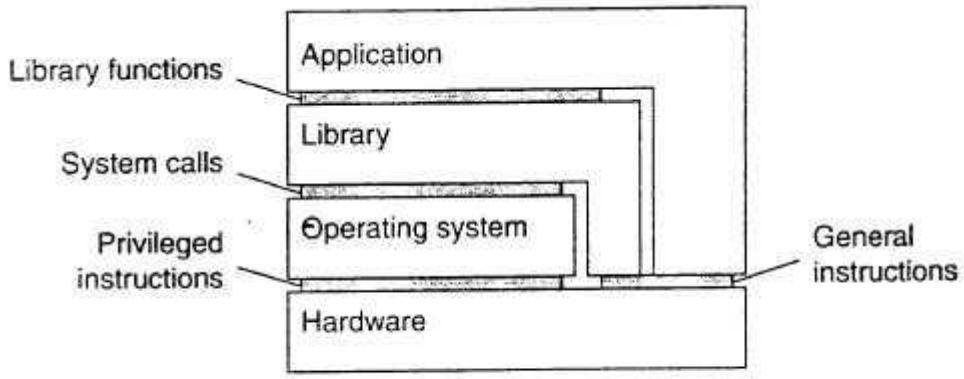


Figure 3-6. Various interfaces offered by computer systems.

در قسمت b یک لایه اضافه قرار می‌دهیم که ماشین A را روی ماشین B بصورت نرم‌افزاری پیاده سازی می‌کند. Virtualization خواص دیگری هم دارد. مثلاً امروزه Pervasive Networking داریم یعنی شبکه فراگیر حتی وسایل خانه می‌تواند در این شبکه باشد. روی چینن شبکه‌ای که همه نودها با هم فرق دارند این تفاوت‌ها را از طریق Virtualization کم کنیم تا کارمان راحت شود. تا Portability و Flexibility داشته باشیم. بعنوان مثال فرض کنیم

چندین Edge Server داریم که اگر از مجازی سازی پشتیبانی کنند و بار یکی زیاد است می‌خواهیم یک کپی از این را به Edge Server دیگر انتقال دهیم. احتمالاً Platformها با هم متفاوتند اما از طریق مجازی‌سازی می‌توان کل آن را منتقل نمود.

- **معماری ماشین‌های مجازی**

می‌توان دو نوع مجازی سازی داشت:

1. Process Virtual Machine
2. Virtual Machine Monitor (VMM)

شکل زیر این دو روش را نشان می‌دهد:

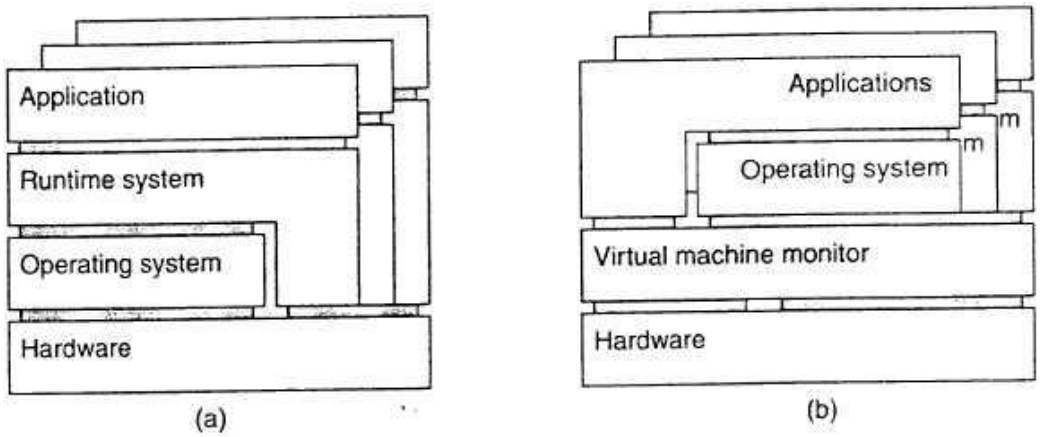


Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

در Process Virtual Machine سخت افزار و سیستم عامل ثابت هستند و از طریق یک Runtime System کار Virtualization را انجام می‌دهیم. و Application روی Runtime System قرار دارد قبلاً دیدیم Application به چه چیزی احتیاج داشت؟ یکی به General Instruction و دیگری به Library Function ها که اینها را Runtime System تامین می‌کند. پس چیزی که Application می‌بیند واقعی نیست چیزی است که Runtime System به او می‌دهد اما خود Runtime System از امکانات System Call و General Instruction استفاده می‌کند تا ماشینی که می‌خواهد را تقلید کند. پس Application هم Library Functionهای مورد نظر خودش را می‌تواند در Runtime System پیدا کند و هم Instruction Setی که متعلق به ماشین قبلی بود در Runtime System پیدا کند. مشاهده می‌شود که برنامه کاربردی اصلاً با ماشین اصلی ارتباط ندارد. از مجموعه Runtime System و برنامه کاربردی اینجا چند تا موجود است.

اما در Virtual Machine Monitor برای نرم‌افزارهای باقی مانده از گذشته شکل بهتری است. این شکل شامل یک سخت‌افزار جدید است و یک Virtual Machine Monitor یعنی یک نرم‌افزاری را روی این ماشین برقرار می‌کنیم که دقیقاً تقلید ماشین دیگری را می‌کند. یعنی مجموعه دستورالعملهای عمومی و Privilege را تامین می‌کند. این کار از طریق Interpretation و هم Emulation ممکن است. Interpretation یعنی کاراکتر به کارکتر بخوانیم و اجرا کند و یا Emulation که سریعتر است و به ازای هر دستورالعمل ماشین قبلی

یک دستور یا چند دستور ماشین جدید را اجرا کنیم. در اینجا مجموعه‌ای که روی این ماشین تکرار می‌شود سیستم عامل و Application ها هستند یعنی می‌توان در آن واحد چندین برنامه مختلف روی ماشین داشت و آنها را اجرا کرد. چون بسیاری از نرم‌افزارهای قدیمی روی سیستم عامل خاصی اجرا می‌شوند. و به جای تقلید System callهای آن خود سیستم عامل را اینجا قرار می‌دهیم.

- **ساختار Clientها و سرورها**
  - Processها در قالب Clientها و سرورها سازماندهی می‌شوند.
- **Clients**

Network User Interface هایش چگونه است؟ دو انتخاب وجود دارد:

1. Application Specific Protocol
2. Application Independent Protocol

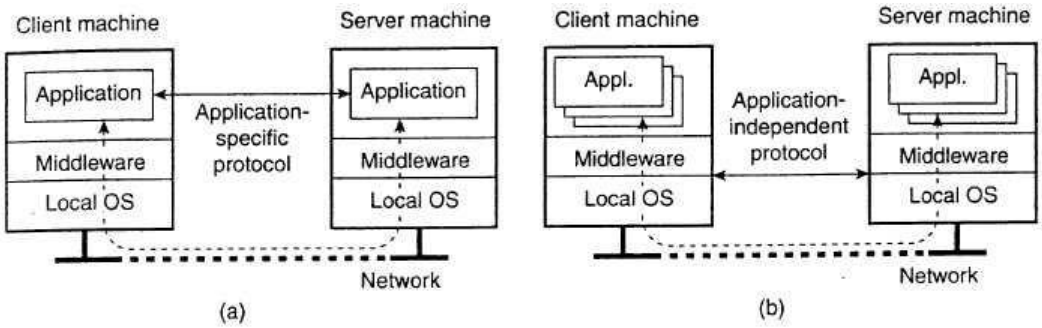


Figure 3-8. (a) A networked application with its own protocol. (b) A general solution to allow access to remote applications.

در Application Specific Protocol برنامه‌های کاربردی می‌توانند مستقیماً با هم تماس بگیرند. لایه‌های پایینی شامل Middleware و Local OS است. سرویسی که این لایه‌ها به برنامه کاربردی می‌دهند همان عبور پیغام است. اما در دومی ارتباط از طریق Middle ware برقرار می‌شود. این پروتکل را می‌توان گفت Application Independent Protocol هست. در اولی می‌شود کارایی بیشتری بدست آورد همیشه بین Generality و Performance یک Trade off وجود دارد. دومی شفافیت بدست می‌آورد.

- **نرم‌افزار سمت مشتری چه تاثیری روی شفافیت توزیع دارد؟**

خیلی کارها را می‌توان سمت مشتری انجام داد. Client و Server شامل Middleware نیز هستند. در اینجا تاکید روی Middleware است. این چه تاثیری روی شفافیت دارد؟ اولاً می‌تواند Access Transparency را تامین کند به این معنا که ماشین سمت سرور و ماشین سمت مشتری می‌تواند متفاوت باشند با نمایش اطلاعات متفاوت مثلاً یکی little Endean و دیگری big Endean. یا فایل سیستم. Location Transparency را انجام میدهد. مثلاً اگر مشتری یک فایل سرور بخواهد کافی است اسم فایل سرور را بگوید اینکه کجا هست نیاز ندارد بداند Middleware مکان دقیق فایل سرور را می‌داند. همینطور Migration و Relocation. یا می‌تواند Replication Transparency ایجاد کند. Middleware همه سرورها را

Update می‌کند. می‌شود خود سرودها مسئول Update باشد. Failure Transparency اگر یک سرور Fail کند تلاش می‌کند سرور دیگری پیدا کند.

- **Servers**

معمولاً همگی به یک صورت سازماندهی شده‌اند. روش کار این است که منتظر Request هستند جواب میدهند و این کار تکرار می‌شوند.

انواع سرورها:

1. Iterative Severs
2. Concurrent Servers

Iterative Severs: همان سروری است که تقاضاها را یکی یکی می‌گیرد.

Concurrent Sever: همانگونه که در مثال Multi Thread دیدیم می‌تواند Multi Request انجام دهد. هزینه این بیشتر است ولی کارایی بالاتر است.

کجا Client با مشتری تماس بگیرد؟ هر سرور یک Process است. Process از کجا شناخته می‌شود؟ از طریق یک End Point یا Port که یک شناسه است که وقتی Process ایجاد شده است سیستم عامل به آن می‌دهد. Client کجا تماس می‌گیرد از طریق پورت سرور. چگونه Client این End point را می‌شناسد؟ یک را این است که آن End Point یک Well known Service باشد مثلا برای FTP از پورت 21 استفاده می‌شود و یا HTTP پورت 80 است. اما خیلی وقتها Dynamic Assignment است که توسط OS محلی نسبت داده می‌شود. برای شناخت این سرور باید خودش را Register کند جایی که بشود از آنجا پرسید. اصلاحاً Processی هست بنام Demon که آنجا Register می‌کنند و یک Well known Address دارد. و او در پاسخ شماره پورت سرور را می‌دهد.

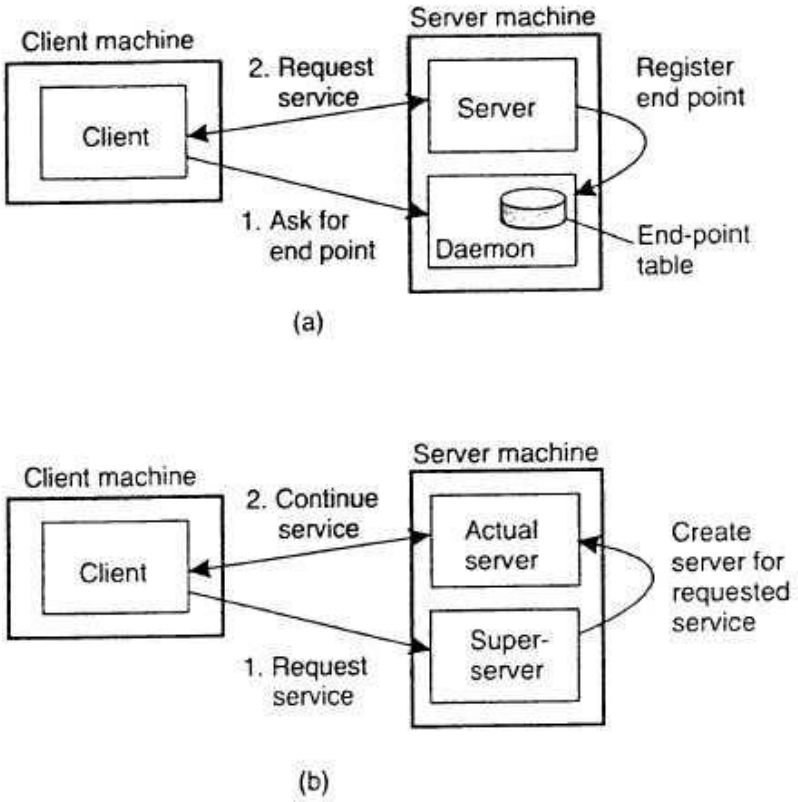


Figure 3-11. (a) Client-to-server binding using a daemon. (b) Client-to-server binding using a superserver.

شکل a این را نشان می‌دهد. ابتدا Server نام و End point خودش را در Demon رجیستر می‌کند. حالا Client می‌خواهد با سرور تماس بگیرد اگر بداند در این Server Machine است اما End point را نمی‌داند او ابتدا از Demon یک end point از سرور می‌خواهد و او بازمی‌گرداند. بعد مستقیم با سرور تماس برقرار می‌کند. این برای سرورهایی بود که دارای end point پویا بودند.

Super Server: کجا به کار می‌رود؟ اگر در سیستمی تعداد زیادی سرور باشد که منابع سرور را مصرف می‌کنند و Request کم بیاید ممکن است این waste of resources باشد. در این حالت بهتر است از Super Server استفاده کنیم شکل b بالا. یک سرور است که به جای چندین سرور است و به چندین end point گوش می‌کند اگر Clientی سروری را می‌خواهد با Super sever تماس می‌گیرد, Super sever آن سرور را ایجاد می‌کند یعنی یک Process جدید بعد به Client سرویس می‌دهد آخر هم که تمام شد دوباره این سرور از بین می‌رود و در صورت نیاز جدید مجدداً ایجاد می‌شود به این ترتیب از اتلاف منابع جلوگیری می‌شود. دقت شود ایجاد و حذف سرویس هزینه دارد و سربار اضافی است و در کارایی تاثیر می‌گذارد اگر کارایی می‌خواهیم باید هزینه منابع را بدهیم و بالعکس.

چگونه یک سرور را Interrupt کنیم؟ ممکن است Clientی از سروری درخواست عملیاتی کند بعد کاربر Client متوجه می‌شود آن عملیات را لازم نداشته. چگونه آن را قطع کند؟

روش اول: این است که کاربر Client بطور ناگهانی برنامه کاربردی را خاتمه دهد یعنی Client دیگر نیست. و بعد دوباره شروع کند. در اینصورت Server دیگر Client را نخواهد دید تلاش می‌کند Client را پیدا کند بعد که پیدا نکرد فرض می‌کند از بین رفته است و ارتباط را قطع می‌کند. ر

روش دوم: داده خارج از مسیر اصلی بفرستیم که دو نوع است:

1) یکی اینکه یک پورت اضافی داریم که برای کنترل است علاوه بر پورت داده.
2) دیگری اینکه روی همان پورت داده اضطراری بفرستیم. داده خیلی مهم که فرمت خاصی دارد وقتی رسید سرور می‌فهمد ارتباط را قطع کند.

آیا سرور قرار است Stateful باشد یا Stateless؟

Stateful یعنی سرور اطلاعاتی از Clientها در خودش ذخیر می‌کند نه هر اطلاعاتی بلکه اطلاعاتی که اگر گم شود ارائه سرویس میسر نباشد. اگر ذخیره نکند می‌شود Stateless. مثالها مثلاً تقاضاهای http. سروری که این تقاضاها را می‌گیرد Stateless است. هر تقاضا را مستقلاً جواب می-دهد.

آیا می‌توان Stateless باشیم اما هنوز اطلاعاتی را از Clientها ذخیره کرد؟ بله یک وب سرور می‌تواند اطلاعاتی از Clientها نگه دارد و همچنان stateless باشد. چه فایده‌ای دارد؟ می‌تواند رفتار او را پیش‌بینی کند تا خودش را تطبیق دهد و بهترین پاسخ را بدهد.

مثال State full: سروری موظف است Updateهایی را به Clientها بدهد. این سرور باید بداند که کدام Clientها از او سرویس می‌گیرد پس باید Table of Clients داشته باشد اگر Table گم شود دیگر نمی‌تواند سرور نمی‌تواند درست عمل کند.

مقایسه Stateful در مقابل Stateless: از لحاظ Performance, Stateful بهتر است اما اگر سرور Crash کند و Table از دست برود همه چی از دست می‌رود ولی Stateless در مقابل Crash مقاوم است وساده است.

- **Code Migration یا مهاجرت کد**
  - چرا Code Migration کنیم؟ دو حالت دارد Code Migration و Process Migration که بخشی از Process اجرا شده حالا بخش دیگر را می‌خواهیم ببریم جای دیگر. چرا Code Migration؟ اول Performance. می‌شود Process یا کدی را از ماشین دارای بار زیاد به ماشینی با بار کم منتقل کرد. بار ماشین را می‌توان از میانگین صف ماشین متوجه شد. شاید مسئله مهمتر از بار مسئله Communication باشد که باز این هم Performance می‌شود. پس شد **Performance Processing** و **Performance Communication**. ارتباط را به دو طریق می‌توان نگاه کرد بخشی از کد client را به سرور منتقل کرد حالت دوم برعکس بخشی از بار سرور را به Client منتقل می‌کنیم که هر دو برای کم کردن ارتباط است. مثلاً Client برنامه ای اجرا کرده که نیاز دارد Query های زیادی از سرور بگیرد می‌توان بخشی از برنامه کاربردی را به سرور منتقل کرد. برعکس می‌خواهیم Table را در سرور بسازیم که به ازای هر فیلد باید ارتباط انجام شود پس بهتر است ساخت جدول را در Client انجام داد. یا Java applet از سرور به Client منتقل می‌شود. دلیل دیگر می‌تواند **Flexibility** باشد. همان روش سنتی ساخت سیستم‌های موازی است که کد با به بخش‌هایی تقسیم می‌کنیم که دستمان باز است هر جور خواستیم می‌توان اینها را سازماندهی کرد. دلیل دیگر **Dynamic Configuration** است. برای اینکه دو یا چند موجودیت بتوانند با هم کار کنند باید تنظیم شوند در اینجا هم Client و Server باید با هم configure شود. مثلاً یک سرور برای اینکه یک

مشتری را قبول کند پروتکل خاصی دارد تنظیم یعنی کاری کنیم که این مشتری از این پروتکل استفاده کنیم. و کد از طرف سرور برای مشتری ارسال می‌شود.

پس فواید مهاجرت کد:

1) Performance Processing
2) Performance Communication
3) Flexibility
4) Dynamic Configuration

وقتی از مهاجرت کد صحبت می‌کنیم باید Security را در نظر بگیریم. به خصوص وقتی کد از Client به سرور می‌رود.

- **مدل‌های مهاجرت کد:**

مدلی وجود دارد که یک Process را به سه سگمنت تقسیم می‌کند. Code Segment, Resource Segment و Execution Segment. وقتی از مهاجرت صحبت می‌کنیم می‌تواند همه این سگمنت‌ها باشد که می‌شود کل Process و یا بخشی از سگمنت‌ها باشد. Process یک موجودیت در حال اجراست. Process وقتی در حال اجراست پس Resourceهایی را در اختیار دارد که می‌تواند اجرا شود. Execution Segment می‌گوید الان Process, State چیست.

پس Process از چه بخشهایی تشکیل شده است؟

1) Code Segment
2) Resource Segment
3) Execution Segment

مهاجرت تقسیم بندی میشود:

1) ضعیف یا Weak: فقط Code Segment منتقل میشود. مثال Java applet.
2) قوی یا Strong: هر سه تا سگمنت منتقل می‌شود. که دشوارترین حالت است.

تقسیم بندی دیگر برای مهاجرت کد:

1) Sender Initiated
2) Receiver Initiated

طرف Send یا طرف Receive مهاجرت کد را شروع کرده است. Send و Receive نسبت به Client سنجیده می‌شود. مثلاً Java applet, Receiver Initiated است. در حالتی که Sender Initiated هست Security مهمتر است.

جدول زیر این تقسیم بندی‌ها را نشان می‌دهد:

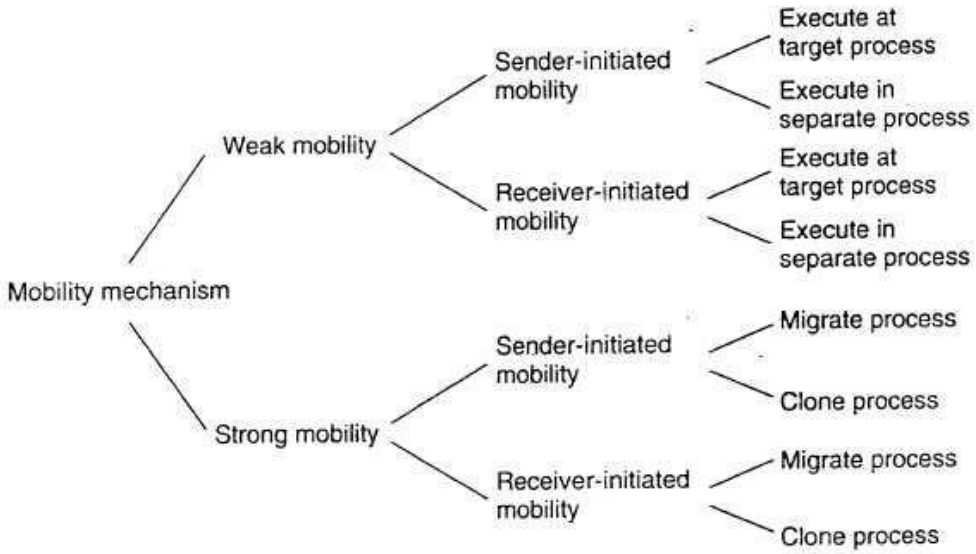


Figure 3-18. Alternatives for code migration.

Weak Mobility یعنی فقط کد سگمنت را انتقال دهیم. اگر برای اجرا یک Process جدید ایجاد کنیم می‌شود Execute at Separate process ولی اگر در Process سرور یا کلاینت منتقل کنیم یعنی اضافه کرده‌ایم به کد آنها می‌شود Execute at target process. Separate Process هزینه بیشتر دارد و از لحاظ Security بهتر است چون به خود Process ما نمی‌تواند دسترسی داشته باشد. Java applet بصورت Execute at target process است زیرا در browser اجرا می‌شود.

Migrate Process یعنی این Process را از این نقطه که وجود دارد به جای دیگر منتقل شود که می‌تواند Sender-Initiated یا Receiver-Initiated باشد. Sender-Initiated یعنی Process از جایی که دچار مشکل شده است اقدام به چنین کاری کند. یعنی ببیند محیط بار زیاد دارد مهاجرت کند. می‌شود همین بصورت Receiver-Initiated انجام شود یعنی ماشینی که بار کم دارد به همه اعلام کند و شما به من بار بدهید. Clone کردن Process یعنی fork Remote کردن. Fork کردن یک Processیعنی از توی یک Process یک Process دیگر را ایجاد کنیم این Remote fork است یعنی شما که یک Process هستی بیا یک Process در یک ماشین دیگر ایجاد کن. این می‌شود Cloning. این یک کپی از Process است. Clone کردن یعنی یک کپی از خودش بوجود بیاورد.

- **رابطه مهاجرت با Local Resource چیست؟**

یک مطلب رابطه Process با Resource چیست و یکی رابطه Resource با ماشین چیست؟

رابطه Process با Resource:

سه حالت تشخیص داده می‌شود:

1) Binding by identifier – مثال Communication end point دقیقاً باید خودش باشد.
2) Binding by Value – مثال Standard Library که Content را می‌خواهیم. هر کپی باشد قبول است. پس ضغیف‌تر می‌شود.

3) Local Devices – Binding by Type هستند از همه ضغیف‌تر هستند. مثلاً یک چاپگر می‌خواهم. حتی می‌تواند فرق هم بکند در بالایی کپی بود نمی‌توانست فرق کند.

اینها از بالا به پایین ضعیف می‌شود. اولی یعنی دقیقاً Identify کنیم Resource را. مثلاً communication end point یک Identifier است یعنی همن End point چیز دیگری جایگزین نمی‌شود.

حال رابطه Resource به ماشین:

سه حالت تشخیص داده می‌شود:

1) Unattached: مثلاً یک data base
2) Fastened: می‌شود از ماشین جدا کرد ولی دردسر دارد. مثلاً یک Web server.
3) Fix: نمی‌شود از ماشین جدا شود. مثلاً یک مانیتور را نمی‌شود ب Process منتقل کرد.

هر چه پایین می‌آییم ارتباط قوی تر می‌شود. اگر اینها را با هم ترکیب کنیم نه حالت مختلف بدست می‌آید.

Resource-to-machine binding

Process-to-resource binding	Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV,GR)	GR (or CP)	GR
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR	Establish a global systemwide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally-available resource

Figure 3-19. Actions to be taken with respect to the references to local resources when migrating code to another machine.

اگر by identifier باشد و unattached باشد می‌توان Process را Move کرد. حالتی که by identifier باشد Unattached هم باشد نمی‌توان انتقال داد این است که share باشد. یعنی بین چند Process مشترک باشد. راه حل دیگر Global Reference است. یعنی فرض کنید یک فایل shared بوده نمی‌توانستیم انتقال دهیم از Global Reference استفاده کنیم. مثلاً یک URL یک Global Reference است. اگر fastened باشد دردسر دارد از Global Reference استفاده می‌کنیم. اگر Fix باشد که نمی‌توان آنرا انتقال داد. اگر by value باشد و unattached باشد راه حل بهتر استفاده از یک کپی است. برای by type دیگر مهم نیست شبیه هم باشد کافی است از Rebind استفاده می‌شود مثلاً اگر یک مانیتور باشد Rebind می‌کنیم به یک مانیتور دیگر. چه حالتی است که در type می‌شود به جای Rebind از GB استفاده کنیم؟ حافظه مشترک. Global Reference همیشه هم راه حل خوبی نیست. مثال یک Multimedia workstation که تصاویر را پردازش می‌کند. خود workstation محاسبات را به یک سرور compute server می‌دهد حجم ارتباطات زیاد است و ارتباط مزیت سرور را از بین

می‌برد. آیا بهتر نبود به جای اینکه Global Server کنم خود مشتریها مستقیماً به سرور محاسباتی بدهیم؟ بحث این است که Process را Migrate می‌کنیم آیا می‌شود Resource با هم با آن Migrate کرد؟

- **مهاجرت در سیستم های ناهمگون**

مشکل مهاجرت در سیستم‌های ناهمگون است. همگون هم سیستم عامل یکی است هم پردازنده یکی است لذا کدی که روی این یکی انجام می‌شود می‌تواند روی دیگری هم انجام شود یعنی Platform یکی است. در سیستم های ناهمگون اگر weak mobility را در نظر بگیریم که فقط Code Segment مهاجرت می‌کرد یک Recompile انجام شود می‌تواند سازگار با OS جدید کد تولید کرد سازگار با Platform جدید به شرطی که کامپایلر موجود باشد.

در اینجا به Virtual machine هم اشاره شده است. که می‌توان از آن برای مهاجرت کد استفاده کرد. الان برنامه جاوا را می‌توان روی هر ماشینی اجرا کرد کافی است Java virtual machine داشت و برنامه را اجرا کرد.

فصل 4: ارتباطات

مطالب

- کلیات
  - پروتکل‌های لایه بندی
  - انواع ارتباطات
- RPC
  - عملیات اصلی در RPC
  - پاس کردن پارامتر
  - Asynchronous RPC
- Message Oriented Communication
  - Message Oriented Transient Communication
  - Message Oriented Persistence Communication
- Stream Oriented Communication
  - پشتیبانی Multimedia یا Mediaی پیوسته
  - Stream و کیفیت سرویس
  - همگام سازی Stream
- ارتباط چند پخشی یا Multicast
  - Application-level Multicasting
  - انتشار پیام بر اساس شایعه یا Gossip-Based Data Dissemination

Process ها چگونه با هم ارتباط داشته باشند.

- **کلیات**
  - **پروتکل‌های لایه‌ای**

در سیستم توزیعی به دلیل عدم وجود حافظه اشتراکی, تمامی ارتباطات بر اساس ارسال و دریافت پیام استوار است. یک مدل 7 لایه‌ای داریم.

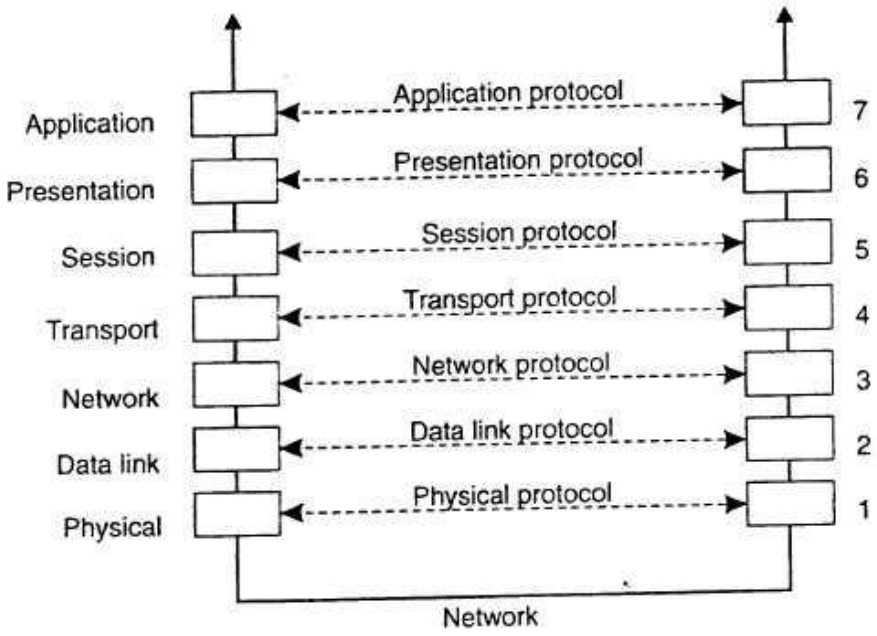


Figure 4-1. Layers, interfaces, and protocols in the OSI model.

اگر در Application بخواهند با هم در تماس باشند باید تمام این ساختار لایه‌ای در هر دو نود پیاده سازی شده باشد. ارتباط لایه‌های نظیر به نظیر را پروتکل می‌نامند. و ارتباط لایه بالا به پایین را Interface می‌نامند. همواره لایه پایین به لایه بالا سرویس می‌دهند, برعکس نیست. و از طریق Call است و Interface ساختار Call است. سرویس می‌تواند اتصال‌گرا یا بدون اتصال باشد در هر لایه‌ای. سرویس اتصال‌گرا سه مرحله دارد. این ساختار لایه‌ای سربار زیادی دارد. برای همین در سیستم‌های توزیعی پیشنهاداتی داده-اند. برای مثال در لایه Transport پروتکل غالب در اینجا TCP است که اتصال‌گرا است. اگر بخواهیم یک ارتباط Client/Server برقرار شود چند پیام باید رد و بدل شود. در ابتدا مشتری یک پیام Syn می‌فرستد بعد سرور Ack می‌دهد و Syn می‌دهد بعد مشتری Ack می‌دهد. بعد مشتری Request را می‌فرستد سپس پیام finish. سرور این پیامها را Ack می‌دهد. بعد Reply می‌دهد ... . نه پیام ردو بدل می‌شود برای ارسال که سربار زیادی است. این پروتکل TCP عادی است. یک پروتکل Transaction TCP پیشنهاد داده-اند TTCP. برای کاربردهای Transaction. Syn و Request و Finished را یکجا می‌فرستد. از آن طرف سرور Syn و Ack و Reply و Finish می‌فرستد و نهایتاً مشتری Ack می‌دهد. پس با سه پیام کار تمام شد.

در ساختار 5 لایه‌ای همه چیز رفته در لایه Application. مثلاً Application Specific Protocol مثل FTP یا HTTP یا General Purpose Protocol مانند پروتکل RPC یا RMI یا Streams که اصلاً Middleware همین است. پروتکل‌هایی برای Middleware

Services مانند Authentication Protocol یا Authorization Protocol مجوز دادن یا Distributed Commit Protocol. لایه Application تبدیل می‌شود به Middleware و Application.

- **انواع مدهای ارتباط:**

1- **PRC (Remote Procedure Call) یا RMI** که برای سیستم های Object oriented است.
2- **MOM (Message Oriented Middleware):** وقتی از پیام صحبت می شود قدری از توزیع شدگی دیده می شود.
3- **Data Streaming:** زمانی که ارسال اطلاعات محدودیت زمانی دارند. مثلاً صدا و تصویر اطلاعات باید با فواصل معین برسد. ولی در ارسال مثلاً ارسال اطلاعات فایل این محدودیت وجود ندارد.
4- **Multicasting:** یک Process با چند Process ارتباط برقرار می کند یعنی ارتباط بین یک گروه است.

- **RPC یا Remote Procedure Call**

یک Call عادی داخل خود Application است که تابعی را Call می‌کند. حالا می‌خواهیم سرویس را که هر سرور می‌دهد به شکل یک تابع ببینیم و آن را Call کنیم و از او Return بگیریم. برنامه کاربردی نباید متوجه شود این یک Call دور است بخاطر شفافیت.

دشواری های PRC

1- درخواست کننده و پاسخ دهنده در دو ماشین مجزا هستند
2- Remote Parameter Passing
3- ممکن است یک ماشین Crash کند و ماشین دیگر به کار خود ادامه می دهد. که کار را مشکل می‌کند.

انواع Parameter Passing

1) Call by reference
2) Call by value
3) Call by copy / restore مثل Call by value است اما هر تغییری داده شود دیده می‌شود. اثرش مانند Call by reference است.

در Remote, Call by reference اصلاً معنی ندارد. دو ماشین مختلف است آدرس اولی را به دومی می‌دهیم این کاملاً بی‌معنی است. راه حل چیست؟ فرض کنیم دو تا Process هستند که اولی می-خواهد دومی را Call کند. در حالت عادی فقط یک Process بود. و می‌خواهیم کاری کنیم که هیچ کدام از آن Process ها این Call دور را نفهمند و فکر کنند Call محلی است. روتین فراخوان تمام تابع را Packing می‌کند شامل اسم Procedure و پارامترها و تمام آرایه نمی‌تواند آدرس آرایه را بفرستد

که. بعد System Call, Send را انجام می‌دهد. در گیرنده پیام را دریافت می‌کند آنرا Unpack می‌کند و متوجه می‌شود یکی می‌خواهد روال Read را Call کند. خودش این روال را Call می‌کند. در اینجا Call by reference هم می‌تواند انجام دهد. بعد از Call, Return می‌شود بعد Pack می‌کند و ارسال می‌کند و فرستنده دریافت می‌کند حالا او Unpack می‌کند و یک Return درست می‌کند برای این Call. از نظر این Application هیچی عوض نشده است. آخر شفافیت است. این روتین‌ها اسم خاصی دارند آنی که در Client است را Client Stub می‌نامند و آن که در Server است را Server Stub است که در واقع همان Middleware ما هستند. کارشان این است که Pack کنند, Unpack کنند, Call میکنند, Return می‌کند. Client Stub و Server Stub در Application هیچ نقشی ندارند یعنی وابسته به Application نیستند. فقط باید Interface را بشناسد. دیدیم Call by reference برایمان مقدور نیست از Call by copy / restore استفاده کردیم.

10 مرحله Call کردن در RPC

1- Client Procedure فراخوانی می کند Client Stub را.
2- Client Stub یک پیام درست می کند.
3- Client Stub آن را به OS ماشین دیگر ارسال می کند.
4- OS سرور پیام را به Server Stub میدهد.
5- Server Stub پیام را باز کرده و Server را Call می کند.
6- Server کار را انجام داده و جواب را به Server Stub بر میگرداند.
7- Server Stub جواب را Pack کرده به OS خود می دهد.
8- OS سرور پیام را به OS کلاینت می دهد.
9- OS کلاینت پیام را به Client Stub میدهد.
10- CS پیام را باز کرده به Application می دهد.

مشکلات Parameter Passing

یک مثال تابع add(I,J) را در نظر می‌گیریم. در پیغام نام روال add بعد نام و نوع پارامترها ارسال می‌شود val(I) و val(J). و پیغام ارسال می‌شود و ارسال by value است. در این مثال call by value هم مشکل دارد. که نمادگذاری است Notation یعنی یک ماشین Little Endean است و دیگری Big Endean. راه حل قراردادی از قبل بگذاریم. حتی در اینجا مشکل دیگری نیز وجود دارد با معکوس کردن همه چیز درست نمی شود مثلاً اگر اسم یک تابع را ارسال می کنیم نباید این اسم تغییر کند.
راه حل: یک **Canonical Form** داشته باشیم که یک جدول است می گوید Integer از نوع دیگر Little Endean است Character از نوع Ascii می باشد Floating point با استاندارد IEEE 804 است. یعنی قرار می‌گذاریم اگر ارسال می‌کنیم با این قالب ارسال شود.
مشکل: فرض کنیم در جدول بالا دو ماشین که یک قالب دارند به فرمتی تبدیل می کنند که ندارند یعنی تبدیل اضافه. راه حل: Machine type را در پیغام بگذاریم. گیرنده چک می کند اگر مانند همان ماشین فرستند است تبدیل انجام نمی دهد و راندمان بالا می رود. اینکه این تفاوتها مشکلشان حل شود وظیفه Middleware است.

Call by reference: مشکل این است که آدرس این ماشین ربطی به آن ماشین ندارد. مثلاً زمان ارسال یک آرایه خود آرایه را باید ارسال کرد نه آدرس را. در واقع Call by reference نمی توانیم استفاده کنیم تبدیل کردیم به Call by copy/restore بعد تغییرات در آرایه می‌دهیم و آرایه را بازمی-گردانیم و مشکل حل است. اگر ساختار پیچیده بود مثلاً یک Complex Graph چکار کنیم؟ اصلاً نمی شود انتقال داد می شود ولی راه حل خوبی نیست. توصیه از ساختار پیچیده صرف نظر کنیم. راه حل: کلاینت واقعاً اشاره گر را ارسال کند نه خود Structure را. سرور دستگاری کند اشاره‌گر را برگرداند حالا کلاینت داده مرتبط را تغییر می دهد! از دید سرور برخی پارامترها Input هستند برخی Output. مثلاً اگر از یک File Server اطلاعاتی را می‌خواهیم بخوانیم در آرایه بریزیم. پس این آرایه Output است و لازم نیست زمان ارسال پیام از مشتری این آرایه را در پارامتر پاس کنیم پس پیام کوچک می‌شود. این که Optimization است برای بالا بردن کارایی. قرار شد این Interfaceها را با IDL تعریف کنیم و در آنجا می‌گوییم کدام Input است و کدام Output نسبت به سرور.

- **Extension هایی برای RPC وجود دارد. یکی Door یکی Asynchronous RPC**

مدلی از RPC ارئه شده است که هم Client و هم Server در همان ماشین است. چه چیزی ساده شده است؟ مهم این است که یک سیستم عامل داریم. پس RPC با ساده‌تر با کارایی بالاتر می-شود انجام داد. یک مثال مفهوم Doors است. این مفهوم را می‌توان در سیستم عامل قرار داد تا این کار با کارایی بهتر انجام شود. یک Client Process و یک Server Process در یک ماشین داریم. که اولی می‌خواهد از سرویس سرور استفاده کند. یک Main در سرور وجود دارد که ابتدا یک DoorCreate می‌دهیم در واقع می‌گوییم یک Door درست کن تا بقیه از این سرویس استفاده کنند که یک File Descriptor برمی‌گراند سیستم عامل این را Register می‌کند به عنوان سرویسی که دیگران می‌توانند از آن استفاده کنند. سمت Client هم از این استفاده می‌کند. سرعت این کار زیاد است زیرا از امکان سیستم عامل استفاده می‌شود.
اشکال این روش این است که شفافیت ندارد ولی کارایی دارد.

Extension دیگر **Asynchronous RPC** است که باز برای کارایی است. خود RPC بصورت Blocking است. یعنی وقتی شما Client هستی و Call میکنی Client Stub, Block می‌شود تا زمانی که پاسخ را از سمت سرور بگیرد که این ارتباط سنکرون هست. اما این انتظار کارایی را پایین می‌آورد. گاهی نیاز به پاسخ نداریم مثلاً Update a database مثلاً سرور می‌خواهد بگوید انجام شد. در این مواقع می‌شود منتظر نشد.

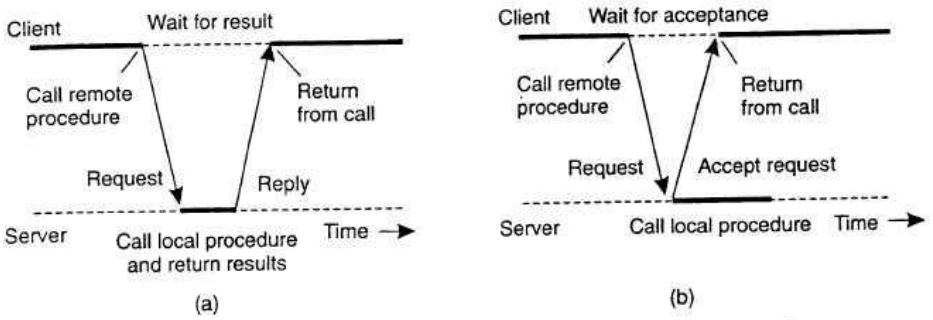


Figure 4-10. (a) The interaction between client and server in a traditional RPC. (b) The interaction using asynchronous RPC.

اگر خواستیم این را داشته باشیم در Middleware باید Asynchronous RPC را تعریف کنیم که داریم شفافیت را از دست می‌دهیم. در این حالت سرور Ack می‌دهد و بعد Client به کار خودش ادامه می‌دهد. حالت دیگر این است که Client اطلاعاتی را می‌خواهد ولی همین الان به آن احتیاج ندارد بعداً لازم دارد. پس Request را می‌دهد Ack را می‌گیرد و ادامه می‌دهد. زمانی که پاسخ سرور آماده شد Client را Call می‌کند که مفهوم Client و سرور به هم می‌ریزد و بعد هم Ack می‌گیرد.

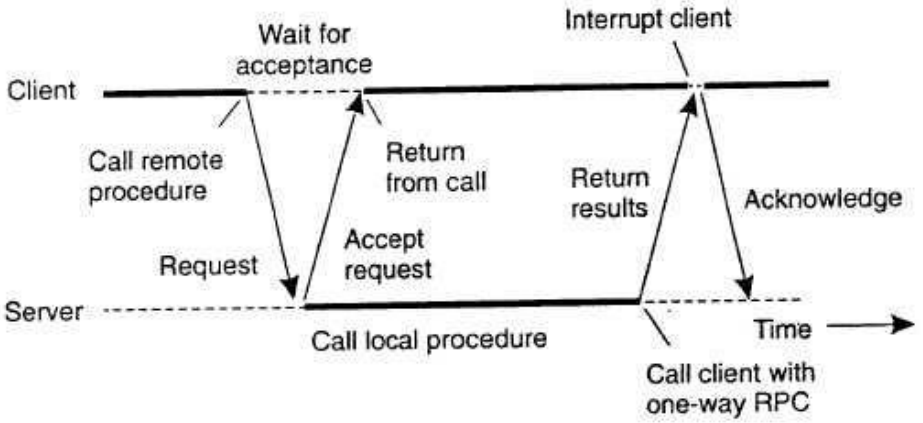


Figure 4-11. A client and server interacting through two asynchronous RPCs.

- **در هر RPC این کارها باید انجام شود:**
  1- ماشین سرور پیدا شود
  2- Binding انجام شود. یعنی ارتباط بین Client و سرور ارتباط برقرار شود.
  3- Data Type conversion انجام شود.

چگونه Client Stub و Server Stub ایجاد شود؟ Interface باید برای دو طرف مشخص باشد که از IDL استفاده می‌شود. همین Definition File (IDF) کافی است تا Stub تعریف شود. Stub چکار می‌کند؟ Call را به Message تبدیل می‌کند Pack می‌کند و برعکس. این فایل را به یک IDL Compiler میدهیم و بعد Stubها بدست می‌آیند.

ماشین سرور چگونه پیدا می‌شود؟ یا Binding a client to a server؟

اول باید ماشین سرور را پیدا کنیم بعد داخل آن سرور را پیدا کنیم.

1- در Daemon خودش سرویس را Register می‌کند و آدرس داخلی خودش را می‌دهد همان End point را . هر Process یک End point دارد. Daemon سرویسی است که همیشه کار می‌کند.
2- سرور سرویس خودش را در Directory ثبت می‌کند و آدرس ماشین را می‌دهد.
3- حالا Client اول look up می‌کند اگر Register شده باشد به او Return می‌شود که آدرس ماشین است.
4- Daemon یک Well known Address دارد. از Daemon آن ماشین End point آن سرور را می‌گیرد.
5- RPC انجام می‌دهد.

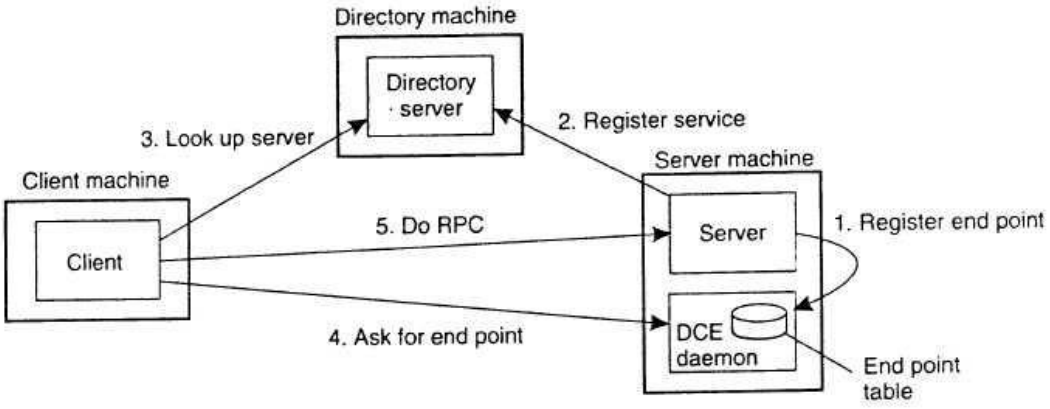


Figure 4-13. Client-to-server binding in DCE.

تمام این کارها را Middleware انجام می‌دهد. خود Application فقط یک Call انجام می‌دهد.

مواردی درباره RPC و RMI:

1- Receiving Side باید Executing باشد یعنی در حال اجرا باشد بدیهی است Sending Side هم در حال اجراست.
2- Client, Block خواهد شد.

آیا این دو لازم است یعنی دو طرف زنده باشند و یکی Block شود؟ مثالهایی وجود دارد که پاسخ منفی است. مثلاً کاربردهایی در محدوده جغرافیایی وسیع. مثال Email. یک Host داریم و یک mail server. اولاً نیازی نیست این دو Executing باشد و نیاز نیست یکی Block شود. ممکن است یکی ایمیل بفرستد و دیگری دو روز بعد بیاد چک کند که ممکن است در این لحظه فرستنده حتی زنده نباشد. این نوع را **Persistence** می‌گویند. در مقابل آن Transient است که این دو شرط را می‌خواهد.

- **Message Oriented Communication**

سیستم های که بر اساس حالتهای مختلف ارسال پیام باشد را Message Orient Communication می گویند در مقابل RPC و RMI است. که به زیبایی RPC و RMI نیست چون در آنجا همه چیز مخفی بود. در اینجا همه چیز بر پایه Message است.

Messaging System دو نوع است:

1- Persistence: نیاز نیست دو طرف زنده باشند.
2- Transient: دو طرف باید زنده باشند.

از دید دیگر Messaging System باز دو نوع است:

1- Blocking یا Synchronous
2- Non Blocking یا Asynchronous

RPC و RMI از نوع Synchronous و Transient هستند.

چهار ترکیب مختلف می‌توان داشت. شکل زیر Persistence و Asynchronous است.

شکل زیر Persistence و Synchronous است: یعنی دو ماشین همزمان فعای نیستند و پیغام را ارسال می‌کند چون سنکرون است سیستم عامل ماشین سرور به او Acknowledge می‌دهد. سرور Ack نمی‌دهد بلکه سیستم عامل می‌دهد. بعد از مدتی سرور پیداش میشه. آن موقع پیام را دریافت می‌کند.

شکل زیر Transient و سنکرون است: هر دو وجود دارند اما آسنکرون است. زیرا بعد از اینکه Client پیام را فرستاد کارش را ادامه داد.

شکلهای زیر سنکرون و Transient است سومی همان چیزی که RPC و RMI است تفاوت در این است که نقطه سنکرون فرق دارد.

دو مثال می‌زنیم یکی در باره Transient Communication System و دیگری درباره Permanent Communication System.

Message Oriented Transient Communication System

مثال سوکت‌هاب برکلی: در Transport Layer پیاده سازی شده است. این لایه یک ارتباط انتها به انتها برقرار می‌کند. از طریق System Call باید این کار را کرد که به آن Primitive گفته می‌شود. Primitiveهای لازم عبارتنداز:

1- Socket: سیستم عامل منابع لازم را برای این ارتباط تامین می‌کند مثلاً بافر.
2- Bind: به یک end point متصل شویم. Bind یعنی آدرس دادن به end point.
3- Listen: به همه اعلام می‌کند من در حالت Listen هستم.
4- Accept: اگرچیزی ارسال شد می‌تواند قبول کند.
5- Connect
6- Send
7- Receive

8- Close: ارتباط قطع می‌شود.

مثال: یک سرور دارم و یک Client. سرور ابتدا Socket درست می‌کند یعنی System Call, Socket را فراخوانی می‌کند و می‌گوید می‌خواهم بادیگران ارتباط برقرار کنم بعد bind می‌کند یعنی به end point خودش یک آدرس می‌دهد. بعد به همه اعلام می‌کند که من Listen می‌کند بعد می‌رود در حالت Accept. یعنی آمادگی دارد تقاضاهای دیگران را بپذیرد. چون سرور Passive است. Client چکار می‌کند؟ Socket ایجاد می‌کند. Connect می‌کند. ارتباط انتها به انتها برقرار می‌شود. حالا تبادل داده می‌کنند و در نهایت ارتباط را قطع می‌کنند. این ارتباط اتصال‌گرای لایه چهار است.

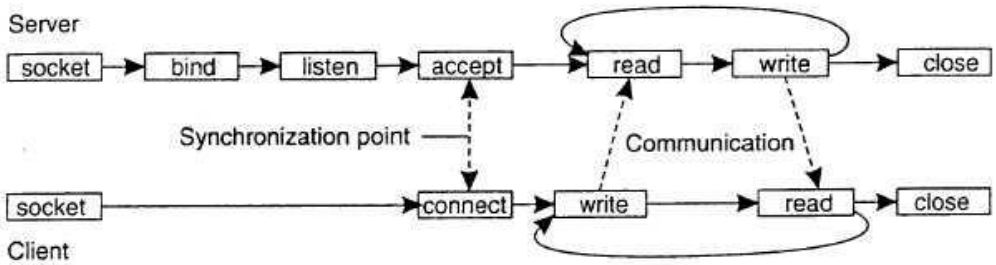


Figure 4-15. Connection-oriented communication pattern using sockets.

چرا Client, bind نکرد؟ چون هر آدرسی بگیرد اصلاً اهمتی ندارد.

- **MPI (Message Passing Interface)**

برای رفع مشکل Socket یک Middleware ایجاد شد بنام MPI. پس دیگر به لایه Transport وابسته نیست.

فصل 5: نام گذاری

مطالب

- نام‌ها, شناسه‌ها و آدرس‌ها
- نامگذاری تخت

  چند را ه حل ساده

  Home-Based Approaches

  DHT (Distributed Hash Table)

  روش سلسله مراتبی
- نامگذاری ساخت‌یافته

  فضای نام یا Name Space

  Name Resolution

  پیاده سازی فضای نام

ارتباطات را گفتیم حالا چگونه Process ها همدیگر را بشناسند و پیدا کنند تا با هم ارتباط برقرار کنند. در Naming سه کار مهم می‌کنیم Resolve, Insert و Delete.

تعریف نام: رشته ای از بیت ها یا کارکتر ها که به یک موجودیت Entity (ماشین, دیسک, چاپگر, فرایندها, آدرس های وب و ...) خاص اشاره می کند.

نام می‌تواند Resolve شود یعنی پیداش کنیم. برای اینکه Resove کنیم باید یک Name System داشته باشیم.

شناسه: نامی است که سه ویژگی دارد:
1- حداکثر به یک موجودیت اشاره می کند.
2- شناسه رابطه یک به یک دارد با موجودیت
3- همیشه به یک موجودیت اشاره می کند.

Access point یا نقطه دسترسی: آدرس یک موجودیت را نشان می دهد. **نقطه دسترسی را همان آدرس می گویند**. مثلاً اگر تلفن نقطه دسترسی به یک فرد باشد شماره تلفن همان آدرس او خواهد بود. ممکن است یک موجودیت چند آدرس داشته باشد مانند یک نفر که چندشماره تلفن دارد. مثلاً وب سایت شرکتهای بزرگ روی چند کامپیوتر توزیع می شود پس باید نامی انتخاب کرد که مستقل از کامپیوترهای اجرا کننده سرویس وب باشد. اگر موجودیت متحرک باشد آدرس یا نقطه دسترسی ممکن است تغییر کند. ضمناً مثلاً سرویسی که امروز روی این ماشین اجرا می شود ممکن است فردا روی ماشین دیگری اجرا شود اگر ارجاع با آدرس باشد همه چیز به هم می ریزد. پس بهتر است نامهایی انتخاب شود که مستقل از مکان (Location Independent) باشد. شناسه نوعی نام است که سه ویژگی دارد: شناسه فقط به یک موجودیت اشاره می کند, هر موجودیت فقط یک شناسه دارد و هر شناسه همیشه به همان موجودیت اشاره می کند. با این حساب نمی توان از شماره تلفن به عنوان آدرس استفاده کرد چون بین افراد دست به دست می شود.

راه حل ساده Name Resolution

راه حل ساده این است که یک جدول نام, آدرس داشته باشیم. واضح است این راه حل Scalabe نیست. برای همین از اسامی ترکیبی استفاده می‌کنیم که از چند بخش تشکیل شده است. مثل ftp.ac.vu.nl که اول می‌رویم سراغ (.)NS - Name Server و nl را می‌گیریم بعد از (nl)NS و ... .

انواع نامگذاری:

1- نام گذاری تخت
2- نام گذاری ساخت یافته
3- نام گذاری صفت محور

نامگذاری تخت (Broadcasting, Multicasting, DHT)

خود نام ساختاری ندارد که مارا به سمت Resolution هدایت کند. ترکیبی نیست فقط یک بخش است که یک String است. خودش هیچ اطلاعاتی ندارد که بگوید چگونه Resolve کنیم. مشکل Scalability دارد.

- با استفاده از شناسه ها (Identifiers) می توان موجودیت ها را بطور منحصر به فرد شناسایی کرد.
- دو راه حل شناسایی موجودیت‌ها (Name Resolve) در نامگذاری تخت یکی همه پخشی و چند پخشی و دیگری (Hash Table) هستند.
- **Broadcast و Multicast**: پروتکل ساده است پیغام حاوی شناسه را در یک شبکه منتشر می کنیم هر ماشین که آدرس نقطه دسترسی آن موجودیت را در خود داشت پاسخ می دهد. چون Scalable نیستند در محیطهای کوچک مثل LAN کار می‌کنند. طرز کار این است که Identifire را به همه اطلاع می‌دهیم و آدرس Entity را می‌گیریم. همه Identifire را می‌گیرند و کسی جواب می‌دهد که آدرس را دارد.
- مثال همه پخشی پروتکل ARP (Address Resolution Protocol) در شبکه اینترنت در لایه Data Link است. که آدرس IP را داریم دنبال نام ماشین می گردیم.
- مشکلات Broadcasting
  1- پهنای باند اشغال می‌کند
  2- بی‌جهت همه را Interrupt میدهد. زیرا کامپیوتر ها باید کار خود را رها کنند و پیامهایی را پردازش کنند که مربوط به آنها نمی باشد.
  3- اگر به جای Broadcasting از چند پخشی استفاده شود اوضاع کمی بهتر می شود.
- از چند پخشی برای یافتن **نزدیکترین** موجودیت تکثیری (Replicate) نیز می توان استفاده کرد.
- **اشاره گرهای هدایت کننده یا Forwarding Pointers:** زمانی که موجودیتی متحرک است از این روش استفاده می شود. وقتی موجودیتی از فضای آدرس A به فضای آدری B می رود آدرس جدید خود را در A به جا می گذارد. بنابر این برای یافتن A باید از زنجیره ای از اشاره گرهای هدایت کننده استفاده کنیم.
- **نقاط ضعف اشاره گرهای هدایت کننده:**
  1- ممکن است زنجیره اشاره گرهای هدایت کننده بسیار بزرگ شود که دنبال کردن آن مشکل شود.
  2- تا کی این اشاره‌گرها را نگه دارند؟
  3- ممکن است زنجیره پاره شود.
- در Pointers Forwarding آدرس مبدا حمل می شود و زمانی که به هدف رسیدیم نیازی به برگشت از همان مسیر نسیت و مستقیم به مبداء باز می گردیم و آدرس مقصد در مبدا قرار داده می شود تا در آینده نیاز به طی کردن کل زنجیره نباشد. ایراد این روش این است که اگر از همان مسیر رفت باز می گشتیم گره های میانی نیز می توانستند آدرس مقصد را اصلاح کنند.

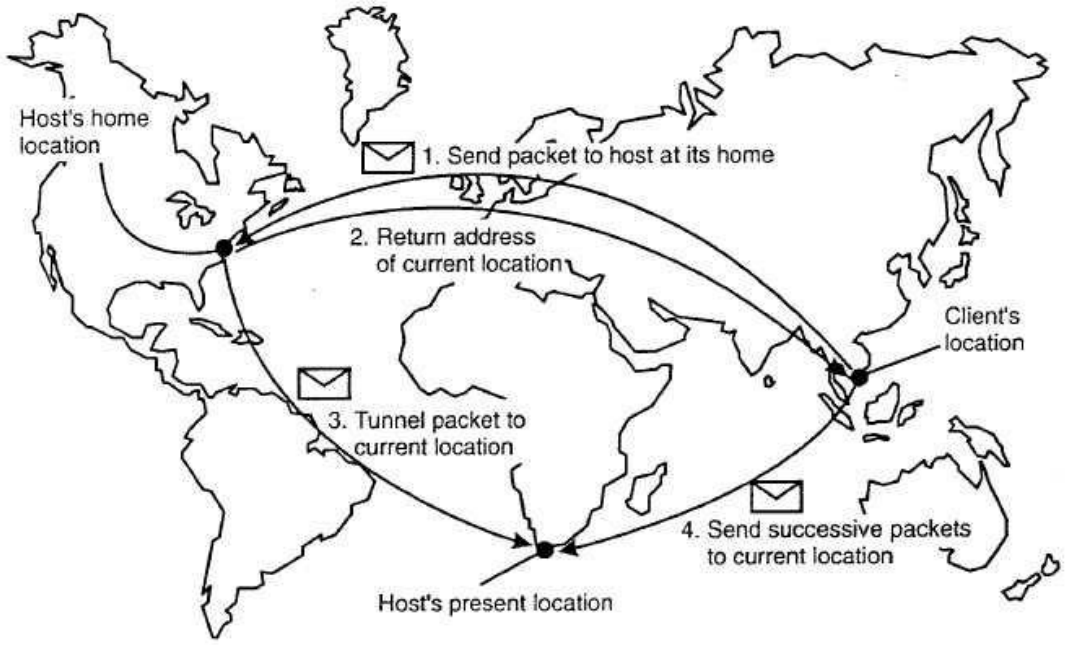
- مسئله دیگر این است که Forwarderهایی داریم که هیچ موقع استفاده نمی‌شوند این شامل Gabage Collector می‌شود.

- **سیستم‌های خانه – محور یا Home-Based Approaches:** روش های اشاره گرهای هدایت کننده و همه پخشی مشکل گسترش پذیری دارند. یک روش پشتیبانی از موجودیت های سیار مفهوم home location است. مفهوم خانه همان محلی است که موجودیت در آنجا ایجاد شده است. یکی از جاهایی که از این مفهوم استفاده می شود IP سیار است. تمام درخواست ها برای ارتباط با موجودیت سیار ابتدا به Home agent می رود. زمانی که یک نود به شبکه دیگری می رود یک آدرس در Home agent از خود باقی می گذارد. همزمان وقتی Home agent بسته را به شبکه میزبان فعلی می رساند آدرس جدید نود را نیز به فرستنده اطلاع می‌دهد و از این به بعد فرستنده مستقیم با نود تماس می گیرد. این آخرین آدرس نگه داشتن در Home Agent را **Care off** می گویند. این روش Scalable هست.

شکل زیر این مراحل را نشان می‌دهد:

1- Client بسته‌ای را برای Host در Home ارسال می‌کند.
2- Home پاسخ او را می‌دهد و آدرس جدید را به او اعلام می‌کند.
3- تقاضای Client را هم به اطلاع Host در محل جدید می‌دهد. به هر شکل Request از بین نمی‌رود.
4- Client و Host از این به بعد مستقیم با هم در تماس خواهند بود.

- مشکلات: مشکل وقتی است که نود برای مدت طولانی به شبکه دیگری منتقل می شود. و آنجا پایدار شود و دائمی شود.
-
- راه حل: Home را منتقل می‌کنیم پس برای Homeها می‌توان یک Directory داشته باشیم یک جای دیگر که Homeها خودشان را اینجا Register می‌کنند. هر چه با Home تماس می‌گیریم نمی‌شود می‌رویم سراغ Home Directory و مکان جدید را پیدا می‌کنیم. این بار زیادی هم برای Client ندارد زیرا از Cache استفاده می‌کند.



- **DHT (Distributed Hash Table) یا جدول های درهم توزیعی:** راه حل دیگر Flat Naming این است. برای توضیح طرز کار به عنوان نمونه از سیستم Chord استفاده می کنیم. یک Finger

Table برای هر نود داریم که هر یک دارای یک جدول مثلاً 5 سطری و دو ستونی است. ستون دوم Successor به توان 2 هاست. یعنی برای هر نود P داریم ($FT_p[i] = Succ(p + 2^{i-1})$ اگر successor ها مانند قبل ترتیبی بودند می شد $FT_p[i] = Succ(P + i)$. در اینجا می شود $FT_p[i]$ = Succ(P + 1), Succ(P + 2) , Succ(P + 4), Succ(P + 8), Succ(P + 16)

مثال: از نود 28 خواسته‌اند Resolve کند K = 12 را. به جدول خودش نگاه می‌کند می‌بیند 12 بین 4 و 14 است می‌رود سراغ گره 4. 4 در جدول خودش نگاه می‌کند می‌بیند 12 بین 9 و 14 است می‌رود سراغ گره 9. 9 می‌رود سراغ 11. و نهایتاً یازده در می‌کند به نود بعدی یعنی 14 زیرا 12 از اولین نود آن کوچکتر است. و Resolve انجام شد. تعداد پرش‌ها 4 تا است اگر در یک سیستم بزرگ می‌خواست یکی یکی برود زیاد میشد.

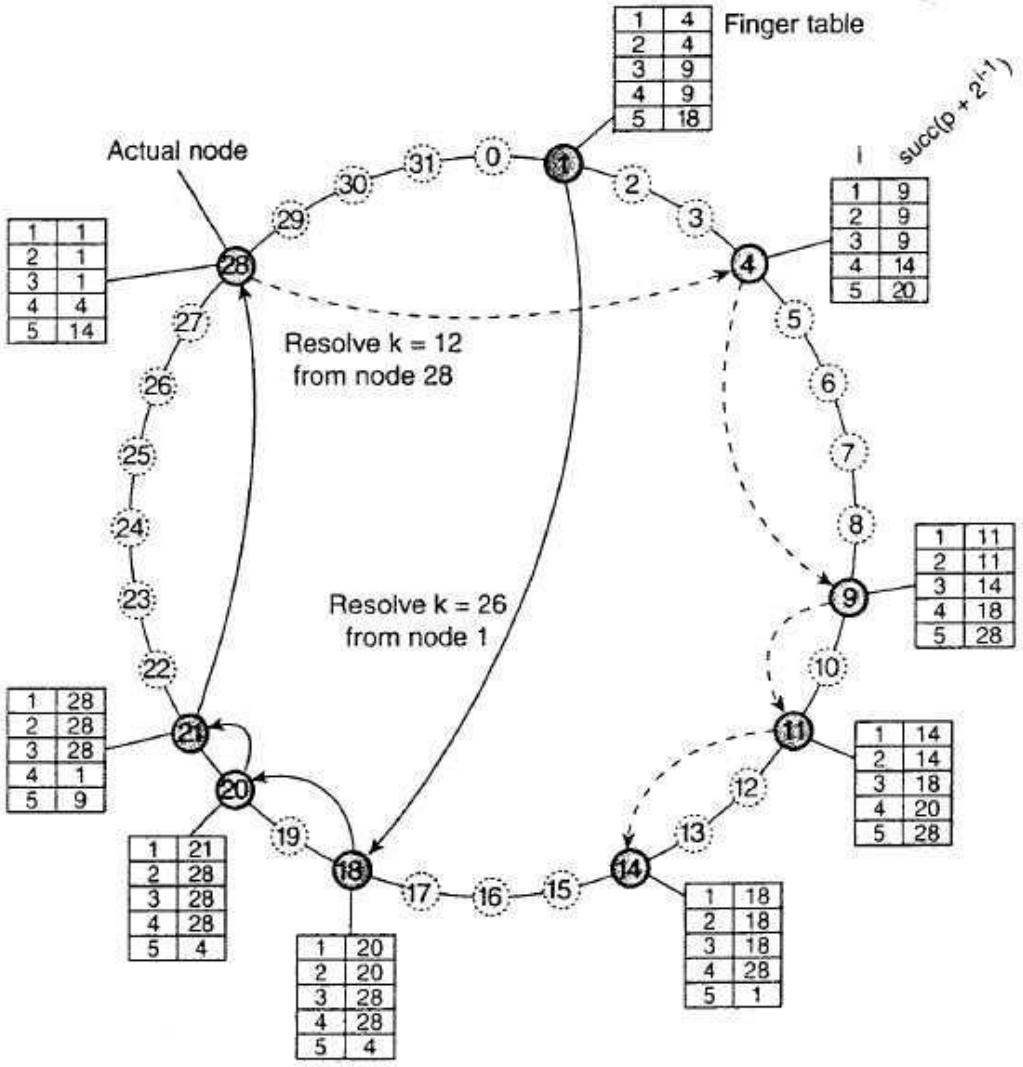


Figure 5-4. Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.

- **Join در سیستم فوق:**
  1. یک Random Id به نود مربوطه اختصاص می‌دهیم.
  2. Succ(id) را پیدا می‌کنیم.
  3. به Successor(id) می‌گوید من را بعنوان Predecessor خودت اضافه کن.
  4. Entityهای خودش را از Successor(id) می‌گیرد.

- **Leave:** Leave می‌کند بعد از مدتی می‌بینند Node مربوطه نیست اصلاح می‌کنند.

در Join و Leave جدول FT همیشه باید درست شود یعنی این جداول همیشه باید به روز باشند.

یکی از مشکلات سیستم فوق این است که ممکن است درخواست مسیرهای عجیب و غریبی کنند مثلاً از آمستردام بروند کالیفرنیا دوباره برگردند. مشکل از اینجا ناشی می‌شود که دید منطقی بادید فیزیکی تفاوت دارد. راه حل این مشکلات در زیر آمده است.

- **راه حل مشکلات DHT (سه راه حل):**

1- تخصیص شناسه گره بر اساس توپولوژی یا topology based assignment of nodes:
در این روش Successorی را انتخاب می‌کنیم که از نظر فیزیکی نزدیکتر است (Round Trip Delay).
اشکال این است که اگر یک شبکه از کار بیفتد در Ring ما یک گپ بزرگ کنار هم از دست می‌رود در حالیکه اگر کاملاً Random بود اگر یک Network کامل از کار می‌افتاد نودهایی که از کار می‌افتند پراکنده هستند در نتیجه مشکلی بوجود نمی‌آمد.

2- مسیر یابی بر اساس مجاورت یا Proximity Routing
این روش بهتر است. می‌توان انتخاب داشت. هر کس به جای یک Successor چند Successor داشته باشد. حالا می‌خواهیم Resolve کنم کدام را انتخاب کنم؟ آنی را که از نظر فیزیکی از همه نزدیکتر است. دیگر مجبور نیستیم Sub Space ایجاد کنیم. جدول چه می‌شود. جدول هم چندستونی میشود.

- **Hierarchal Flat Naming:** ساختار درختی شامل دامنه ها و زیر دامنه ها و درآخرین سطح leaf خود یک زیر دامنه است و در بالاترین سطح root است. برگ ها معمولا یک شبکه محلی هستند. هر دامنه یک گره منتاظر Directory دارد که اطلاعات موجودیت های آن دامنه را نگه می دارد. در واقع درختی از گره های دایرکتوری داریم. گره دارکتوری ریشه اطلاعات تمام موجودیت ها را دارد. موجودیت های تکثیر شده می توانند چندین آدرس داشته باشند. اگر موجودیتی دارای دو آدرس در دامنه های D1 و D2 باشند آنگاه کوچکترین نودی که بالای هر دوی اینهاست دارای دو اشاره گر به D1 و D2 است. در خواست آدرس به یک برگ می رود لازم نیست حتما از ریشه شروع شود اگر نداشت به گره بالایی میدهد. و به همین ترتیب نودی که آدرس را داشت به سمت پایین هدایت می شود. اگر دو آدرس داشت مثلاً به سمت نزدیکتر می رود. برگ جدولی دارد که شامل ستونهای Entity و آدرس است.

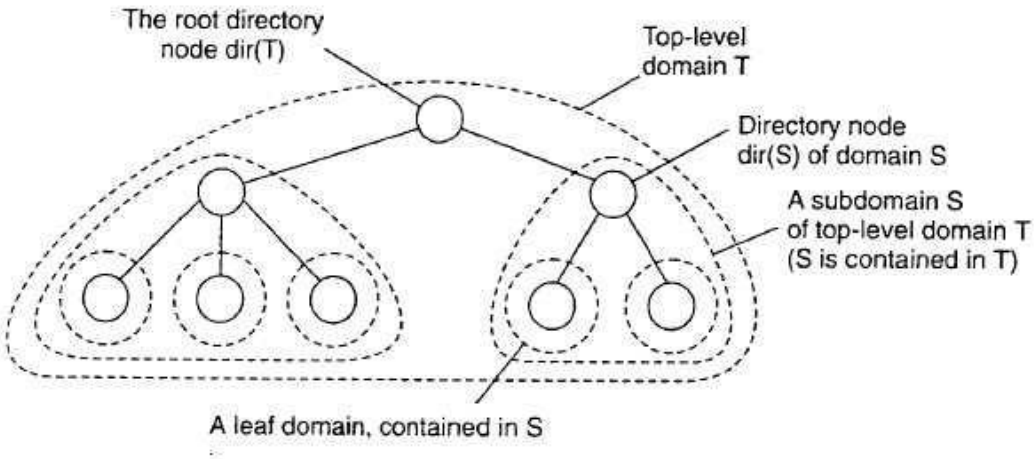


Figure 5-5. Hierarchical organization of a location service into domains, each having an associated directory node.

1- Resolve: تقاضا به برگ می‌رود. ندارد می‌رود سراغ Parent. دوباره ندارد در جدول خودش می‌رود بالاتر. گره M در جدول خود آن موجودیت را دارد می‌رود پایین تا به برگ برسد.

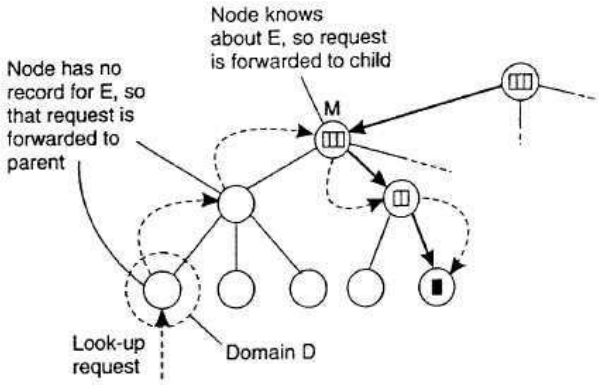


Figure 5-7. Looking up a location in a hierarchically organized location service.

2- درج: کافی است اطلاعات شناسه موجودیت و آدرس را در دایرکتوری برگ وارد می کنیم و آن اصلاح می شود و به همین ترتیب این اصلاح در تمام Parent ها صورت می گیرد. اگر این درج کپی بود لازم نیست تا ریشه برود تا جایی که که گره هر دو را ببیند می رود و به آن نود می گوید که جدول خود را اصلاح کند و بفهمد دو اشاره گر دارد و برگردد پایین به نود قبلی بگوید یک کپی داری.

3- حذف: وقتی یک Entity را از برگ حذف می کنیم تمام جداول Directory Entity تا Root باید اصلاح شود. مگر اینکه به نودی برسیم که دو آدرس از نسخه های کپی موجودیت دارد که آن را اصلاح می کنیم و دیگر لازم نیست Parent های این نود را اصلاح کنیم.

4- کپی: هرگاه در یک برگ یک موجودیت را تکرار کنیم آدرس Parent ها باید اصلاح شوند تا به یک Parent برسیم که به هر دو نود اشاره می کند در Directory آن نود باید دو اشاره گر ایجاد کرد. گاهی هر دو را لازم دارم مثلاً Update داریم باید هر دو اصلاح شوند.

- **مقایسه این سه روش** (Broadcast و Multicast, DHT و سلسله مراتبی)**:** از نظر Response بودن روش Broadcasting از همه سریعتر است ولی مشکل توسعه پذیری دارد. از نظر توسعه پذیری سلسله مراتبی بهتر است.

- **تفاوت بین جستجوهای تکراری (Iterative) و جستجوهای بازگشتی (Recursive):** در تکراری گرهی که درخواست تجزیه کلید از او شده است, آدرس شبکه گره بعدی را به فرایند درخواست کننده می دهد ولی در دومی این گره خود قدم بعدی را بر می دارد.

نام گذاری ساخت یافته مثال File System یا DNS

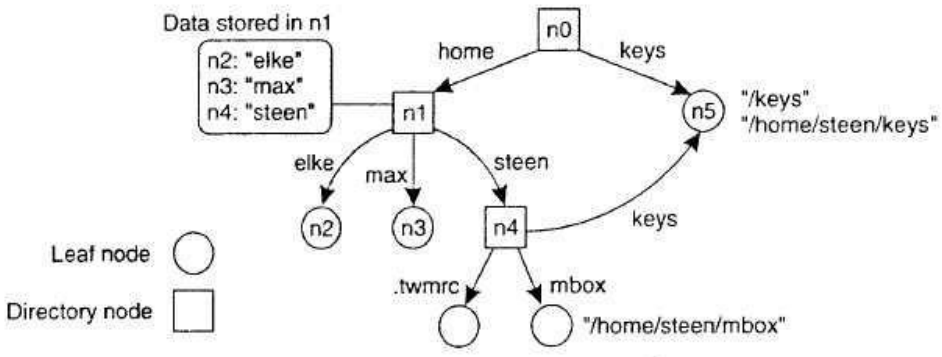


Figure 5-9. A general naming graph with a single root node.

- نکته این است که خود نام شامل ساختار است و ما را هدایت میکند. اینجا هم مانند Hierarchal Flat Naming یک ساختار سلسله مراتبی داریم. مثال فوق ساختار فایل سیستم است که به

صورت یک گراف نشان داده می شود. موجودیت ها در این ساختار همان فایل‌ها هستند و گره-های میانی Directory Node هستند که هر یک دارای جدولی هستند که شناسه گره و برچسب انشعاب را نشان می دهد که از طریق این جدول می توان به گره مورد نظر رسید. در این ساختار هر نود تنها اطلاعات بچه‌های خود را دارد و نیاز نیست اطلاعات بیشتر داشته باشد برعکس Hierarchal Flat Naming.

- این هم یک ساختار سلسله مراتبی دارد ولی خود نام این ساختار را نشان می‌دهد. موضوع Naming در فایل سیستم ها نیز مطرح است در اینجا موجودیت‌ها همان فایل‌ها هستند و باید آدرس فایل‌ها را پیدا کرد. مثال نام گذاری ساخت یافته را با استفاده از فایل سیستم توضیح داده است. این ساختار Tree است یا گراف. در این ساختار محل شروع مشخص است و میدانیم چگونه ادامه دهیم.
- در اینجا یک فضای نام داریم که با یک گراف نشان می دهیم که دارای دو نوع گره است گره برگ که مثلاً با دایره نشان می‌دهیم و گره دایرکتوری که با مربع نشان می دهیم. که دارای شاخه-های خروجی است که با یک نام منحصر بفرد مشخص می شود. اسامی می توانند Absolute باشند یا Local.
- **Resolve:** مثلاً /home/steen/keys را می‌خواهیم Resolve کنیم. از Root شروع می‌کنیم می-رویم سراغ home بعد steen بعد keys. کاملاً مشخص است از کجا شروع کنیم و چگونه ادامه دهیم. Naming System باید بسته باشد Close در غیر اینصورت ابهام دارد.

- **Merging Name Spaces**: یعنی آیا می‌توان یک File System که در یک سیستم است با با یک File System در سیستم دیگر Merge کرد؟ Name Space اینها فرق دارد. آیا می‌شود این دو را ادغام کرد؟ یک روش Mounting است.

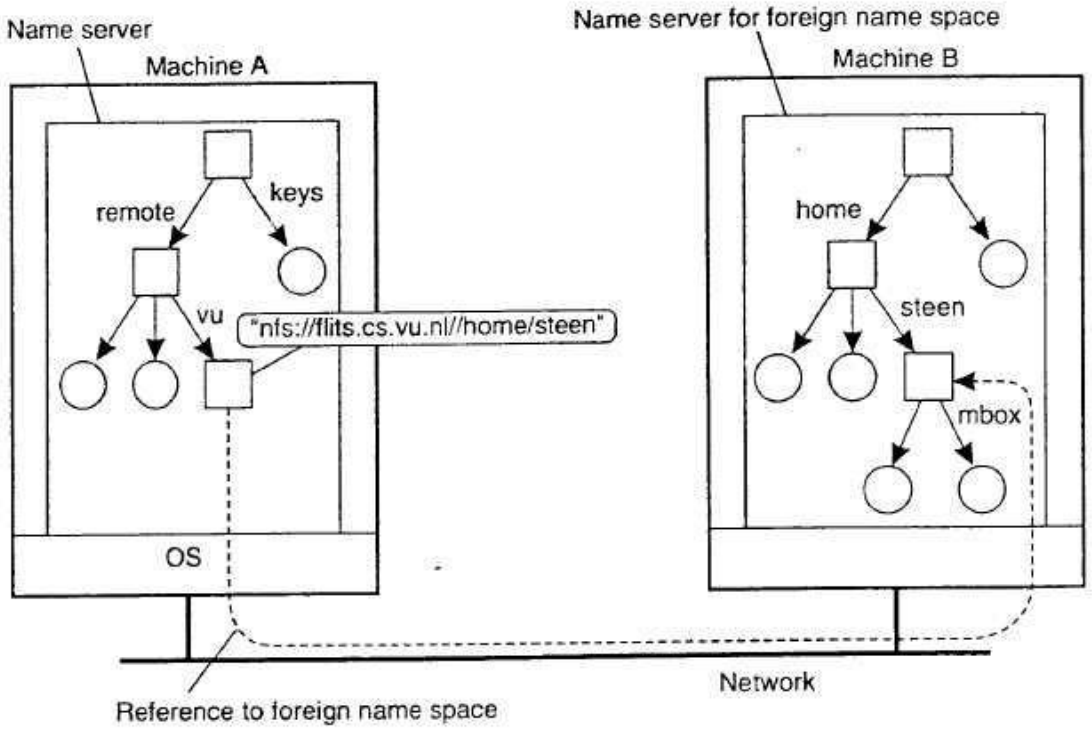


Figure 5-12. Mounting remote name spaces through a specific access protocol.

می‌خواهیم یک بخش از ماشین را به ماشین دیگر منتقل کنیم. اگر در vu لیست بگیریم mbox و دیگری را ببینیم. برای اینکار سه نام را باید Resolve کنیم:

1. اسم پروتکل دسترسی
2. نام سرور
3. نام Mounting Point

یعنی هم ماشین را پیدا کنیم هم تو ماشین آن نقطه را پیدا کنیم و هم ارتباط اینها بدانیم چگونه است. چیزی که در گره میزبان قرار میدهیم یک URL است بصورت: nfs://flits.cs.vu.nl//home/steen. این می‌گوید پروتکل ارتباطی nfs است (Network file system) به چه Resolve می‌شود؟ به سرور flits.cs.vu.nl و داخل ماشین این نقطه /home/steen.

- مثال دیگر DNS است. Distribution بصورت سلسله مراتبی و درختی است Flat نیست. سه لایه دارد:
  1. جهانشمول
  2. Administrational Layer
  3. Manager Layer. هر نود اطلاعات بچه‌های خودش را دارد.

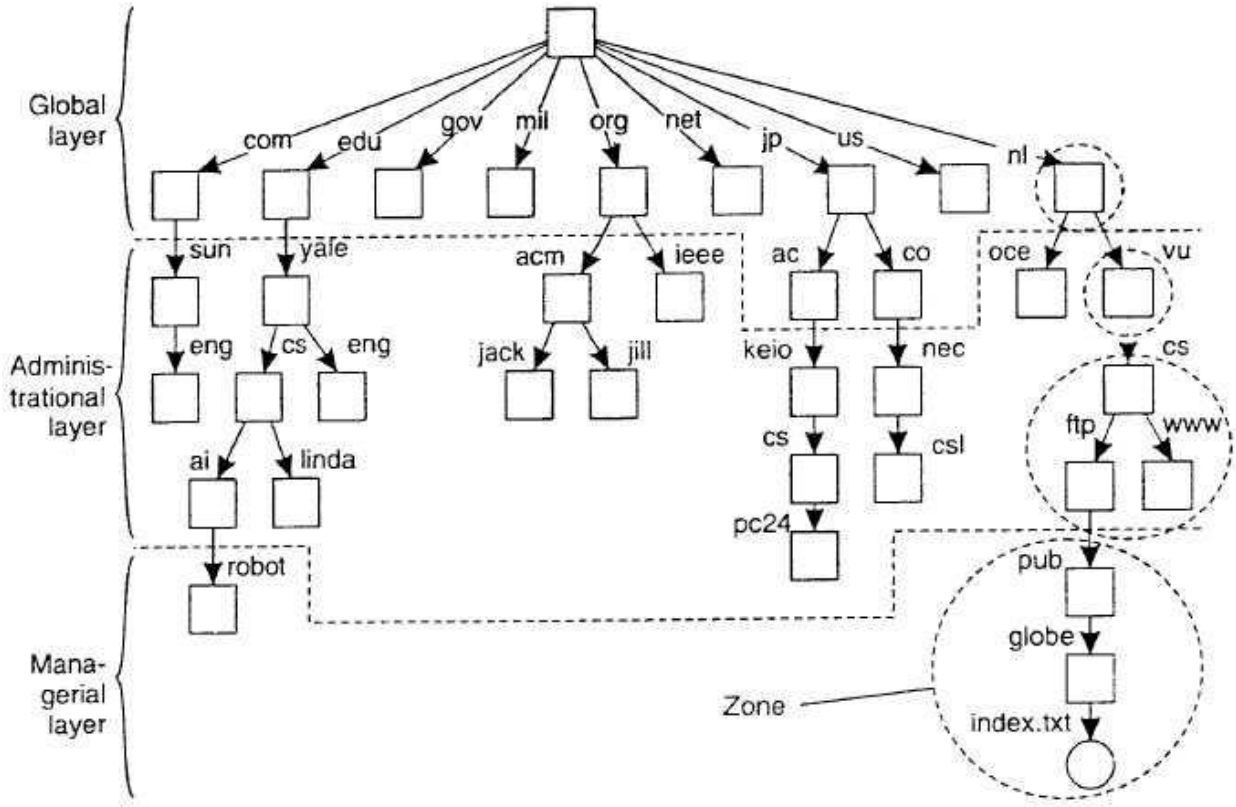


Figure 5-13. An example partitioning of the DNS name space, including Internet-accessible files, into three layers.

از لحاظ Stability کدام لایه اطلاعاتش عوض نمی‌شود؟ Global Layer. از لحاظ Updating لایه Managerial Layer باید فورا انجام شود.
جدول زیر مقایسه لایه‌ها رانشان می‌دهد:

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

Figure 5-14. A comparison between name servers for implementing nodes from a large-scale name space partitioned into a global layer, an administrational layer, and a managerial layer.

- در این سیستم‌ها Name Resolution می تواند دو جور باشد: Iterative و Recursive

1. Iterative: می‌رویم سراغ Root کل نام را می‌دهیم او nl را Resolve می‌کند. فقط این بخش را می‌دهد بفیه را نمی‌داند. می‌رویم سراغ بعدی. vu.cs.ftp را می‌دهیم و vu را Resolve می‌کند تا آخر ftp , Resolve می‌شود.

2. Recursive: فقط nl, vu, cs, ftp را به Root میدهیم و خودش nl را پیدا می‌کند و دنبال vu, ca, ftp Resolve می‌گردد. حالا این یکی vu را Resolve می‌کند و به همین ترتیب ادامه میدهد و در نهایت از همین مسیر برگشت می‌شود و کار پایان می‌یابد.

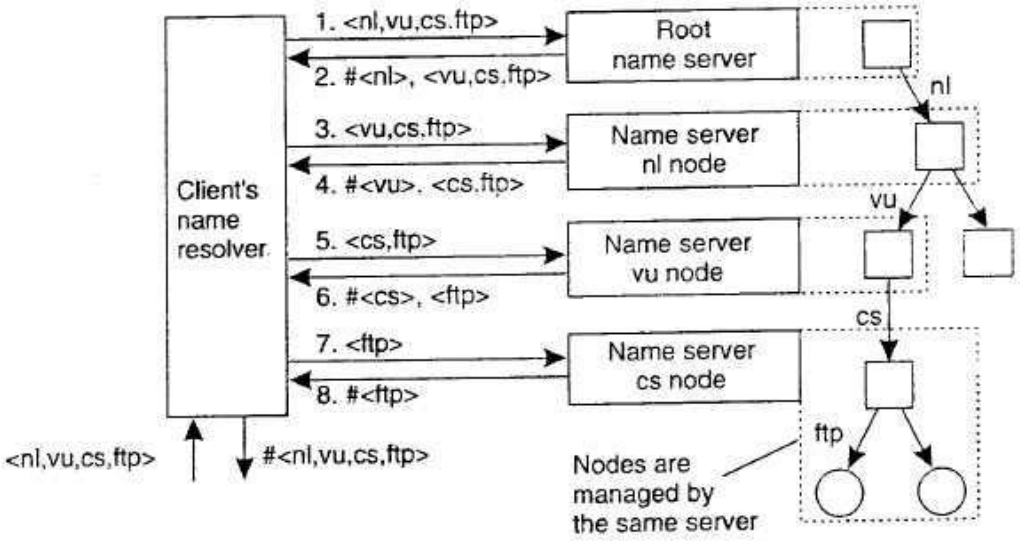


Figure 5-15. The principle of iterative name resolution.

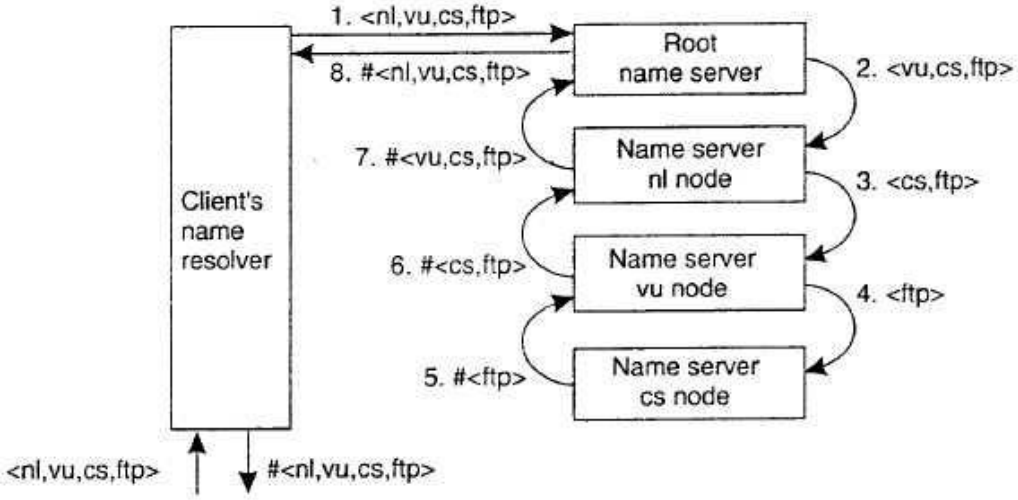


Figure 5-16. The principle of recursive name resolution.

مقایسه این دو روش:

تعداد ارتباط: بازگشتی بهتر است. مثلاً اولی در هلند است هی باید برویم و بیاییم.
از نظر Caching: بازگشتی بهتر است زیرا وقتی یکبار برود پایین برای دفعه بعد دارد.
پس:
مزایای بازگشتی: Communication Delay کم است. از Caching بهتر استفاده می‌کند.
معایب بازگشتی: لود روی ریشه خیلی زیاد است.
نتیجه: ترکیبی: لایه های بالا تکراری باشد لایه های پایین می‌توانند بازگشتی باشد.

فصل 6: همگام سازی

فهرست مطالب

- همگام سازی ساعت یا Clock Synchronization

  الگوریتم‌های همگام سازی ساعت

  کریستیان (Time Server)
  برکلی
  همگام‌سازی در شبکه‌های بیسیم

- ساعت‌های منطقی یا Logical Clocks

  ساعت‌های منطقی لامپورت
  ساعت‌های برداری

- انحصار متقابل یا Mutual Exclusion

  الگوریتم متمرکز
  الگوریتم غیرمتمرکز
  الگوریتم توزیعی
  الگوریتم Token Ring
  مقایسه الگوریتم‌ها

- موقعیت یابی جهانی گره‌ها یا Global Positioning of Nodes

- الگوریتم‌های گزینش یا Election Algorithms

  الگوریتم‌های سنتی گزینش
  گزینش در محیطهای بیسیم
  Election in Large-Scale Systems

- **Clock Synchronization:** هیچ دو سیستم امکان ندارد دارای یک Clock باشند و به تدریج از هم دور می شوند. به اینکه از هم دور می شوند می گویند Drift. مثال برنامه Make که Editor روی یک ماشین است و کامپایلر روی ماشین دیگر. فایلی که ادیت شده کامپایل نمی شود.

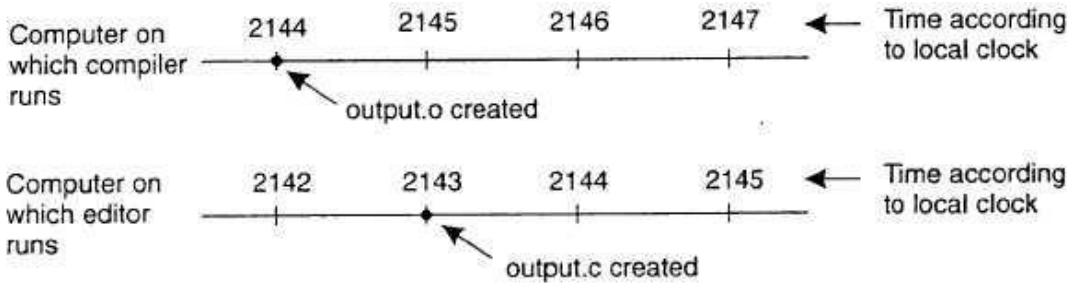


Figure 6-1. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

- **آیا می شود یک Global Time داشت؟** حتی ساعتهای اتمی نیز مانند هم نیستند اما از هر چیز دیگری دقیقتر است.

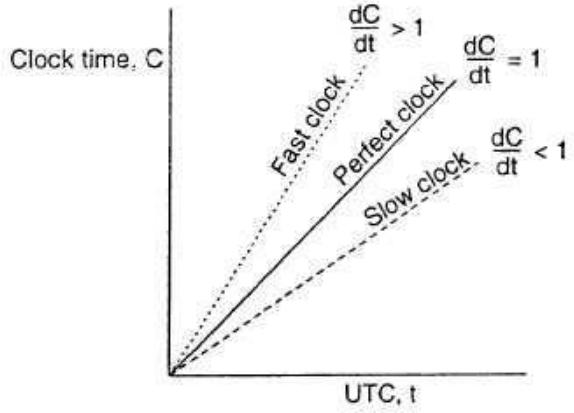


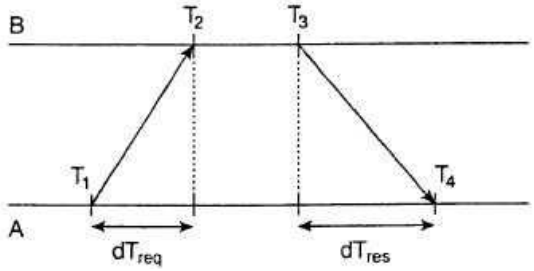
Figure 6-5. The relation between clock time and UTC when clocks tick at different rates.

محور عمودی ساعات ماست. در بدترین حالت dC / dt = 2P. سوال اگر این مقدار خطا را تحمل کنیم ساعت را چند وقت به چند وقت باید تنظیم کنیم؟

- **UTC (Universal Time Coordinated):** ساعت جهانی است.
- **الگوریتم‌های همگام‌سازی ساعت**
  - **الگوریتم همگام‌سازی کریستیان (Time Server)**

در این روش از یک Time Server استفاده می‌شود. پس ارسال پیام می‌کنیم و خواهیم داشت:

$$Q = T_3 + [(T_2 - T_1) + (T_4 - T_3)] / 2$$



باید دقت کنیم زمان یک سیستم هرگز به عقب باز نمی‌گردد. اگر ساعات ما جلو بود ساعت را کند می‌کنیم. اگر عقب بود می‌توانیم جلو ببریم.

  - **الگوریتم برکلی**

مثلاً سه سیستم را می‌خواهد با هم سنکرون کند. هر کدام را می‌توان مرجع گرفت. مثلاً بالایی را مرجع می‌گیریم اسمش را می‌گذاریم Time daemon. اگر از کار افتاد دیگری را جایگزین می‌کنیم. هر چند وقت به چند وقت می‌خواهیم سنکرون‌سازی کنیم.

1. Time daemon ساعت خودش را به همه اعلام می‌کند.
2. پاسخ می‌گیرد هر کدام چقدر عقب یا جلو هستند. بین اینها میانگین می‌گیرد. مثلاً همه باید بشود 3:5.
3. به هر کس اعلام می‌کند تا ساعتشان را تنظیم کنند. کسی نمی‌تواند عقب برگردد. باید کند شود تا بقیه به او برسند. شکل c غلط است نمی‌تواند برگردد.

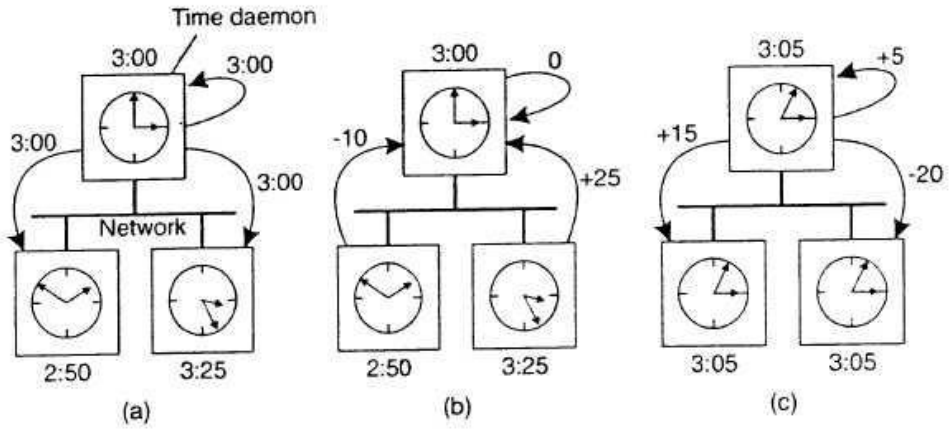


Figure 6-7. (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

- **همگام‌سازی در شبکه‌های بیسیم**

بر اساس پروتکل RBS (Reference Broadcast Synchronization) کار می‌کند. در شبکه بیسیم یکی Broadcast می‌کند. در این روش هدف این است که فقط گیرنده-ها هماهنگ شوند. دو نود p و q از فرستنده ساعت را می‌گیرند و با هم در تماس هستند و سعی می‌کنند ساعت خودشان را هماهنگ کنند. زمان انتشار برای همه یکسان در نظر گرفته می‌شود. و Single hop هستند. پس:

$$\text{Offset}[p,q] = \frac{\sum_{K=1}^{M}(Tp,k - Tq,k)}{M}$$

اختلاف دو نود

از پیام‌های بیشتری میانگین میگیریم تا میانگین دقیق‌تر باشد. بعد نودی که عقب است خودش را جلو می‌کشد. M تعداد پیام‌ها می‌باشد. زمان هم از لحظه‌ای است که پیام ارسال می‌شود.

- **ساعت‌های منطقی**
  - **الگوریتم لمپارت:** سعی می‌کند Clockها را زمانی که لازم هست با هم سنکرون کند. وقتی دو ماشین کاری با هم ندارند چرا ساعتشان هماهنگ باشند. **اصل است است زمان ارسال پیام باید قبل از زمان دریافت باشد.**

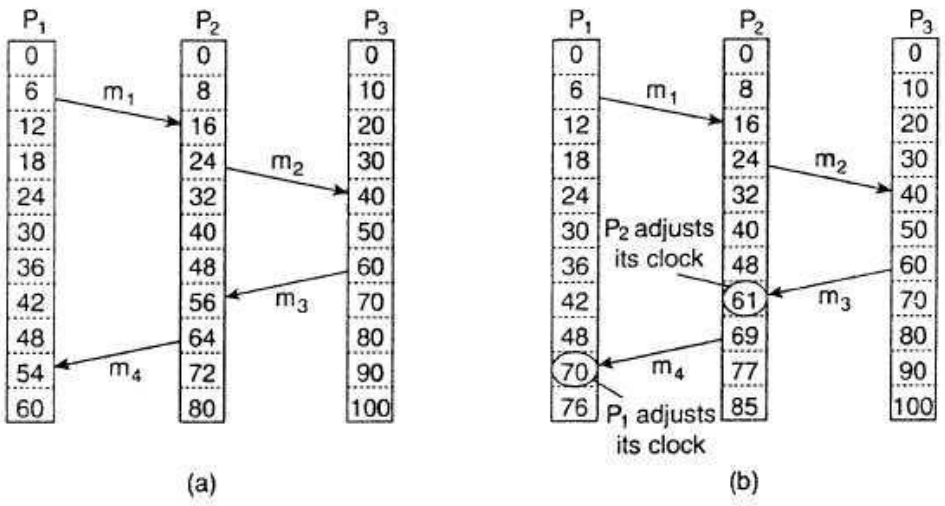


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

در سیستم اول هر گام 6 واحد زمانی دومی 8 واحد زمانی و سومی 10 واحد زمانی افزایش می‌یابد. در شکل a وقتی پسام m_3 ارسال می‌شود زمان 56 عقب تر از 60 است اصلاح می‌شود به 61 که در شکل b دیده می‌شود. بقیه هم می‌شوند 69 و 77 و 85. در شکل b هم پیام m_4 ممکن نیست اصلاح می‌شود.

این اصلاح زمان کجا انجام می‌شود کی انجام میدهد؟ Middleware. برنامه کاربردی درگیر شود که کار Develop خیلی سخت می‌شود.

- **چند پخشی مرتب یا Ordered Multicasting**

بانکی هست که database آن Replication دارد. یک حساب 1000 دلار موجودی دارد و قرار است یک جا 1% سود دهد و یک جا 100 دلار به حسابش اضاف شود. در اینجا ترتیب مهم است. برای رفع مغایرت ابتدا یک صف تشکیل می‌دهیم برای هر طرف. ابتدا اینها را در صف می‌گذاریم ولی اجرا نمی‌کنیم تا Ack نگرفته‌ایم اجرا نمی-کنیم. زمانهای گیرنده و فرستنده با الگوریتم لمپارت همگام شده‌اند. و اینها که در صف گذاشتیم Time Stamp دارند. وقتی Ack آمد برای M1 حالا M2 را هم که از قبل داریم زمانها را مقایسه می‌کنیم. در اولی Ack را می‌گیرد ولی ترتیبش درست است. هر دو آخر یک چیز را می‌بینند.

- **ساعت‌های برداری یا Vector Clocks**

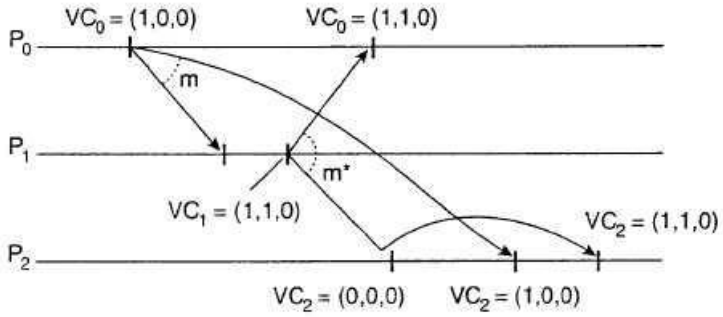


Figure 6-13. Enforcing causal communication.

در اینجا فرض شده است سه Process داریم. رابطه علیتی در اینجا مهم است. یک بردار سه تایی برای هر ارسال داریم. زمان ارسال مولفه خودش را ابتدا یکی اضافه می‌کنیم بعد ارسال می‌کنیم. در ابتدا P_0 دو پیام برای P_1 و P_2 می‌فرستد. قبل از ارسال یک واحد به مولفه خودش اضافه می‌کند پس (1,0,0) را می‌فرستد. P_1 می-گیرد بعد P_1 دو پیام برای P_0 و P_2 می‌فرستد یعنی (1,1,0). P_2 چون هنوز پیام قبلی را دریافت نکرده است دریافت را به تاخیر می‌اندازد تا پیام اول رابگیرد بعد پیام دوم را قبول می‌کند مثلاً دومی بافر می‌شود. به این روش ترتیب دریافت پیام‌ها رعایت شده است. این Ordering کجا انجام شود؟ در برنامه کاربردی چون مفهوم اطلاعات را می‌داند که اشکال این است که شفافیت از بین می‌رود و نیز Develop سخت می-شود.

- **انحصار متقابل یا Mutual Exclusion**

کل مسئله این است که یک منبع داریم و می‌خواهیم ببینیم در یک سیستم توزیعی چگونه بطور انحصاری از آن استفاده کنیم.

- **الگوریتم متمرکز یا Centralized**

یک Coordinator Process درست می‌کنیم که منبع را کنترل می‌کند و برای اینکار یک صف دارد. فرض کنیم سه تا Process دیگر می‌خواهند به یک منبع دسترسی داشته باشند. هر کس که منبع را می‌خواهد یک تقاضا می‌دهد اگر منبع آزاد بود پاسخ می‌دهیم که اشکال ندارد اما اگر منبع آزاد نبود و تقاضایی آمد آنرا در صف می‌گذاریم و پاسخ هم نمی‌دهیم. وقتی اولی کارش با منبع تمام شد Release می-دهد بعد Coordinator Process به تقاضای بعدی پاسخ می‌دهد. دقیقاً کار سمافور را

می‌کند. برای اینکه مطمئن شویم Coordinator Process زنده است می‌تواند پاسخ دهد ولی سرویس ندهد.
ایراد: Coordinator Process, Single Point of Failure است. بعداً می‌بینیم با استفاده از رای گیری اگر زنده نبود جایگزین می‌کنیم.

- **الگوریتم غیرمتمرکز یا Decentralized**

فرض کنید از برای هر منبعی n تا Coordinator Process داریم. این باعث می‌شود اگر چند تا از کار بیفتد هنوز کار کند. حال می‌خواهم از منبع استفاده انحصاری کنم. اگر m مجوز بگیرم به شرطی که $m > n/2$ باشد می‌توانم از منبع استفاده کنم. مسئله این است که هر Coordinator Process ممکن است Fail کند و دوباره برگردد. اشکال اینجاست که وقتی زنده شد آیا پاش هست قبلا مجوز داده؟ احتمالا نه. سیستم باید این موارد را تحمل کند. Hash Table اینجا به کار می‌آید. یک Ring از Coordinator Processها داریم . هر یک هم یک Key دارد. به این ترتیب هر کس رفت بیرون ایرادی ندارد جانشین دارد می‌توان رای گرفت. وقتی هم حالش خوب شد Join می‌کند.

حسن این روش: Fault Tolerant است.

- **الگوریتم توزیعی**

سه تا Process داریم. ایندفعه Coordinator ندارند. هر کدام خواست بصورت انحصاری به منبع دسترسی داشته باشد به دیگران اطلاع می‌دهد. از الگوریتم لمپارت استفاده می‌کنند و به همراه تقاضا Timestamp هم می‌زهد. به خودش هم Request می‌فرستد. Process شماره یک به هر دو OK میدهد چون اصلاً داوطلب نبود. Process شماره دو به صفر OK میدهد چون Timestamp 8 کمتر از دوازده است و Process شماره سه هم به 2 OK نمی‌دهد. نهایتا 0 از بقیه OK می‌گیرد و انتخاب می‌شود. الگوریتم ساده‌ای است ولی به درد نمی‌خورد.
ایراد: الگوریتم قبلی یک Single Point of Failure است این هر نود از کار بیفتد کل الگوریتم خراب می‌شود. نکته دوم گروه را باید کامل بشناسد یعنی گروه ثابت است.

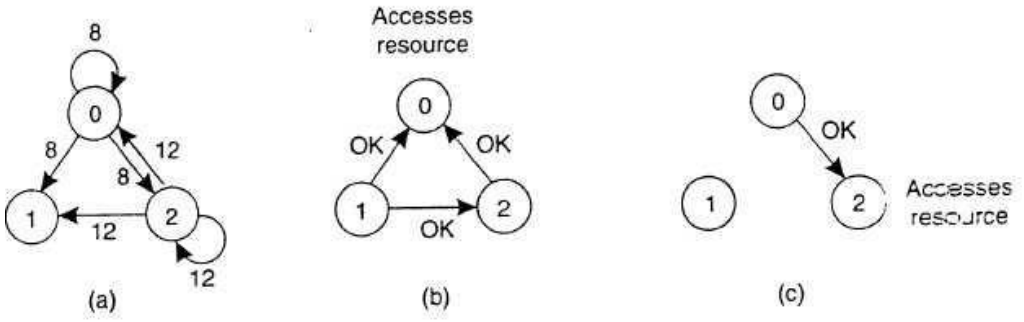


Figure 6-15. (a) Two processes want to access a shared resource at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now go ahead.

- **الگوریتم Token Ring**

فرایندهایی که قرار است در استفاده انحصاری استفاده کنند را در یک حلقه منطقی قرار می‌دهد. یعنی هر کس بداند قبل و بعدش کیست. یک پیام در این حلقه حرکت می‌کند و هرکس این توکی را دارد مجوز دارد از منبع استفاده کند. اگر توکن را داشت فقط یک عمل می‌تواند انجام دهد.
مشکل: اگر یک نود از کار بیفتد حلقه قطع میشود. اگر یک نود از کار افتاد قبلی باید برای بعدی این بفرستد. مشکل بعد گم شدن توکن است. تشخیص اینکه توکن گم شده یا تاخیر دارد. شاید باید محدوده زمانی در نظر بگیریم و توکن ایجاد کنیم.

راه حل: هر نود به جز قبلی, قبلی و بعدی بعدی را هم بشناسد. با این منطق یعنی همه باید تمام رینگ را بشناسند.

- **مقایسه روشها**

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Figure 6-17. A comparison of three mutual exclusion algorithms.

- **الگوریتم‌های Election**

در اغلب الگوریتم‌های توزیع شده نیاز به هماهنگ کننده داریم. اشکال بزرگ هماهنگ کننده Single Point of Failure است. بحثی که داریم این است که اگر هماهنگ کننده Fail کرد, سیستم بتواند یک هماهنگ کننده دیگر انتخاب کند.

- **الگوریتم Bully**

همه قصد دارند Coordinator شوند. در این مثال 8 نود در نظر گرفته شده است. سیستم بر اساس معیاری مثلاً توان پردازشی با حتی بدون معیار به هر نود یک شماره Identifier می‌دهد. گره‌ای که شماره بزرگتر را دارد استعداد Coordinate شدن دارد. شماره هفت Coordinator بوده از کار می‌افتد. اولین نودی که متوجه می‌شود Coordinator نیست گره 4 است و فرض می‌کند بین خودش و هفت ممکن است گره‌هایی باشند. یک Election Message برای 5 و 6 و 7 می‌فرستد. اگر آنها زنده باشند به 4 جواب می‌دهد که ما هستیم برو کنار. بعد 5 و 6 پیام Election Message برای باتریهایشان می‌فرستند. 6 به 5 می‌گوید برو کنار. اما خود 6 پاسخی از کسی نمی‌گیرد پس می‌شود Coordinator. حالا Broadcast می‌کند به همه می‌گوید من Coordinator هستم. حالا اگر 7 زنده شد یک Election Message می‌فرستد. نمی‌داند بالاتر از خودش کیست به همه می‌فرستد و کسی جواب نمی‌دهد پس خودش Coordinator می‌شود.

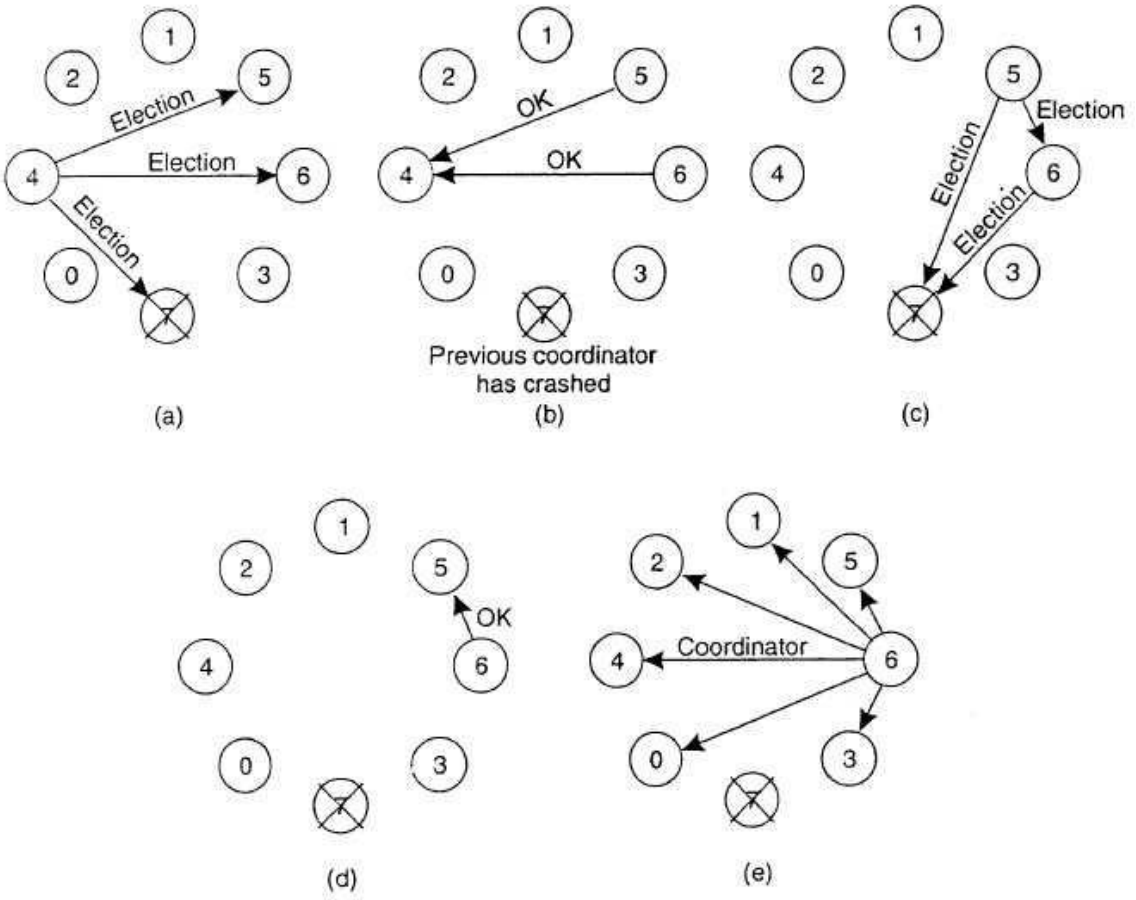


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

- **الگوریتم انتخاب Ring**

این هم یک حلقه منطقی است. یعنی هر نودی باید قبل و بعدش و کل Ring را بشناسد. این رینگ فقط برای Election بکار می‌رود. باز نود 7, Fail می‌کند. یکی از نودها شماره 5 زودتر از بقیه می‌فهمد که Coordinator نیست یک پیام به 6 میدهد و شماره خودش را اعلام می‌کند [5]. گره شماره 6 میخواهد پیام به 7 بدهد نمی‌تواند به بعدی یعنی گره 0 پیام [5,6] را می‌دهد. و به همین ترتیب تا به 5 برسد. بعد 5 اعداد را مقایسه می‌کند متوجه می‌شود 6, Coordinator است و بعد یک پیام در حلقه می‌فرستد و به بقیه اطلاع می‌دهد که 6, Coordinator است.
حالا اگر بعد از 5 شماره 3 هم بفهمد Coordinator نیست شروع به ارسال پیام کند. هیچ اتفاقی نمی‌افتد 3 هم میفهمد 6, Coordinator است. و 6 بصورت انحصاری استفاده ای یک منبع را مدیریت می‌کند.

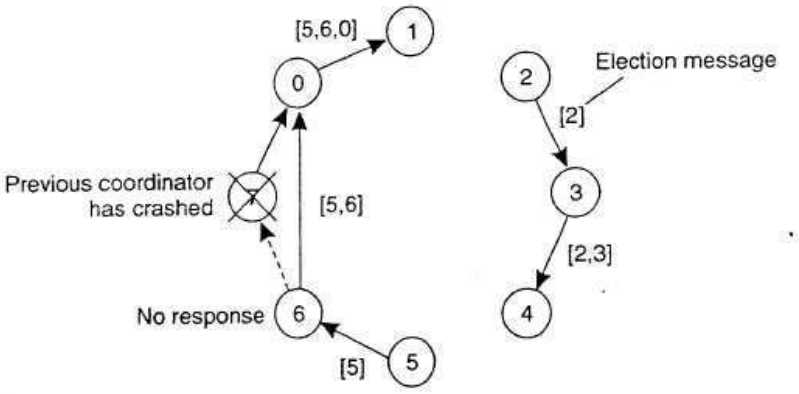


Figure 6-21. Election algorithm using a ring.

- **الگوریتم انتخاب در محیط بیسیم**

نودها ارزش گذاری شده است مثلاً بر اساس انرژی یا هر فاکتور دیگر. هر نود یک جفت (ارزش, نام) دارد. فرض کنیم می‌خواهیم یک Coordinator انتخاب کنیم. هر نود می‌تواند شروع کند. در مثال ما نود 4 شروع می‌کند. پیام Election را به b و j می‌فرستد. و آنها می‌فهمند او دارد Election انجام می‌دهد. آنها هم شروع به Forward می‌کنند. g وقتی پیام Election را از b گرفت دیگر از j قبول نمی‌کند. این کار ادامه پیدا می‌کند تا به آخر.

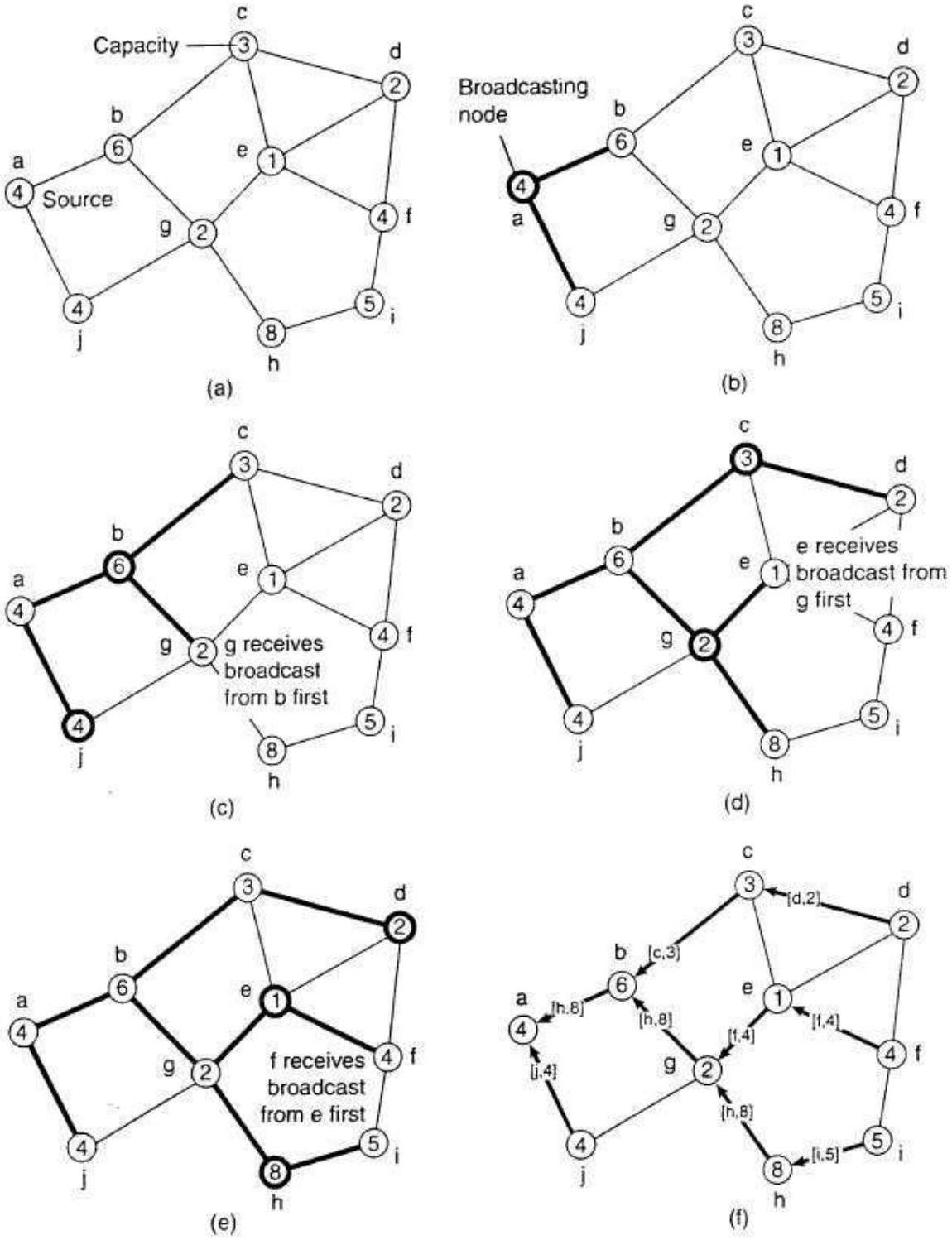


Figure 6-22. Election algorithm in a wireless network, with node *a* as the source. (a) Initial network. (b)–(e) The build-tree phase (last broadcast step by nodes f and i not shown). (f) Reporting of best node to source.

حالا همه پاسخ را بر می‌گردانند. مثلاً f پیام [f,4] را ارسال می‌کند یعنی می‌گوید من f هستم ارزشم هم 4 است. وقتی مثلاً g می‌خواهد اطلاعات را برگرداند ارزش‌های 8 و 4 را دارد و بزرگتر یعنی [h,8] را بازمی‌گرداند.

- **الگوریتم انتخاب در سیستم های بزرگ**

در سیستم‌های بزرگ نمی‌آیند بین چندین هزار نود انتخابات برگزار کنند تا Coordinator مشخص شود. در سیستم‌های بزرگ Super Peer انتخاب می‌شود. این Super Peer ها باید بطور متوازن در سیستم توزیع شده باشند تا به یک سری نود سرویس بدهند. از مثال Hash Table استفاده می‌کنیم. در آن سیستم می‌خواهیم تعدادی Super Peer انتخاب کنیم. فرض کنیم فضا 8 بیتی است. در این 256 نود می‌خواهیم 8 نود بعنوان Supper Peer انتخاب کنیم. پس سه بیت کافی است. می‌گوییم Supper Peerها آنهایی هستند که بصورت 0 0 0 0 0 x x x هستند. یعنی آدرس Super Peerها مشخص است. یک نود از کجا بفهمد Super Peer است؟ کافی است آدرس خودش را با 11100000 , And کند. هر نود هم کافی است آدرس خودش را با 11100000, And کند تا Super Peer خودش را پیدا کند.