16 Internet Multimedia Content Distribution

In the previous chapter, we have introduced the basic Internet infrastructure and protocols for real-time multimedia services. These protocol suites have been incorporated by client-side media players receiving streams from media servers over the Internet. The key functionality for multimedia data transfer is provided by the Real-Time Transport Protocol (RTP), including payload identification, sequence numbering for loss detection, and timestamping for playback control. Running on top of UDP, RTP itself does not guarantee Quality of Service (QoS), but relies on its companion, the RTP Control Protocol (RTCP), to monitor the network status and provide feedback for application-layer adaptation. The Real-Time Streaming Protocol (RTSP) coordinates the delivery of media objects and enables a rich set of controls for interactive playback.

Figure 16.1 shows a basic client/server-based multimedia media streaming system using the real-time protocol suite. It works fine for small-scale media content distribution over the Internet, in which media objects such as videos can be served by a single server to these users. Such an architecture has quickly become infeasible when more media contents are made available online and more users are network- and multimedia-ready.

There have been significant studies on efficient content distribution over the Internet, targeting a large number of users. Most of them were optimized for delivering conventional web objects (e.g., HTML pages or small images) or for file download. Streaming media however poses a new set of challenges [1–3]:

Huge size: A conventional static web object is typically of the order of 1–100 K bytes. In contrast, a media object has a high data rate and a long playback duration, which combined yield a huge data volume. For example, a one-hour standard MPEG-1 video has a total volume of about 675 MB. Later standards have successfully improved the compression efficiency, but the video object sizes, even with the latest H.265 compression, are still large, not to mention the new High Definition (HD) and 3D videos.

Intensive bandwidth use: The streaming nature of delivery requires a significant amount of disk I/O and network bandwidth, sustaining over a long period.

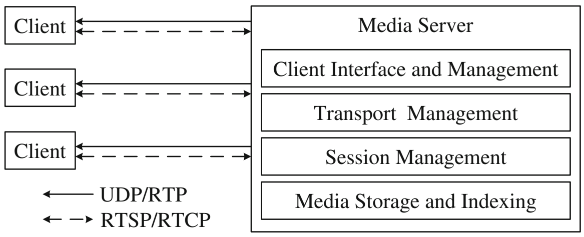

Fig. 16.1 A basic client/server-based media streaming system

Rich interactivity: The long playback duration of a streaming object also enables various client–server interactions. As an example, existing studies found that nearly 90 % media playbacks are terminated prematurely by clients [4]. In addition, during a playback, a client often expects VCR-like operations, such as fast-forward and rewind. This implies that access rates might be different for different portions of a stream.

Many emerging applications, such as Internet TV and live event broadcast, further demand real-time multimedia streaming services with a massive audience, which can easily overwhelm the server. The scaling challenge for such multimedia content distribution is enormous. To reach 100 million viewers, delivery of TV quality video encoded in MPEG-4 (1.5 Mbps) may require an aggregate capacity of 1.5 Tbps. To put things into perspective, consider two large-scale Internet video broadcasts: the CBS broadcast of the NCAA tournament in March 2006, which at the peak has 268,000 simultaneous viewers, and the opening ceremony of the London Summer Olympics in July 2012, which drew a peak broadcast audience of 27.1 million, of which 9.2 million were via BBC's mobile site and 2.3 million on tablets. Even with low bandwidth Internet video of 400 Kbps, the CBS/NCAA broadcast needs more than 100 Gbps server and network bandwidth; on the busiest day of the London Olympics, BBC's website delivered 2.8 Petabyte data, with the peak traffic at 700 Gbps. These can hardly be handled by any single server.

In this chapter, we discuss content distribution mechanisms that enable highly scalable multimedia content streaming, including proxy caching, multicast, content distribution networks, peer-to-peer, and HTTP streaming.

16.1 Proxy Caching

To reduce client-perceived access latencies as well as server/network loads, an effective means is to cache frequently used data at proxies close to clients. It also enhances the availability of objects and mitigates packet losses, as a local transmission is generally more reliable than a remote transmission. Proxy caching thus has become one of the vital components in virtually all web systems [5]. Streaming media, particularly those pre-stored, could also benefit significant performance improvement

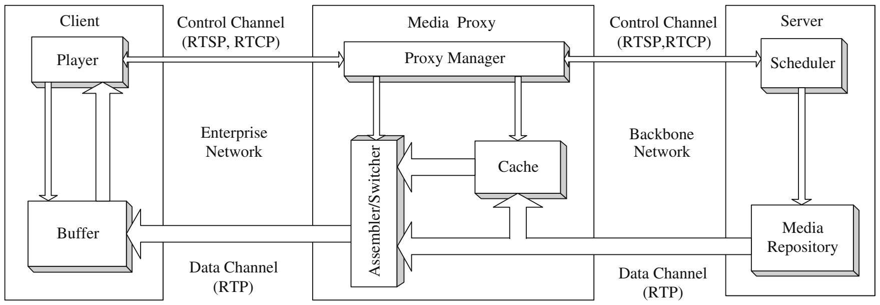

Fig. 16.2 A generic system diagram of proxy-assisted media streaming using RTP/RTCP/RTSP

from proxy caching, given their static nature in content and highly localized access interests.

Media caching however has many distinct focuses from conventional web caching [6]. On one hand, traditional web caching spends considerable effort to ensure that the copies at the origin servers and the proxy are consistent. Since the content of an audio/video object is rarely updated, such management issues are less critical in media caching. On the other hand, given the high resource requirements, caching each media object entirely at a proxy is hardly practical. It is necessary to decide which portions of which objects to be cached under cache space, disk I/O, and network I/O constraints, so that the benefit of caching outweighs the overhead for synchronizing different portions of a video stream in the proxy and in the server. A generic system diagram of proxy-cache-assisted media streaming is depicted in Fig. 16.2.

The proxy must reply to a client's PLAY request and initiate transmission of RTP and RTCP messages to the client for the cached portion, while requesting the uncached portion(s) from the server. Such fetching can be achieved through an RTSP `Range request` specifying the playback points, as illustrated in Fig. 16.3. The `Range request` also enables clients to retrieve different segments of a media object from multiple servers or proxies, if needed.

According to the selection of the portions to cache, we can classify existing algorithms into four categories: *sliding-interval caching*, *prefix caching*, *segment caching*, and *rate-split caching*.

16.1.1 Sliding-Interval Caching

This algorithm caches a sliding interval of a media object to facilitate consecutive accesses [7,8]. For illustration, given two consecutive requests for the same object, the first request may access the object from the server and incrementally store it into the proxy cache; the second request can then access the cached portion and release it after the access. If the two requests arrive close in time, only a small portion of the media object needs to be cached at any time instance, and yet the second request can be completely satisfied from the proxy, as illustrated in Fig. 16.4. In general, if

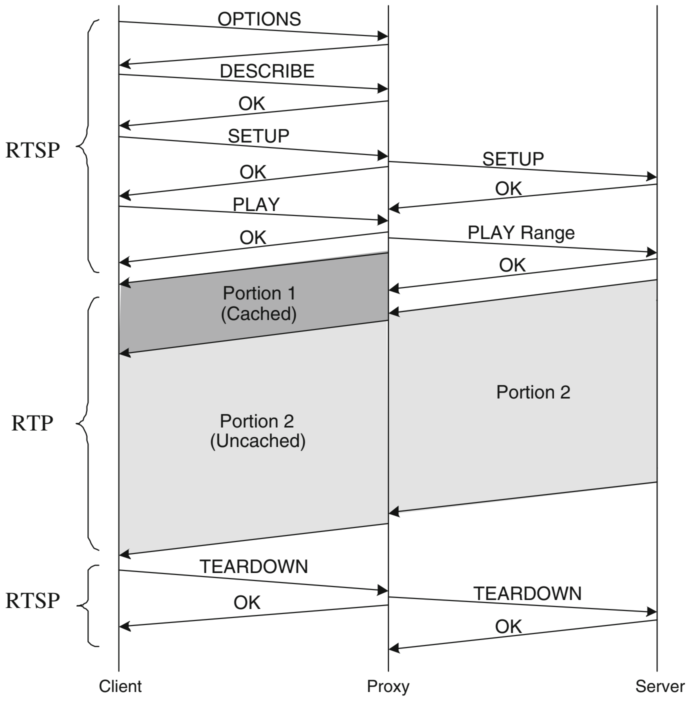

Fig. 16.3 Operations for streaming with partial caching

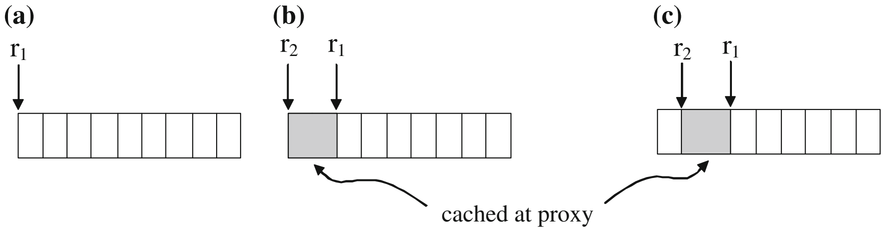

Fig. 16.4 An illustration of sliding-interval caching. The object consists of 9 frames, each requiring one unit time to deliver from the proxy to a client. Requests 1 and 2 arrive at times 0 and 2, respectively. To serve request 2, only two frames need to be cached at any time instance. **a** Time 0: request 1 arrives; **b** Time 1–2: frames 1 and 2 accessed by request 1 and cached; request 2 arrives; **c** Time 2–3: frame 3 accessed by request 1 and cached; frame 1 read by request 2 and released

multiple requests for an object arrive in a short period, a set of adjacent intervals can be grouped to form a *run*, of which the cached portion will be released only after the last request has been satisfied.

Sliding-interval caching can significantly reduce the network bandwidth consumption and start-up delay for subsequent accesses. However, as the cached portion is dynamically updated with playback, the sliding-interval caching involves high disk bandwidth demands; in the worst case, it would double the disk I/O due

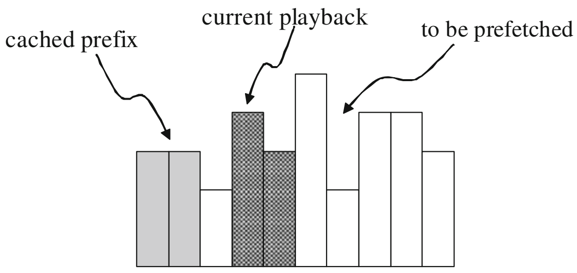

Fig. 16.5 A snapshot of prefix caching

to the concurrent read/write operations. To effectively utilize the available cache resources, the caching policy can be modeled as a two-constraint knapsack problem given the space and bandwidth requirements of each object [7], and heuristics can be developed to dynamically select the caching granularity, i.e., the run length, so as to balance the bandwidth and space usages. Given that memory spaces are large nowadays, it is also possible to allocate memory buffers to accommodate media data and thus avoid the intensive disk read/write [8].

The effectiveness of sliding-interval caching diminishes with the increase in the access intervals. If the access interval of the same object is longer than the duration of the playback, the algorithm is degenerated to the unaffordable full-object caching. To address this issue, it is preferable to retain the cached content over a relatively long time period, and most of the caching algorithms to be discussed in the rest of this section fall into this category.

16.1.2 Prefix Caching and Segment Caching

This algorithm caches the initial portion of a media object, called *prefix*, at a proxy [9]. Upon receiving a client request, the proxy immediately delivers the prefix to the client and, meanwhile, fetches the remaining portion, the *suffix*, from the origin server and relays it to the client (see Fig. 16.5). As the proxy is generally closer to the clients than the origin server, the start-up delay for a playback can be remarkably reduced.

Segment caching generalizes the prefix caching paradigm by partitioning a media object into a series of segments, differentiating their respective utilities, and making caching decision accordingly (see Fig. 16.6). A salient feature of segment-based caching is its support to preview and such VCR-like operations as random access, fast-forward, and rewind. For example, some key segments of a media object (*hotspots*), as identified by content providers, can be cached [10]. When a client requests the object, the proxy first delivers the hotspots to provide an overview of the stream; the client can then decide whether to play the entire stream or quickly jump to some specific portion introduced by a hotspot. Furthermore, in fast-forwarding and rewinding operations, only the corresponding hotspots are delivered and displayed, while other portions are skipped. As such, the load of the server and backbone network can be greatly reduced, but the client will not miss any important segments in the media object.

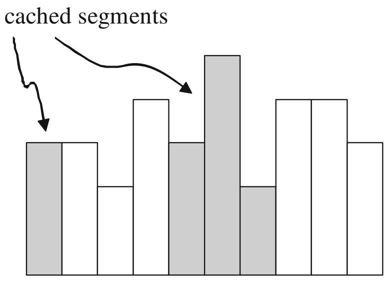

Fig. 16.6 An illustration of segment caching

The segments are not necessarily of the same length, nor predefined. One solution is to group the frames of a media object into variable-sized segments, with the length increasing exponentially with the distance from the start of the media stream, i.e., the size of segment i is 2^{i-1}, which consists of frames 2^{i-1}, $2^{i-1}+1, \ldots, 2^{i}-1$ [11]. The utility of a segment is calculated as the ratio of the segment reference frequency over its distance from the beginning segment, which favors caching the initial segments as well as those with higher access frequencies. The proxy can also quickly adapt to the changing access patterns of cached objects by discarding big chunks as needed. If the access frequencies are not known in advance, segmentation should be postponed as late as possible (called *lazy segmentation*), thus allowing the proxy to collect a sufficient amount of access statistics to improve cache effectiveness [4].

16.1.3 Rate-Split Caching and Work-Ahead Smoothing

While all the aforementioned caching algorithms partition a media object along the time axis, the rate-split caching (also known as *video staging*) [12] partitions a media along the rate axis: the upper part will be cached at the proxy, whereas the lower part will remain stored at the origin server (see Fig. 16.7). This type of partitioning is particularly attractive for VBR streaming, as only the lower part of a nearly constant rate has to be delivered through the backbone network. For a QoS network with resource reservation, if the bandwidth is reserved based on the peak rate of a stream, caching the upper part at the proxy significantly reduces the rate variability, which in turn improves the backbone bandwidth utilization.

If the client has buffer capability (refer to Sect. 15.5.3 in the previous chapter), *work-ahead smoothing* [13] can be incorporated into video staging to further reduce the backbone bandwidth requirement.

Define $d(t)$ to be the size of frame t, where $t \in 1, 2, \ldots, N$, and N is the total number of frames in the video. Similarly, define $a(t)$ to be the amount of data transmitted by the video server during the playback time for frame t (for short, call it at time t). Let $D(t)$ be the total data consumed and $A(t)$ be the total data sent at time t. Formally, we have

$$D(t) = \sum_{i=1}^{t} d(i) \tag{16.1}$$

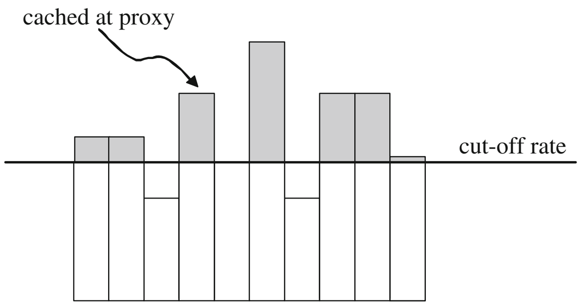

Fig. 16.7 An illustration of rate-split caching

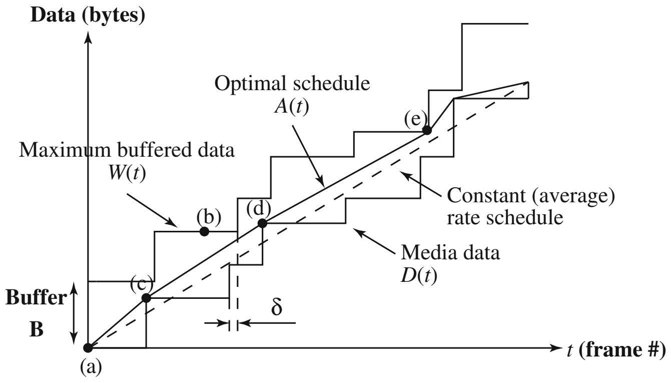

Fig. 16.8 The optimal smoothing plan for a specific video and buffer size. In this case, it is not feasible to transmit at the constant (average) data rate

$$A(t) = \sum_{i=1}^{t} a(i) \tag{16.2}$$

Let the buffer size be B. At any time t, the maximum amount of data that can be received without overflowing the buffer during time $1, \ldots, t$ is $W(t) = D(t-1)+B$. Now it is easy to state the conditions for a server transmission rate that avoids buffer overflow or underflow:

$$D(t) \leq A(t) \leq W(t) \tag{16.3}$$

To avoid buffer overflow or underflow throughout the video's duration, Eq. (16.3) has to hold for all $t \in 1, 2, \ldots, N$. Define S to be the server transmission schedule (or plan), i.e., $S = a(1), a(2), \ldots, a(N)$. S is called a *feasible transmission schedule* if for all t, S obeys Eq. (16.3). Figure 16.8 illustrates the bounding curves $D(t)$ and $W(t)$ and shows that a constant (average)-bitrate transmission plan is not feasible for this video, because simply adopting the average bitrate would cause underflow.

When frame sizes $d(t)$ for all t are known ahead of the transmission time, the server can plan ahead to generate an optimal transmission schedule that is feasible and minimize the peak transmission rate [13]. Additionally, the plan minimizes schedule variance, optimally trying to smooth the transmission as much as possible.

We can think of this technique as stretching a rubber band from $D(1)$ to $D(N)$ bounded by the curves defined by $D(t)$ and $W(t)$. The slope of the total-data-transmitted curve is the transmission data rate. Intuitively, we can minimize the slope (or the peak rate) if, whenever the transmission data rate has to change, it does so as early as possible in the transmission plan.

As an illustration, consider Fig. 16.8 where the server starts transmitting data when the prefetch buffer is at state (a). It determines that to avoid buffer underflow at point (c), the transmission rate has to be high enough to have enough data at point (c). However, at that rate, the buffer will overflow at point (b). Hence it is necessary to reduce the transmission rate somewhere between points (c) and (b).

The earliest such point (that minimizes transmission rate variability) is point (c). The rate is reduced to a lower constant bitrate until point (d), where the buffer is empty. After that, the rate must be further reduced (to lower than the average bitrate!) to avoid overflow until point (e), when the rate must finally be increased.

Consider any interval $[p, q]$ and let $B(t)$ represent the amount of data in the buffer at time t. Then the maximum constant data rate that can be used without overflowing the buffer is given by $R_{\max}$:

$$R_{\max} = \min_{p+1 \le t \le q} \frac{W(t) - (D(p) + B(p))}{t - p} \tag{16.4}$$

The minimum data rate that must be used over the same interval to avoid underflow is given by $R_{\min}$:

$$R_{\min} = \max_{p+1 \le t \le q} \frac{D(t) - (D(p) + B(p))}{t - p} \tag{16.5}$$

Naturally it is required that $R_{\max} \geq R_{\min}$, otherwise no constant bitrate transmission is feasible over interval $[p, q]$. The algorithm to construct the optimal transmission plan starts with interval $[p, q = p + 1]$ and keeps incrementing q, each time recalculating $R_{\max}$ and $R_{\min}$. If $R_{\max}$ is to be increased, a rate segment is created with rate $R_{\max}$ over interval $[p, q_{\max}]$, where $q_{\max}$ is the latest point at which the buffer is full (the latest point in interval $[p, q]$ where $R_{\max}$ is achieved).

Equivalently, if $R_{\min}$ is to be decreased, a rate segment is created with rate $R_{\min}$ over interval $[p, q_{\min}]$, where $q_{\min}$ is the latest point at which the buffer is empty.

Planning transmission rates can readily consider the maximum allowed network jitter. Suppose there is no delay in the receiving rate. At time t, $A(t)$ bytes of data have been received, which must not exceed $W(t)$. Now suppose the network delay is at its worst—δ sec maximum delay. Video decoding will be delayed by δ seconds, so the prefetch buffer will not be freed. Hence the $D(t)$ curve needs to be modified to a $D(t - \delta)$ curve, as depicted in Fig. 16.8. This provides protection against overflow or underflow in the plan for a given maximum delay jitter.

We can either perform smoothing first and then select the cut-off rate for video staging, or select the cut-off rate and then perform smoothing. Empirical evaluation has shown that a significant bandwidth reduction can be achieved with a reasonably small cache space [12].

Table 16.1 Comparison of Proxy Caching Algorithms

		Sliding-interval caching	Prefix caching	Segment caching	Rate-split caching
Cached portion		Sliding intervals	Prefix	Segments	Portion of higher rate
VCR-like support		No	No	Yes	No
Resource demand	Disk I/O	High	Moderate	Moderate	Moderate
	Disk space	Low	Moderate	High	High
	Sync overhead	Low	Moderate	High	High
Performance improvement	Bandwidth reduction	High*	Moderate	Moderate	Moderate
	Start-up latency reduction	High*	High	High**	Moderate

* There is no reduction for the first request in a run.
** Assume the initial segment is cached.

16.1.4 Summary and Comparison

Table 16.1 summarizes the caching algorithms introduced above. While these features and metrics provide a general guideline for algorithm selection, the choice for a specific streaming system also largely depends on a number of practical issues, in particular, the complexity of the implementation. Many of these algorithms have been employed in commercial systems, demonstrating their viability and superiority. These algorithms are not necessarily exclusive to each other, and a combination of them may yield a better performance. For example, segment caching combined with prefix caching of each segment can reduce start-up latency for VCR-like random playback from any key-segment. If the cache space is abundant, the proxy can also devote certain space to assist work-ahead smoothing for variable-bit-rate (VBR) media [9]. With this smoothing cache, the proxy can prefetch large frames in advance of each burst to absorb delay jitter and bandwidth fluctuations of the server-to-proxy path. The delay of prefetching can be hidden by the prefix caching. Similar to sliding-interval caching, the content of the smoothing cache is dynamically updated with playback. The purposes however are different: the former is to improve cache hit for subsequent requests, while the latter is to facilitate work-ahead smoothing.

16.2 Content Distribution Networks (CDNs)

Caching is generally passive, in the sense that only if a user fetches an object would the object be cached at a proxy. In other words, a proxy needs time to fill up its cache space and there will be no immediate benefit for the first user accessing an object.

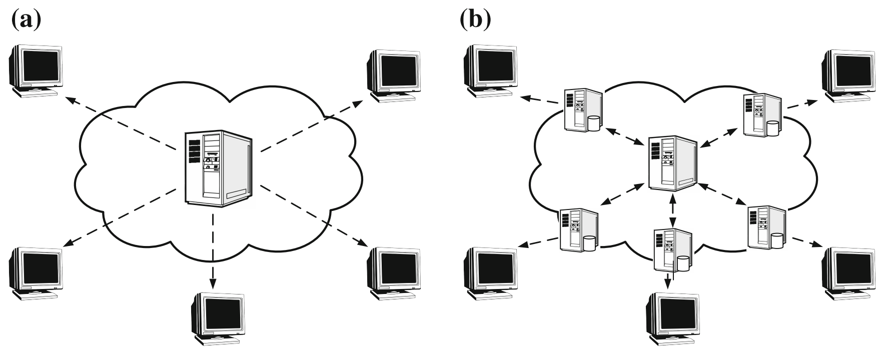

Fig. 16.9 Comparison between traditional single server and CDN. **a** Traditional Client/Server solution. **b** Content distribution network (CDN) solution

A more proactive solution is a *Content Delivery Network* or *Content Distribution Network* (CDN), which is a large geo-distributed system of servers deployed in datacenters across the Internet; these servers replicate content from the origin server, pushing them to network edges close to end-users, so as to avoid middle-mile bottlenecks as much as possible (see Fig. 16.9). Originally for accelerating web accesses, this technology has rapidly evolved beyond facilitating static web content delivery. Today, CDNs serve a large fraction of the Internet data distribution, including both conventional web accesses and file download, and new generation of applications like live streaming media, on-demand streaming media, and online social networks.

A CDN provider hosts the content from content providers (i.e., CDN customers) and delivers the content to users of interest. This is done by mirroring the content in replicated servers and then building a mapping system accordingly. When a user types a URL into his/her browser, the domain name of the URL is translated by the mapping system into the IP address of a CDN server that stores a replica of the content. The user is then redirected to the CDN server to fetch the content. This process is generally transparent to the user.

Figure 16.10 provides a high-level view of the request-routing in a CDN environment. The interaction flows are as follows:

Step 1. The user requests content from the content provider by specifying its URL in the web browser, and the request is directed to its origin server.

Step 2. When the origin server receives the request, it makes a decision to provide only the basic content (e.g. index page of the website), leaving others to CDN.

Step 3. To serve the high bandwidth demanding and frequently asked content (e.g., embedded objects fresh content, navigation bar, banner ads, etc.), the origin server redirects user's request to the CDN provider.

Step 4. Using the mapping algorithm, the CDN provider selects the replica server.

Step 5. The selected server serves the user by providing the replicated copy of the requested object.

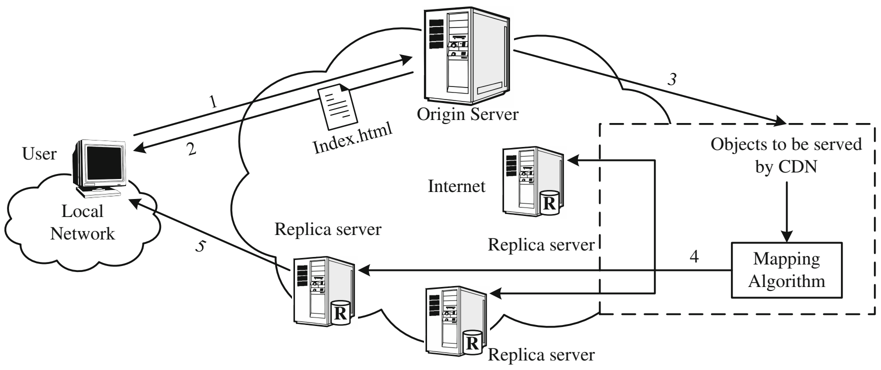

Fig. 16.10 A high-level view of request-routing in a CDN

To assign the user to the best possible CDN server, the mapping system bases its answers on large amounts of historical and real-time data that have been collected and processed regarding the global network and server conditions. For performance optimization, the locations of the fewest hops or the highest server availability will be chosen. For cost-optimization, the least-expensive locations can be chosen instead. In the real world, these two goals tend to align, as the replicated servers that are close to a user may have the advantage of both performance and cost. As an example, we use the Traceroute tools (`tracert` in MS Windows) to track the path between our institution and Hulu, a major video streaming service provider. The path tracking result is shown below.

```
tracert www.hulu.com
Tracing route to a1700.g.akamai.net [142.231.1.173]
over a maximum of 30 hops:

  1   1 ms   2 ms   1 ms  199.60.1.254
  2   3 ms   2 ms   1 ms  142.58.45.70
  3   1 ms  <1 ms  <1 ms  142.58.45.46
  4   1 ms  <1 ms  <1 ms  van-hcc1360-x-1-bby-sh1125-
                          x-1.net.sfu.ca
                          [142.58.29.10]
  5   1 ms   1 ms   1 ms  ORAN-SFU-cr1.vantx1.BC.net
                          [142.231.1.45]
  6   2 ms   1 ms   1 ms  207.23.240.70
  7   1 ms   1 ms   1 ms  a142-231-1-173.deploy.ak
                          amaitechnologies.com
                          [142.231.1.173]
Trace complete.
```

It can be seen that the Hulu's web server we intended to reach is indeed an Akamai server (a1700.g.akamai.net), which is located in BCNet—a network that is close to our campus, while not in a network near Hulu's headquater in California. This suggests that Hulu is using the CDN service provided by Akamai, one of the world's largest CDN providers, and the specific server offered to us is the nearest to us, which is also of low cost as our campus network is closely associated with BCNet.

The CDN provider gets paid by content providers, i.e., its customers. In turn, it pays ISPs, carriers, and network operators for hosting its servers in their datacenters and for using their network resources. The amount of servers managed by a CDN provider can be very large. For example, Akamai maintains a network of 250,000 servers running in 80 countries worldwide. This large overlay network of servers effectively reduces bandwidth costs and content access delays, and increases the global availability of content. It creates sizable savings in capital and operational expenses, as the CDN customers no longer have to build their own large-size infrastructures that are not only expensive but also underutilized most of the time except during popular events. In addition, the CDN offers a content provider better protection from malicious attacks, because their large distributed server infrastructure can effectively absorb most of the attacking traffic.

16.2.1 Representative: Akamai Streaming CDN

For bandwidth-intensive streaming media, CDN provides better scalability by delivering the content over the last-mile from servers close to end-users. The virtually unlimited resources from a large CDN also reduces the pressure on content providers to accurately predict capacity needs and enables them to gracefully absorb bursts of user demand. This is also one of the key reasons toward the recent success of Cloud Computing, a more general form of CDN as we will discuss in Chap. 19.

For large and comprehensive CDN operators, like Akamai, the service platform could comprise multiple delivery networks, each being tailored to a specific type of content, e.g., static web content, dynamic news update, or streaming media, to name but a few. At a high level, these delivery networks share a similar architecture, but the underlying technology and implementation of each system component may differ so as to best suit the specific type of content.

We now have a closer look at Akamai's media streaming CDN, which has been widely used by such companies as Apple, Microsoft, and BBC for their video services [14]. In this streaming CDN, once a live stream is captured and encoded, the stream is sent to an Akamai server, called *entrypoints*. To avoid having this single entrypoint becoming the single point of failure, multiple copies of the stream can be sent to additional entrypoints. If any entrypoint goes down, other copies can be used for recovery. The stream's packets are then transported from the entrypoint to a subset of *edge servers* that are close to end-users.

Note that the transport system must simultaneously distribute thousands of live streams from their respective entrypoints to the subset of edge servers that are interested in the stream. To perform this task in a scalable fashion, an intermediate layer of servers called *reflectors* is used. Sitting between the entrypoints and the edge servers, each reflector can receive one or more streams from the entrypoints and then send those streams to one or more clusters of edge servers. This enables rapid

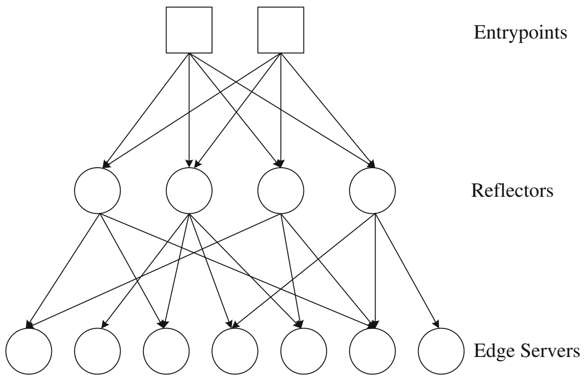

Fig. 16.11 Conceptual relations among entrypoints, reflectors, and edge servers in Akamai's streaming CDN

replicating of a stream to a large number of edge clusters should the streaming event become extremely popular. The conceptual relation among entrypoints (sources), reflectors, and edge servers is shown in Fig. 16.11.

The use of reflectors also makes the content distribution more robust, because now there are multiple alternate paths between entrypoints and edge servers. If no single high-quality path is available between an entrypoint and an edge server, the system uses multiple link-disjoint paths that utilize different reflectors as intermediaries (see Fig. 16.11). Using the data forwarded along multiple paths, the edge servers can recover packet losses in individual paths, and forward the end-users the best combined results.

The Akamai's servers residing in more than 2,000 of the world's networks also monitor the Internet in real-time, gathering information about traffic, congestion, and trouble spots in the distribution network. A set of user agents will also continuously simulate users by repeatedly playing streams and testing their quality. A number of stream quality metrics can then be derived to reflect end-users' perception. These include the start-up time, the effective bandwidth to end-users, the *stream availability*, which measures how often a user can play streams without failures, as well as the frequency and duration of interruptions during playback. Akamai uses these information to optimize routes and replicate data dynamically to deliver streams, offering end-users high-quality of experiences.

16.3 Broadcast/Multicast Video-on-Demand

Both proxy caching and CDN explore the temporal and geographical locality of users' interests in media objects. Such locality can also be explored through broadcast or multicast services to deliver the same content simultaneously to a massive amount of concurrent users. It works well for live media streaming. For media-on-demand services, the users' requests are asynchronous and therefore one single broadcast/multicast channel cannot serve the requests arriving at different times, even if they are for the same audio/video. In this section, we will introduce scalable broadcast/multicast solutions for media-on-demand with such asynchronous requests.

Note that there are subtle differences between broadcast and multicast as described in the previous chapter: the former is to all the destinations and the latter is to a group of destinations only. While broadcast is possible in air, cable networks, or local area networks, it simply cannot be carried over the global Internet. Nevertheless, we do not distinguish them if the context is clear and refer to both as broadcast here.

16.3.1 Smart TV and Set-Top Box (STB)

Among all possible Media-on-Demand services, the most popular is likely to be subscription to video: over high-speed networks, customers can specify the movies or TV programs they want and the time they want to view them. This will realize *Interactive TV* (iTV) or *Smart TV* that supports a growing number of activities, such as

- TV (basic, subscription, pay-per-view)
- Video-on-Demand (VoD)
- Information services (news, weather, magazines, sports events, etc.)
- Interactive entertainment (Online games, etc.)
- E-commerce (online shopping, stock trading, etc.)
- Digital libraries and distance education (e-learning, etc.)

The key differences between a smart TV and the conventional cable TV are (1) A smart TV invites user interactions; hence the need for two-way traffic—downstream (content provider to user) and upstream (user to content provider), and (2) a smart TV is rich in information and multimedia services. With the penetration of *Digital Video Broadcasting* (DVB), the activities mentioned above all have emerged in today's *Multimedia Home Platform* (DVB-MHP).

To perform the above functions, a network-ready computer or a *Set-top Box* (STB) for a conventional TV set is required, which generally has the following components, as Fig. 16.12 shows:

- **Network interface and communication unit**, including a digital tuner, security devices, and a communication channel for basic navigation of Web and digital libraries as well as services and maintenance.
- **Processing unit**, including CPU, memory, and a special-purpose operating system for the STB.
- **Audio/video unit**, including audio and video decoders, Digital Signal Processor (DSP), buffers, and D/A converters.
- **Graphics unit**, supporting real-time graphics for animation and games.
- **Peripheral control unit**, including controllers for disks, audio and video I/O devices (e.g., digital video cameras), external memory card reader and writer, and so on.

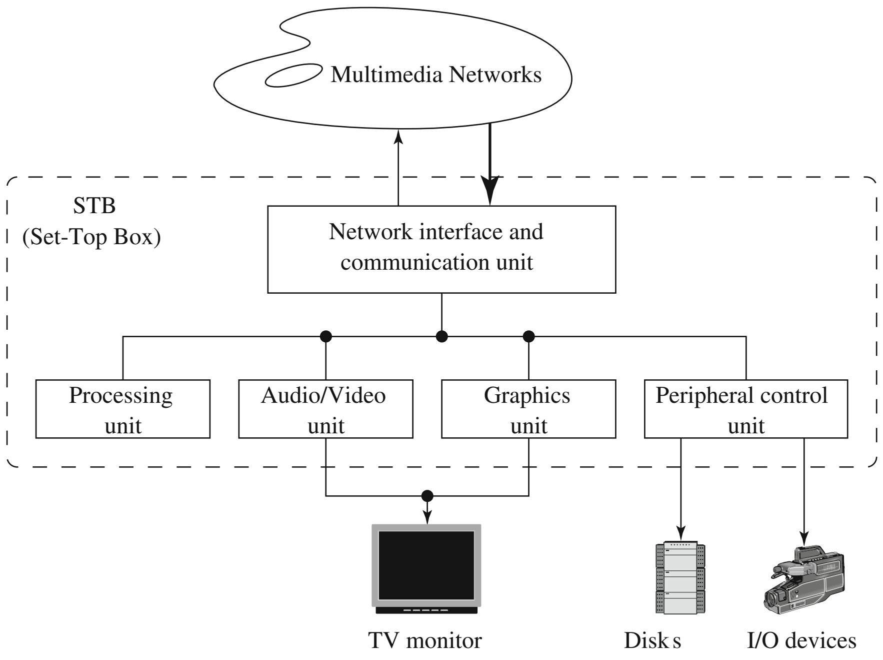

Fig. 16.12 General architecture of set-top box

16.3.2 Scalable Multicast/Broadcast VoD

Consider the Video-on-Demand service with smart TV users. Existing statistics suggest that most of the demands are usually concentrated on a few (10–20) popular movies or TV shows (e.g., new releases and top-ten movies/shows of the season). While one single multicast or broadcast channel cannot satisfy all the user requests arriving at different times, it is possible to smartly multicast or broadcast these videos, so that a number of clients can be put into the different groups following their requests [15].

One earlier solution is *Batching*, which, like sliding interval caching, serves clients arriving close together in time using a single broadcast. An important quality measure of such a broadcast VoD service is the latency. We define the *access time* as the upper bound between the time of requesting a video and the time of actually consuming it. Apparently, the access time with batching increases with an increasing amount of client request aggregation.

Given the potentially high bandwidth of today's broadband networks and the low cost of local storage, it is conceivable that the video can be fed to the client in a relatively shorter time than its playback duration. This leads to the development of a series of *periodical broadcast* VoD solutions.

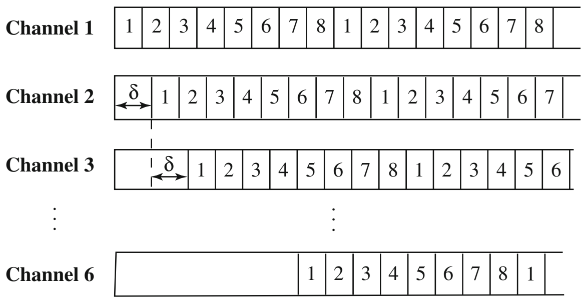

Fig. 16.13 Staggered broadcasting with $M = 8$ videos and $K = 6$ channels

Staggered Broadcasting

For simplicity, we assume all videos are encoded using constant-bit-rate (CBR), are of the same length L (measured in time units), and will be played sequentially from beginning to end without interruption. The available high bandwidth W is divided by the playback rate b to yield the bandwidth ratio B. The bandwidth of the server is usually divided up into K logical channels ($K \geq 1$).

Assuming the server broadcasts up to M videos ($M \geq 1$), all can be periodically broadcast on all these channels with the start-time of each video staggered. This is referred to as *Staggered broadcasting*. Figure 16.13 shows an example of staggered broadcasting in which $M = 8$ and $K = 6$.

For staggered broadcasting, if the division of the bandwidth is equal among all K logical channels, then the access time for any video is $\delta = \frac{M \cdot L}{B}$. Note that the access time is actually independent of the value of K. In other words, the access time will be reduced linearly with an increased network bandwidth.

Pyramid Broadcasting

To improve the staggered broadcasting, *Pyramid broadcasting* [16] divides a video into segments of increasing sizes. That is, $L_{i+1} = \alpha \cdot L_i$, where L_i is the size (length) of Segment S_i and $\alpha > 1$. Segment S_i will be periodically broadcast on Channel i. In other words, instead of staggering the videos on K channels, the segments are now staggered. Each channel is given the same bandwidth, and the larger segments are broadcast less frequently.

Since the available bandwidth is assumed to be significantly larger than the video playback rate b (i.e., $B >> 1$), it is argued that the client can be playing a smaller Segment S_i and simultaneously be receiving a larger Segment S_{i+1}.

To guarantee continuous (noninterrupted) playback, the necessary condition is

$$playback\_time(S_i) \geq access\_time(S_{i+1}) \tag{16.6}$$

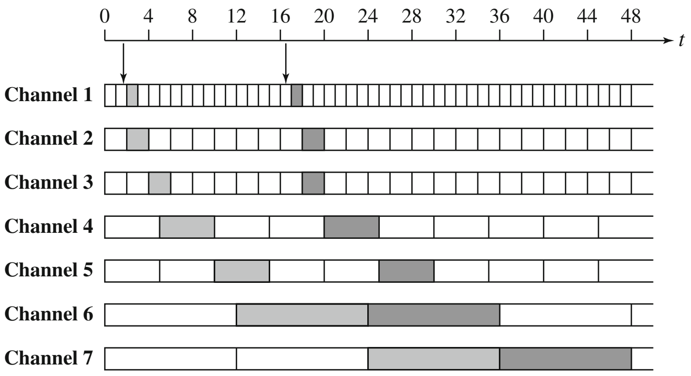

Fig. 16.14 Skyscraper broadcasting with seven segments

where the $playback\_time(S_i) = L_i$. Given the bandwidth allocated to each channel is $B/K \cdot b$, we have $access\_time(S_{i+1}) = \frac{L_{i+1} \cdot M}{B/K} = \frac{\alpha \cdot L_i \cdot M}{B/K}$, which yields

$$L_i \geq \frac{\alpha \cdot L_i \cdot M}{B/K} \tag{16.7}$$

Consequently,

$$\alpha \leq \frac{B}{M \cdot K} \tag{16.8}$$

The size of S_1 determines the access time for pyramid broadcasting. By default, we set $\alpha = \frac{B}{M \cdot K}$ to yield the shortest access time. The time drops exponentially with the increase in total bandwidth B, because α can be increased linearly.

A main drawback of the above scheme is the need for a large storage space on the client side, because the last two segments are typically 75–80 % of the video size. Instead of using a geometric series, *Skyscraper broadcasting* [17] uses {1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52, ...} as the series of segment sizes, to alleviate the demand on a large buffer.

Figure 16.14 shows an example of Skyscraper broadcasting with seven segments. As shown, two clients who made a request at time intervals (1, 2) and (16, 17), respectively, have their respective transmission schedules. At any given moment, no more than two segments need to be received.

Harmonic Broadcasting

Harmonic broadcasting [18] adopts a different strategy. The size of all segments remains constant, whereas the bandwidth of channel i is $B_i = b/i$, where b is the video's playback rate. In other words, the channel bandwidths follow the decreasing pattern $b, b/2, b/3, \ldots b/K$. The total bandwidth allocated for delivering the video

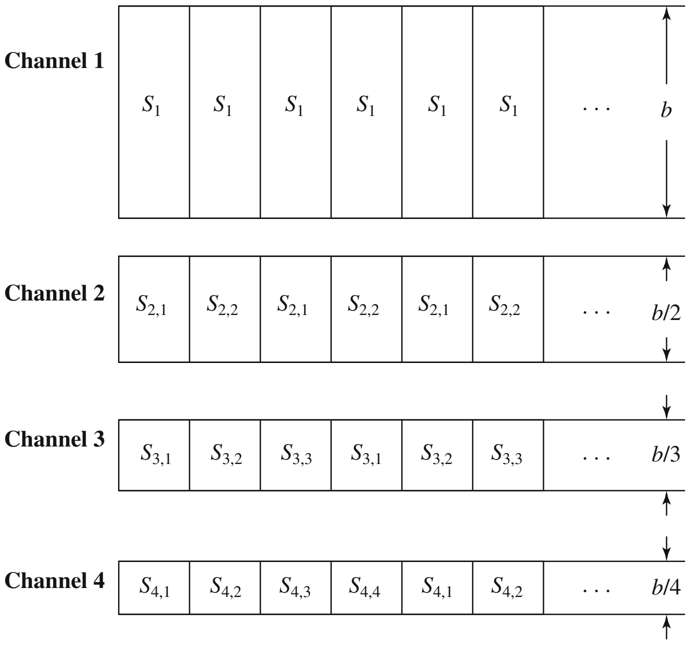
Channel 1
S_1 S_1 S_1 S_1 S_1 S_1 ...
b
Channel 2
$S_{2,1}$ $S_{2,2}$ $S_{2,1}$ $S_{2,2}$ $S_{2,1}$ $S_{2,2}$ ...
$b/2$
Channel 3
$S_{3,1}$ $S_{3,2}$ $S_{3,3}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$ ...
$b/3$
Channel 4
$S_{4,1}$ $S_{4,2}$ $S_{4,3}$ $S_{4,4}$ $S_{4,1}$ $S_{4,2}$ ...
$b/4$

Fig. 16.15 Harmonic broadcasting

is thus

$$B = \sum_{i=1}^{K} \frac{b}{i} = H_K \cdot b \qquad (16.9)$$

where K is the total number of segments, and $H_K = \sum_{i=1}^{K} \frac{1}{i}$ is the *harmonic number* of K.

Figure 16.15 shows an example of harmonic broadcasting. After requesting the video, the client is allowed to download and play the first occurrence of segment S_1 from channel 1. Meanwhile, the client will download all other segments from their respective channels.

Take S_2 as an example: it consists of two halves, S_{21} and S_{22}. Since bandwidth B_2 is only $b/2$, during the playback time of S_1, one-half of S_2 (say S_{21}) will be downloaded (prefetched). It takes the entire playback time of S_2 to download the other half (say S_{22}), just as S_2 is finishing playback. Similarly, by this time, two-thirds of S_3 has already been prefetched, and so the remaining third of S_3 can be downloaded just in time for playback from channel 3, which has a bandwidth of only $b/3$, and so on.

The advantage of harmonic broadcasting is that the Harmonic number grows slowly with K. For example, when $K = 30$, $H_K \approx 4$. If the video is 120 min long, this yields small segments—only 4 min (120/30) each. Hence, the access time for harmonic broadcasting is generally shorter than for Pyramid broadcasting, and the

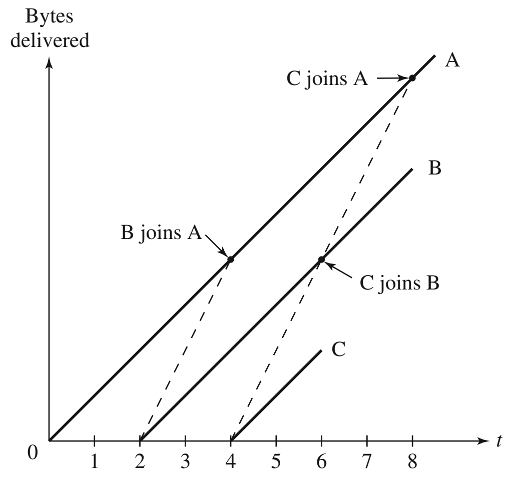

Fig. 16.16 Stream merging

demand on total bandwidth (in this case $4b$) is modest. Its required buffer size at the client side is 37 % of the entire video [18], which also compares favorably with the original pyramid broadcasting scheme.

However, the above Harmonic broadcasting scheme does not always work. For example, if the client starts to download at the second instance of S_1 in Fig. 16.15, then by the time it finishes S_1, only the second half of S_2—that is, S_{22}—is prefetched. The client will not be able to simultaneously download and play S_{21} from channel 2, since the available bandwidth is only half the playback rate.

An obvious fix to the above problem is to ask the client to delay the playback of S_1 by one slot, although it will double the access time.

Stream Merging

The above broadcast schemes are most effective when limited user interactions are expected—that is, once requested, clients will stay with the sequential access schedule and watch the video in its entirety.

Stream merging is more adaptive to dynamic user interactions, which is achieved by dynamically combining multicast sessions [19]. It still makes the assumption that the client's receiving bandwidth is higher than the video playback rate. In fact, it is common to assume that the receiving bandwidth is at least twice the playback rate, so that the client can receive two streams at the same time.

The server will deliver a video stream as soon as it receives the request from a client. Meanwhile, the client is also given access to a second stream of the same video, which was initiated earlier by another client. At a certain point, the first stream becomes unnecessary, because all its contents have been prefetched from the second stream. At this time, the first stream will merge with (or "join") the second.

As Fig. 16.16 shows, the "first stream" B starts at time $t = 2$. The solid line indicates the playback rate, and the dashed line indicates the receiving bandwidth, which is twice the playback rate. The client is allowed to prefetch from an earlier ("second") stream A, which was launched at $t = 0$. At $t = 4$, stream B joins A.

The technique of stream merging can be applied hierarchically [19]. As Fig. 16.16 shows, stream C, which started at $t = 4$, would join B at $t = 6$, which in turn joined A. The original stream B would have been obsolete after $t = 4$, since it joined A. In this case, it will have to be retained until $t = 6$, when C joins A.

A variation of stream merging is *piggybacking*, in which the playback rate of the streams is slightly and dynamically adjusted, to enable merging (piggybacking) of the streams.

16.4 Broadcast/Multicast for Heterogeneous Users

The Internet's intrinsic heterogeneity poses another challenge to multimedia broadcast/multicast. In traditional end-to-end adaptation schemes, the sender adjusts its transmission rate according to some feedback from its receiver. In a broadcast/multicast environment, this solution tends to be suboptimal because there is no single target rate for a group of heterogeneous users.

It is thus necessary to use *multi-rate multicast*, in which the users in a multicast session can receive media data at different rates according to their respective bandwidths or processing capabilities [20]. From the viewpoint of a media source, multi-rate streams can be produced via two methods. The first is *information replication*; that is, the sender generates replicated streams for the same media content but at different rates. The second is *information decomposition*. A commonly used decomposition scheme is *cumulative layering*, in which a raw media sequence is compressed into some nonoverlapping streams, or layers. The reconstructed quality is low if only one layer is decoded, but can be refined by decoding more layers. From media compression's perspective, replication and decomposition can be implemented through transcoding and scalable audio/video coding (see Sect. 10.5.3), respectively. The remaining question is the efficient transmission of multi-rate video streams to a large group of heterogeneous users using the Internet multicast infrastructure.

16.4.1 Stream Replication

Stream replication can be viewed as a trade-off between single-rate multicast and multiple point-to-point connections. Its feasibility is well justified in a typical multicast environment where the bandwidths of the receivers usually follow some clustered distribution. This is because they use standard access interfaces, for example, a 1.5 Mbps ADSL, a 15 Mbps VDSL, and a 100 Mbps fiber access, or they might share some bottleneck links and hence experience the same bottleneck bandwidth. As a result, a limited number of streams can be used to match these clusters to achieve reasonably good fairness. A representative of stream replication is the *Destination Set Grouping* (DSG) protocol [21]. In DSG, a source maintains a small number of media streams (say 3) for the same video content but with different rates. Each user subscribes to a stream that best matches its bandwidth. It periodically monitors the video reception level and reports this to the sender. A stream is then feedback-controlled within prescribed limits by its group of users. Specifically, if the percentile of congested users is above a certain threshold, the bandwidth of the stream should be reduced; if all the users experience no packet loss, its bandwidth can be increased. The user may also move across groups when its available bandwidth changes significantly.

Due to its simplicity, stream replication has been advocated in many commercial video streaming products, such as the SureStream mechanism provided by RelaNetworks' RealSystem. For YouTube, it originally offered videos at only one quality level, displayed at a resolution of 320×240 pixels using the Sorenson Spark codec (a variant of H.263), with mono MP3 audio. Later, 3GP format for mobile phones and high quality mode of 480×360 pixels were added. Today, YouTube videos are available in a range of quality levels, as shown in Table 16.2, matching the demands from highly heterogeneous Internet and mobile users. The former names of standard quality (SQ), high quality (HQ), and high definition (HD) have been replaced by numerical values representing the vertical resolution of the video. The default video stream is encoded in H.264/MPEG-4 AVC format, with stereo AAC audio.

16.4.2 Layered Multicast

For cumulative layered video (or scalable video), *Receiver-driven Layered Multicast* (RLM) [22] has been suggested, which takes advantage of the dynamic group concept in the IP multicast model. An RLM sender transmits each video layer over a separate multicast group. The number of layers as well as their rates is predetermined. Adaptation is performed only at the user's end by a probing-based scheme. Basically, a user periodically joins a higher layer's group to explore the available bandwidth. If packet loss exceeds a tolerable threshold after the join-experiment, i.e., congestion occurs, the user should leave the group; otherwise it will stay at the new subscription level.

Figure 16.17 shows an example of the layer joining/leaving behavior of a receiver. It started from layer 1 (the base layer), and then gradually joined enhancement layers 2, 3, and 4 as there was no congestion. After joining layer 4, however, congestion occured and it had to leave this highest layer. It waited for a while, seeing no congestion, and rejoined layer 4. This triggered congestion again, and the receiver had to leave again, observing the network condition and planning for the next join-experiment. Note that the waiting time for the next join-experiment is longer than that of the previous one; such an *exponential backoff* ensures that the receiver will not be too aggressive in joining new layers and cause frequent congestion.

One drawback of this probing-based scheme is that one user's join-experiments can induce packet losses experienced by others sharing the same bottleneck link. For example, the receiver in Fig. 16.17 dropped from layer 3 to 2 in a later time, which was not caused by its own join-experiment, but by others' experiment that

Table 16.2 Comparison of YouTube media encoding options

itag value	Default container	Video resolution	Video encoding	Video profile	Video bitrate (Mbit/s)	Audio encoding	Audio bitrate (kbit/s)
5	FLV	240p	Sorenson H.263	N/A	0.25	MP3	64
6	FLV	270p	Sorenson H.263	N/A	0.8	MP3	64
13	3GP	N/A	MPEG-4 Visual	N/A	0.5	AAC	N/A
17	3GP	144p	MPEG-4 Visual	Simple	0.05	AAC	24
18	MP4	270p/360p	H.264	Baseline	0.5	AAC	96
22	MP4	720p	H.264	High	2–2.9	AAC	192
34	FLV	360p	H.264	Main	0.5	AAC	128
35	FLV	480p	H.264	Main	0.8–1	AAC	128
36	3GP	240p	MPEG-4 Visual	Simple	0.17	AAC	38
37	MP4	1080p	H.264	High	3–5.9	AAC	192
38	MP4	3072p	H.264	High	3.5–5	AAC	192
43	WebM	360p	VP8	N/A	0.5	Vorbis	128
44	WebM	480p	VP8	N/A	1	Vorbis	128
45	WebM	720p	VP8	N/A	2	Vorbis	192
46	WebM	1080p	VP8	N/A	N/A	Vorbis	192
82	MP4	360p	H.264	3D	0.5	AAC	96
83	MP4	240p	H.264	3D	0.5	AAC	96
84	MP4	720p	H.264	3D	2-2.9	AAC	152
85	FLV	520p	H.264	3D	2-2.9	AAC	152
100	WebM	360p	VP8	3D	N/A	Vorbis	128
101	WebM	360p	VP8	3D	N/A	Vorbis	192
102	WebM	720p	VP8	3D	N/A	Vorbis	192
120	FLV	720p	AVC	Main@L3.1	2	AAC	128

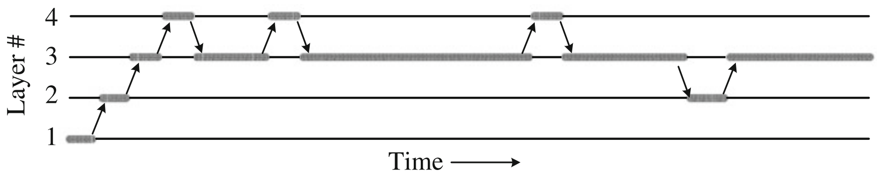

Fig. 16.17 An illustration of received-driven layered multicast

introduced congestion—in the current Internet that has no packet classification and prioritization, packet loss will occur in any layer when congestion happens, not necessarily the highest layer. These losses would occur frequently if all the users perform uncoordinated join-experiments. RLM incorporates a *shared learning* mechanism to solve this problem. With shared learning, a user notifies all other participants about a join-experiment to be conducted, and others will accordingly refrain from conducting their own experiments in the mean time. This avoids misinterpretation of congestion, but can reduce the scalability of RLM and significantly increase its convergence time.

The difficulties associated with coordinating join and leave attempts motivated the design of the *Receiver-driven Layered Congestion Control (RLC) protocol* [23]. RLC uses receiver-driven join/leave actions to mimic the behavior of TCP congestion control, i.e., Additive Increase and Multiplicative Decrease (AIMD). The join experiments among the receivers are synchronized and the rate of each layer is set to twice as much as the subsequent lower layer. This results in an exponential decrease of the bandwidth consumed in case of losses (like TCP).

Nevertheless, the objective of TCP differs significantly from the objective of video transmission protocols. Although this solution interacts better with TCP, it could experience the same saw-tooth behavior of TCP flows as we have seen before, resulting in unstable video quality. Therefore, rather than mimic the behavior of TCP, a more reasonable objective for video streaming should be to achieve a long-term fair share with TCP traffic with smoothly controlled rate, as the TCP-Friendly Rate Control (TFRC) protocol does. In the multicast case, each user can estimate the equivalent throughput of a TCP connection running over the same path from the sender, and performs join and leave actions according to this estimated bandwidth..

As discussed in Sect. 15.3.2, a TCP throughput model generally depends on the packet size, loss event rate, and round-trip time (RTT). The former two can be readily estimated by a receiver, but the estimation of RTT between the sender and a receiver requires feedback packets, which may cause the well-known *feedback implosion* problem in a large multicast session; that is, too many feedbacks from the large number of receivers overwhelm the sender. Smart lightweight feedback loops have been developed to address this issue. For example, the *Multicast Enhanced Loss-Delay based Adaptation* (MLDA) protocol [24] employs an open-loop RTT estimation method as a complement to the closed-loop (feedback-based) method. It tracks the one-way trip time from the sender to the receiver and transforms it to an estimate of the round-trip time. Link asymmetry can be compensated by low-frequency close-loop estimations.

16.5 Application-Layer Multicast

Today the scope and reach of IP multicast remain limited, and many ISPs simply block or disable IP multicast due to various security and economic concerns [25]. The idea of using the application layer for multicast data forwarding came a long time ago [26, 27]. Though both application-layer multicast and IP multicast require intermediate nodes in the network topology to support the replication of data packets, the implementation in the application layer has much less demanding on end-hosts, as compared to switches and routers in the Internet core [28].

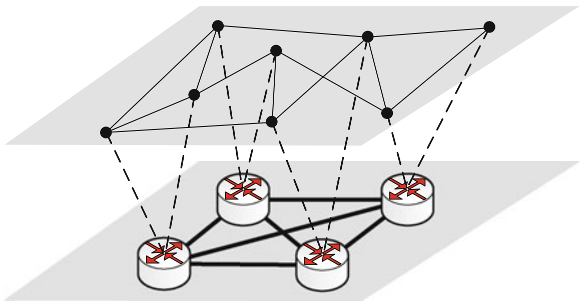

Fig. 16.18 An illustration of application-level overlay network

Figure 16.18 depicts an example of an application-layer multicast network that overlays the underlying Internet routers. When organizing the end-hosts into an overlay for disseminating video streams, a series of important criteria must be considered for overlay construction and maintenance [29].

- *Overlay efficiency*. The overlay constructed must be efficient both from the network and the application perspectives. For multicast video, high bandwidth and low latencies are simultaneously required. However, for applications that are real-time but not interactive, a start-up delay of a few seconds can be tolerated.
- *Scalability and load balancing*. Since multicast sessions can scale to tens of thousands of receivers, the overlay must scale to support such large sizes, and the overhead associated must be reasonable even at large scales.
- *Self-organizing*. The overlay construction should take place in a distributed fashion and must be robust to dynamic changes in group membership. Further, the overlay must adapt to long-term variations in Internet path characteristics (such as bandwidth and latency), while being resilient to inaccuracies. The system must be self-improving in that the overlay should incrementally evolve into a better structure as more information become available.
- *Node constraints*. Since the system relies on users contributing bandwidth, it is important to ensure that the total bandwidth a user is required to contribute does not exceed its inherent access bandwidth capacity. Also, a large fractions of users may stay behind NATs and firewalls—the connectivity restrictions posed by such users may severely limit the overlay capacity.

A number of proposals have emerged for application-layer multicast (also known as *overlay multicast*, *end-system multicast*, etc.) [29]. While they differ on a wide range of dimensions, earlier proposals were largely *push-based*, in which end-nodes are organized into structures (typically trees) for delivering data, with each data packet being disseminated using the same structure. Nodes on the structure have well-defined relationships, for example, the "parent-child" relationship in trees. Since all data packets follow this structure, it becomes critical to ensure the structure is optimized to offer good performance to all receivers. Furthermore, the structure must be maintained, as nodes join and leave the group at will—in particular, if a node crashes or otherwise stops performing adequately, all of its offspring in the tree will stop receiving packets, and the tree must be repaired. Finally, when constructing tree-based structures, loop avoidance is an important issue that must be addressed.

16.5.1 Representative: End-System Multicast (ESM)

The ESM system [30] employs a tree-based overlay protocol that is distributed, self-organizing, and performance-aware. The tree, rooted at the source, is optimized primarily for bandwidth, and secondarily for delay.

Group Management

Each ESM node maintains information about a small random subset of members, as well as information about the path from the source to itself. A new node joins the multicast session by contacting the source and retrieving a random list of members that are currently in the group. It then selects one of these members as its parent. Each node A also periodically picks one member (say B) at random, and sends B a subset of group members that A knows, along with the last timestamp it has heard for each member. When B receives a membership message, it updates its list of known members. Finally, members are deleted if their states have not been refreshed in a period.

Membership Dynamics

Dealing with graceful member leave is fairly straightforward: the member continues forwarding data for a short period, while its children look for new parents using the parent selection method described below. This serves to minimize disruptions to the overlay. The members also send periodic control packets to their children to indicate existence.

Performance-Aware Adaptation

Each node maintains the application-level throughput it is receiving in a recent time window. If its throughput is significantly below the source rate, then it selects a new parent. One key parameter here is the *detection time*, which indicates how long a node must stay with a poor performing parent before it switches to another parent. The ESM system employs a default detection time of 5 s. The choice of this value has been influenced by the fact that a congestion control protocol is running on the data path (TCP or TFRC), and switching to a new parent thus requires going through a slow-start phase, which may take 1–2 s to get the full source rate. The protocol adaptively tunes the detection time because the nodes may not be capable of receiving the full source rate, there may be few good and available parent choices in the system, or the nodes may experience intermittent network congestion on links close to them.

Parent Selection

When a node (say A) joins the multicast overlay, or needs to make a parent change, it probes a random subset of nodes it knows. The probing is biased toward members that have not been probed or have low delay. Each node B that responds provides the information about: (1) the throughput it is currently receiving, and delay from the source; (2) whether it is degree-saturated or not; and (3) whether it is a descendant of A. The probe also enables A to determine the round-trip time (RTT) to B. A waits for responses for a timeout period of 1 second, a large enough value of RTT, so as to maximize the number of responses received from members. From the responses A receives, it eliminates its descendants and the members that are saturated.

For each node B that has not been eliminated, A evaluates the performance (throughput and delay) it expects to receive if B were chosen as a parent. For example, the expected application throughput is the minimum of the throughput B is currently seeing and the available bandwidth of the path between B and A if the estimate is available. History of the past performance is maintained—if A has previously chosen B as parent, then it has an estimate of the bandwidth of the path between B and A. If the bandwidth to the nodes is not known, then A picks a parent based on delay. A identifies the node B that could best improve performance, and switches to the parent B either if the estimated application throughput is high enough for A to receive a higher quality stream, or if B maintains the same bandwidth level as A's current parent, but improves delay. The latter heuristic helps to increase tree efficiency by clustering nearby nodes.

16.5.2 Multi-tree Structure

The tree-based designs are perhaps the most natural approach. One concern with them is that the failure of nodes, particularly those close to the root may disrupt the data delivery to a large number of users, and potentially result in poor transient performance. Furthermore, a majority of nodes are leaves in the structure, and their outgoing bandwidth is not being utilized. More resilient structures, in particular, multi-tree [31,32], thus have been introduced.

In a multi-tree, the source encodes the stream into sub-streams and distributes each sub-stream along a particular overlay tree. The quality experienced by a receiver depends on the number of sub-streams that it receives. There are two key advantages of the multi-tree solution. First, the overall resiliency of the system is improved, as a node is not completely disrupted by the failure of an ancestor on a single tree. Second, the potential bandwidth of all nodes can be utilized, as long as each node is not a leaf in at least one tree.

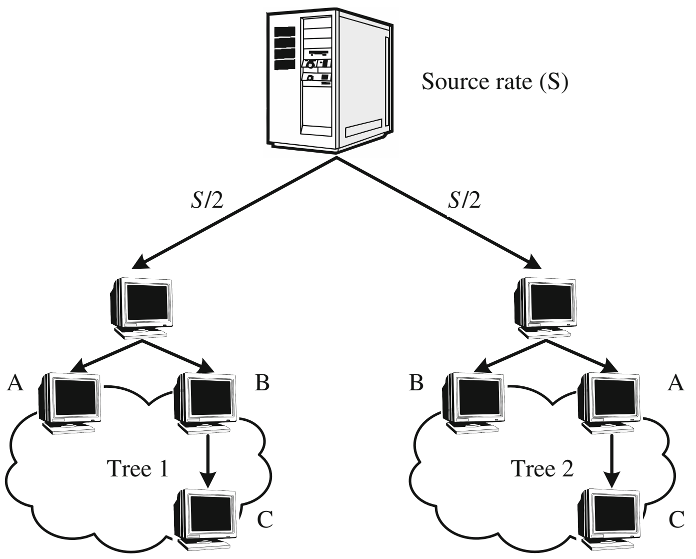

Fig. 16.19 A multi-tree application-layer multicast with two trees. Note that node A in Tree 1 and that in Tree 2 are physically the same nodes, so for node B or C

Figure 16.19 illustrates how the multicast content is delivered with a multi-tree approach using two trees. The source distributes a stream rate $S/2$ over each tree, where S is the source rate. C receives $S/2$ from the tree, with potentially different parents to reconstruct the original content. Nodes A and B each can contribute a bandwidth $S/2$, and allocate their bandwidth in Tree2 and Tree1, respectively. In a single-tree approach, it is hard to utilize the contributions from these nodes. It can be seen that Akamai's streaming CDN that we examined earlier also uses a multi-tree solution (with reflectors), despite the nodes there being dedicated replication servers.

16.6 Peer-to-Peer Video Streaming with Mesh Overlays

Peer-to-peer (P2P) further extends the application-layer multicast paradigm by taking advantage of the ability of participating end-hosts, or *peers*, in a multicast group to contribute their uplink bandwidth. It was first brought to spotlight by the advent of Napster (1998) and Gnutella (2001). Later, the design philosophy in the highly popular BitTorrent software has converged with academic solutions in application-layer multicast, and a new generation of data-driven peer-to-peer streaming protocols on random mesh topologies emerged [29].

Data-driven or mesh overlay designs sharply contrast with tree-based application-layer multicast in that they do not construct and maintain an explicit structure for delivering data. The underlying argument is that, rather than constantly repair a structure in a highly dynamic peer-to-peer environment, we can use the availability of data to guide the data flow. In comparison, the tree-based application-layer multicast adopts a more rigid design [33,34], in that the structure of each tree needs to be actively managed as peers join and leave the session.

A naive approach to distribute data without explicitly maintaining a structure is to use *gossip algorithms* [35]. In a typical gossip algorithm, a node sends a newly generated message to a set of randomly selected nodes; these nodes do similarly in the next round, and so do other nodes until the message is spread to all. The random choice of gossip targets achieves resilience to random failures and enables decentralized operation. However, gossip cannot be used directly for video content distribution because its random push may cause significant redundancy with the high-bandwidth video. Without an explicit structure support, start-up and transmission delays can be significant, too.

To handle this, mesh overlays adopts a *pull-based* technique for data dissemination. More explicitly, each node maintains a set of partners, and periodically exchange data availability information with the partners. The node may then retrieve the unavailable data from one or more partners, or supply the available data to partners. Redundancy is avoided, as the node pulls data only if it has not already possessed it. Since any data segment may be available at multiple partners, the overlay is robust to failures—departure of a node simply means its partners will use other partners to receive data segments. Finally, the randomized partnerships imply that the potential bandwidth available between the peers can be fully utilized. As a result, pull-based protocols are much simpler to design and more amenable to real-world implementations. It has the potential to scale with group size, as greater demand also generates more resources.

There are common issues existing in both peer-to-peer file sharing and video streaming; for example, pricing for uploading/downloading and copyright protecting. The key difference is the timing constraints that a streaming protocol must accommodate: if video segments do not arrive in time, they are not useful when it comes to the time of playing them back. Thus, an important component of the data-driven overlay is a scheduling algorithm, which strives to schedule the segments that must be downloaded from various partners to meet the playback deadlines.

16.6.1 Representative: CoolStreaming

CoolStreaming [36] is the first large-scale data-driven peer-to-peer systems that was deployed in the real world for video streaming. Other successful companies such as PPLive, PPStream, and UUSee also adopted mesh-based pull techniques to deliver live or on-demand media content to millions of users.

Figure 16.20 depicts the system diagram of a CoolStreaming node, which consists of three key modules: (1) a membership manager, which helps the node maintain a partial view of other overlay nodes; (2) a partnership manager, which establishes and maintains the partnership with other known nodes; (3) a scheduler, which schedules the transmission of video data. For each segment of a video stream, a CoolStreaming node can be either a receiver or a supplier, or both, depending dynamically on this

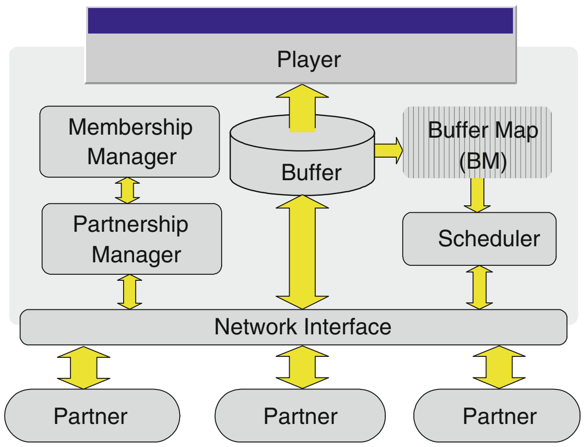

Fig. 16.20 A generic system diagram for a CoolStreaming node

segment's availability information, which is periodically exchanged between the node and its partners. An exception is the video source, which, as the *origin node*, is always a supplier. It could be a dedicated video server, or simply an overlay node that has a live video program to distribute.

Membership and Partner Management

Each CoolStreaming node has a unique identifier, such as its IP address, and maintains a membership cache (*mCache*) containing a partial list of the identifiers for active nodes in the overlay. In a basic node joining algorithm, a newly joined node first contacts the origin node, which randomly selects a *deputy node* from its mCache and redirects the new node to the deputy. The new node can then obtain a list of partner candidates from the deputy, and contacts these candidates to establish its partners in the overlay.

This process is generally viable because the origin node persists during the lifetime of streaming and its identifier/address is universally known. The redirection enables more uniform partner selections for newly joined nodes, and greatly minimized the origin node's load.

A key practical issue here is how to create and update the mCache. To accommodate overlay dynamics, each node periodically generates a membership message to announce its existence; each message is a 4-tuple $<seq\_num, id, num\_partner, time\_to\_live>$, where *seq_num* is the sequence number of the message, *id* is the node's identifier, *num_partner* is its current total number of partners, and *time_to_live* records the remaining valid time of the message. Upon receiving a message of a new *seq_num*, the node updates its mCache entry for node id, or create the entry if not existing. The entry is a 5-tuple $<seq\_num, id, num\_partner, time\_to\_live,$

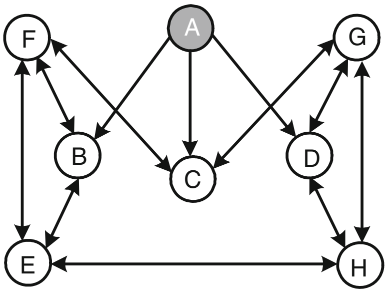

Fig. 16.21 An illustration of partnerships in CoolStreaming with A being the origin node. The partnership is bi-directional, except for node A, which serves as a supplier only. For example, node F is a partner of nodes B, C, and E, and node E is a partner of nodes B, F, and H

$last\_update\_time>$, where the first four components are copied from the received membership message, and the fifth is the local time of the last update for the entry.

The following two events also trigger updates of an mCache entry: (1) the membership message is forwarded to other nodes through gossiping; and (2) the node serves as a deputy and the entry is included in the partner candidate list. In either case, *time_to_live* is decreased by $current\_local\_time - last\_update\_time$. If the new value is less than or equal to zero, the entry will be removed; otherwise, *num_partner* will be increased by one in the deputy case.

Buffer Map Representation and Exchange

An example of partnership in an overlay is shown in Fig. 16.21. As said, neither the partnerships nor the data transmission directions are fixed. More explicitly, a video stream is divided into segments of a uniform length, and the availability of the segments in the buffer of a node can be represented by a *Buffer Map* (BM). Each node continuously exchanges its BM with the partners, and then schedules which segment is to be fetched from which partner accordingly.

Timely and continuous segment delivery is crucial to media streaming, but not to file download. In BitTorrent, the download phases of the peers are not synchronized, and new segments from anywhere in the file are acceptable. In CoolStreaming, the playback progress of the peers is roughly synchronized, and any segment downloaded after its playback time will be useless. A *sliding window* thus represents the active buffer portion, as shown in Fig. 16.22.

Suggested by experimental results, CoolStreaming adopts a sliding window of 120 segments, each of 1-second video. A BM thus consists of a bitstring of 120 bits, each indicating the availability of a corresponding segment. The sequence number of the first segment in the sliding window is recorded by another two bytes, which can be rolled back for extra long video programs (>24 h).

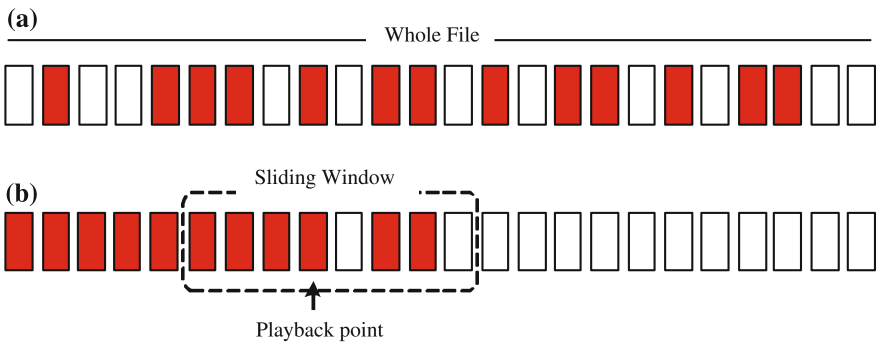

Fig. 16.22 Buffer snapshots of BitTorrent (**a**) and CoolStreaming (**b**), where shaded segments are available in the buffer

Scheduling Algorithm

Given the BMs of a node and its partners, a schedule is then generated for fetching the expected segments from the partners. For a homogeneous and static network, a simple round-robin scheduler may work well, but for a dynamic and heterogeneous network, a more intelligent scheduler is necessary. Specifically, the scheduling algorithm strikes to meet two constraints: the playback deadline for each segment, and the heterogeneous streaming bandwidth from the partners. If the first constraint cannot be satisfied, then the number of segments missing deadlines should be kept minimum, so as to maintain a continuous playback. This problem is a variation of the *Parallel machine scheduling*, which is known to be NP-hard. It is thus not easy to find an optimal solution, particularly considering that the algorithm must quickly adapt to the highly dynamic network conditions. CoolStreaming resorts to a simple heuristic of fast response time.

The heuristic first calculates the number of potential suppliers for each segment (i.e., the partners containing or to contain the segment in their buffers). Since a segment with less potential suppliers is more difficult to meet the deadline constraints, the algorithm determines the supplier of each segment starting from those with only one potential supplier, then those with two, and so forth. Among the multiple potential suppliers, the one with the highest bandwidth and enough available time is selected.

As an example, consider node F in Fig. 16.21, which has partners B, C, and E. Assume that a buffer map contains only four segments and they are 1000, 0010, 0011, and 0101 for nodes F, B, C, and E, respectively. That is, node F has only segment 1 available in its local buffer, but 2, 3, and 4 missing. Among the three missing segments, segment 2 has only one supplier (node E) and segments 3 and 4 each have two suppliers (B, C for 3, and C, E for 4). As such, node F will schedule to fetch segment 2 first, from node E. It will then fetch segments 3 and 4. For segment 3, between the two potential suppliers B and C, the one with the higher bandwidth will be scheduled. The same strategy applies to segment 4.

Given a schedule, the segments to be fetched from the same supplier are marked in a BM-like bit sequence, which is sent to that supplier, and these segments are then delivered in order through the TCP-Friendly Rate Control (TFRC) protocol. There have been many enhancements to the basic scheduling algorithm in CoolStreaming, and existing studies have also suggested that the use of advanced *network coding* can possibly enable optimal scheduling [37,38].

Failure Recovery and Partnership Refinement

A CoolStreaming node can depart gracefully or accidentally due to an unexpected failure. In either case, the departure can be easily detected after an idle time of TFRC or BM exchange, and an affected node can quickly react through rescheduling using the BM information about the remaining partners. Besides this built-in recovery mechanism, CoolStreaming also lets each node periodically establish new partnerships with nodes randomly selected from its local membership list. This operation serves two purposes: First, it helps each node maintain a stable number of partners in the presence of node departures; Second, it helps each node explore partners of better quality, e.g., those constantly having a higher upload bandwidth and more available segments.

16.6.2 Hybrid Tree and Mesh Overlay

We have seen solutions of both tree-based overlays (e.g., ESM) and mesh-based peer-to-peer overlays (e.g., CoolStreamig). A tree is the most efficient structure for multicasting, but has to face the inherent instability, maintenance overhead, and bandwidth under-utilization. The selling point for the mesh is its simplicity and robustness. Its control communication overhead however cannot be overlooked. Intuitively, since the sliding window at a peer advances itself over time, the buffer maps need to be exchanged as frequently as needed, which may lead to a substantial amount of overhead.

A natural question is therefore whether we can combine them to realize a hybrid overlay that is both efficient and robust. The combination can be achieved in different dimensions. An example is Chunkyspread [39], which splits a stream into distinct slices and transmits over separate but not necessarily disjoint trees. The participating nodes also form a neighboring graph, and the degree in the graph is proportional to its desired transmission load. This hybrid design simplifies the tree construction and maintenance, but largely retains its efficiency and achieves fine-grained control.

Another solution is a more explicit tree-backbone-based approach [40]. Existing trace studies have shown evidence that most of the data segments delivered through a data-pull mesh overlay essentially follow a specific tree structure or a small set of trees. The similarity of the trees, defined as the fraction of their common links, can be as high as 70 %. The overlay performance thus closely depends on the set of common internal nodes and their organization. This suggests that, while maintaining

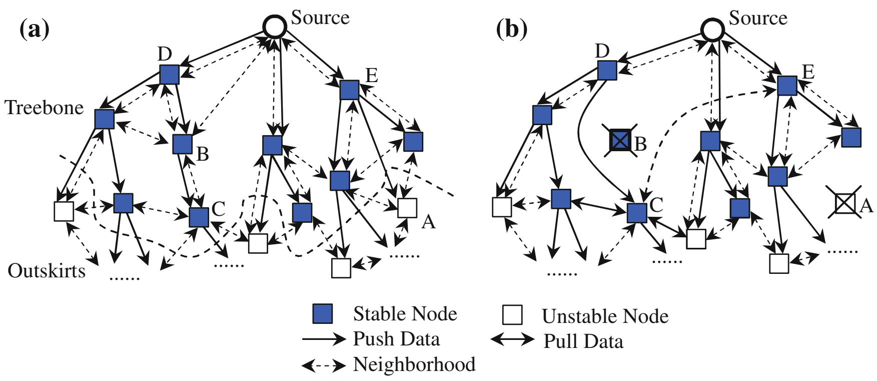

Fig. 16.23 An illustration of a hybrid tree and mesh design. **a** The hybrid overlay; **b** Node B leaves. To make the figures cleaner, we omit many unstable nodes in the outskirts

a prior topology for all the nodes is costly, optimizing the organization for a core subset is worth consideration. In particular, if such a subset consists of the stable nodes, we can expect high efficiency with low overhead and delay simultaneously. Figure 16.23 shows an example of a tree backbone that consists mainly of stable nodes; other nonstable nodes are attached to the backbone as outskirts. Most of the streaming data will be pushed through the backbone, and eventually reaches the outskirts. To improve the resilience and efficiency of the backbone, the nodes can be further organized into a mesh overlay, as indicated by the dotted lines in the figure. In this auxiliary mesh, a node will not actively schedule to pull data blocks from neighbors as in the pure mesh; rather, a pull will be invoked only if there is data outage from the backbone.

When an unstable node, such as node A fails or leaves, it will not affect the data pushed along the backbone. On the other hand, the backbone nodes are stable and seldom leave; even if a leave happens, the impact can be mitigated with the help from the mesh overlay. For example, consider the leave of node B, shown in Fig. 16.23b. While node C is affected, it can easily pull the missing data from its mesh neighbors before it re-attaches to the backbone.

16.7 HTTP-Based Media Streaming

Although peer-to-peer has proven to be highly scalable in video delivery, there are critical issues for peer-to-peer system deployment by content providers: (1) *Ease-of-use*. In peer-to-peer streaming, the users are usually required to install customized client software or plugins to be able to cache the video contents watched and exchange the contents with others—this is not user-friendly given that today's users are so familiar with using web browsers to consume Internet contents directly; (2) *Copyright*. In a peer-to-peer streaming system that arose from illegal file sharing,

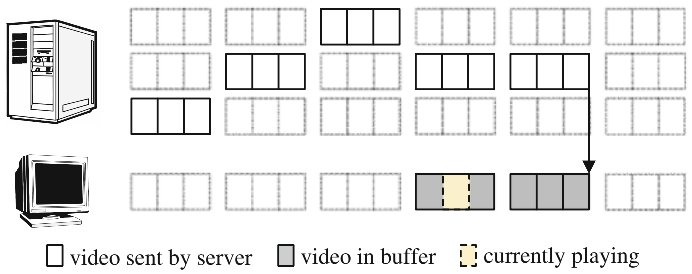

Fig. 16.24 An illustration of HTTP Streaming

the users exchange contents with each other autonomously—it is very difficult for the content providers to control the copyright in the video streaming.

Peer-to-peer also relies on peers' contribution to the system. In the real world, there are many *free riders* who do not want to contribute their resources. Even if the peers are willing to contribute, the upload bandwidth of many peers is often constrained given the asymmetricity in such access networks as ADSL. Moreover, the data exchanged between peers need to traverse NAT in both directions through open ports, which is known to be difficult as we have discussed in the previous chapter. They are exposed to security threats, too, and are often blocked by firewalls.

16.7.1 HTTP for Streaming

As the underlying protocol for web transactions, the *Hyper Text Transfer Protocol* (HTTP) is generally firewall-friendly because almost all firewalls are configured to support connections for web transactions. HTTP server resources are also widely available commodity and therefore supporting HTTP streaming for massive audience can be cost-effective using the existing web infrastructure.

HTTP was not initially designed for streaming applications. It does not provide signaling mechanisms for interactive streaming control, and its underlying transport protocol, TCP, was not originally designed for continuous media, either. The key to support streaming with HTTP is to break the overall media stream into a sequence of small HTTP-based file downloads; each download includes one short chunk of an overall potentially unbounded stream. Using a series of the HTTP's `GET` commands, a user can progressively download the small files while playing those already being downloaded. Any damaged or delayed block will have limited impact, thus ensuring continuous playback. This process is illustrated in Fig. 16.24.

HTTP does not maintain session states on the server. Therefore, provisioning a large number of clients does not impose significant cost on server resources. This is quite different from RTP/RTCP/RTSP-based streaming that have to maintain per-session states. Yet each client can keep a record of its playback progress, and the progressive download also allows a client to seek to a specific position in the media stream by downloading the corresponding file, or more precisely, performing an

HTTP's `byte range` request for the file, realizing similar functionalities offered by RTSP.

HTTP streaming has been implemented in commercial products. Today, representative online video providers, including Netflix, YouTube, Hulu, are using HTTP to stream their videos to the users. Besides the superiorities we have mentioned earlier, HTTP streaming is benefiting from the rapidly expanding capacity and dropping pricing of today's CDNs. Specifically, the emergence of such ad-based business models have boosted the importance of *video quality*, as there is a crucial interplay between video quality and user engagement. As a consequence, content providers, with an objective of generating more revenue, are caring more about the visual quality and stability of their streaming service to maximize user engagement. Since today's CDNs are mainly designed and optimized to serve the web contents [41], HTTP streaming is capable of using the existing infrastructure to deliver high quality media with low cost, better stability and security, and simpler interfaces.

16.7.2 Dynamic Adaptive Streaming Over HTTP (DASH)

Different HTTP-based implementations use different manifest and segment formats and hence, to receive the content from each server, a device must support its corresponding proprietary client protocol. There is a demand for standardization so that different devices can inter-operate. The heterogeneous networks and devices also require the media streaming to be dynamical and adaptive. To this end, the *Dynamic Adaptive Streaming over HTTP* (DASH) standard has been developed by the MPEG group. Work on DASH started in 2010; it became an international standard in November 2011, and was officially published as ISO/IEC 23009-1:2012 in April 2012 [42].

DASH defines a set of implementation protocols across the servers, clients, and description files. In DASH, a video stream is encoded and divided into multiple segments, including initialization segments that contain the required information for initializing the media decoder, and media segments that contain the following data: (1) the media data, and (2) the stream access point, indicating where the client decoder can play. Subsegments can also be used such that a user can download them using the HTTP's `partial GETS` command, which includes a `Range` header field, requesting that only part of the entity be transferred.

A *Media Presentation Description* (MPD) describes the relation of the segments and how they form a video presentation, which facilitates segment fetching for continuous playback. A sample MPD file is shown below.

```
<MPD>
<BaseURL>http://www.baseurl_1.com</BaseURL>//Destination URL(s)
<Period>
   <AdaptationSet>//Video Set
      <Representation bandwidth="4190760" height="1080" width="1920">
         <SegmentInfo>... </SegmentInfo>//Quality_1
```

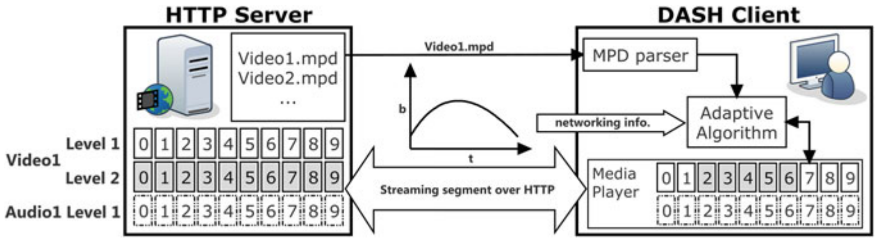

Fig. 16.25 A scenario of DASH-based streaming

```
        </Representation>
        <Representation bandwidth="2073921" height="720" width="1280">
            <SegmentInfo>... </SegmentInfo>//Quality_2
        </Representation>
    </AdaptationSet>
    <AdaptationSet>//Audio Set
        <Representation bandwidth="127234" sampleRate="44100">
            <SegmentInfo>... </SegmentInfo>//Quality_1
        </Representation>
    </AdaptationSet>
</Period>
</MPD>
```

All BaseURLs are shown at the beginning of the MPD file. A client can analyze this part to acquire the destination URLs and then pull streaming data from the servers. In this simple MPD file, there is only one period, which consists of video and audio adaptation sets. There are two video sequences with different resolutions and bitrates, allowing the client to choose based on local and networking conditions. There is only one soundtrack (of 44.1 KHz sampling rate) for the video stream. In a more complex scenario, the audio set may also have several soundtracks with different languages and bitrates.

Figure 16.25 illustrates the DASH-based streaming with the two video levels and one audio level. The client can use an adaption algorithm to choose the appropriate audio and video levels. During playback, the adaption algorithm will monitor the local and network status, so as to achieve the best possible QoS, e.g., request lower quality segments when the network bandwidth is low, and higher quality if enough bandwidth is available.

As a new and open standard, there are many issues worthy of further investigations in its development:

- *Rate adaptation components*. DASH only defines the segmentation and the file description, and leaves rate adaptation for either the client or the server to implement. The client may utilize multi-path and multi-server approaches to receive video segments [43]. Such receiver-driven approaches, customized in the application layer,

Table 16.3 Typical Server/Client configurations for HTTP streaming

Type	Server	Client
Adobe adaptive streaming	Flash media server	Flash media player
Apple HTTP Live streaming	Generic HTTP servers	QuickTime/iOS player
Microsoft Live smooth streaming	Internet information services (IIS)	Silverlight player

are highly flexible and scalable. On the other hand, the servers could also be able to adaptively change the bitrate for its clients, based on the perception of the client download speed and server load.

- *Rate adaptation strategies*. Rate adaptation strategies determine how different versions of segments are received by clients, to achieve such objectives as streaming stability, fairness, and high quality. It has been shown that the existing implementations in popular commercial products were either too aggressive or too conservative, and better strategies that jointly consider efficiency, fairness, and stability are to be developed [44].

DASH is also codec agnostic, though its prime container is MPEG-4. It allows seamless adoption of the coming improved HEVC video codec (i.e., H.265). As compatible clients become available, it promises to be widely adopted in a wide range of devices. In Table 16.3, we list the server and client configurations of popular HTTP/DASH implementations from industry.

16.8 Exercises

1. Consider prefix caching with a total proxy cache size of S and N videos of user access probabilities $r_1, r_2, \ldots, r_N$, respectively. Assume that the *utility* of caching for each video is given by function $U(l_i)$ where l_i is the length of the cached prefix for video i. Develop an algorithm to optimize the total utility of the proxy. You may start from the simple case where $U(l_i) = l_i \cdot r_i$.
2. For the optimal work-ahead smoothing technique, how would you algorithmically determine at which point to change the planned transmission rate? What is the transmission rate?
3. Consider again the optimal work-ahead smoothing technique. It was suggested that instead of using every video frame, only frames at the beginning of statistically different compression video segments can be considered. How would you modify the algorithm (or video information) to support that?
4. Discuss the similarities and differences between proxy caching and CDN. Is it beneficial to utilize both of them in a system?

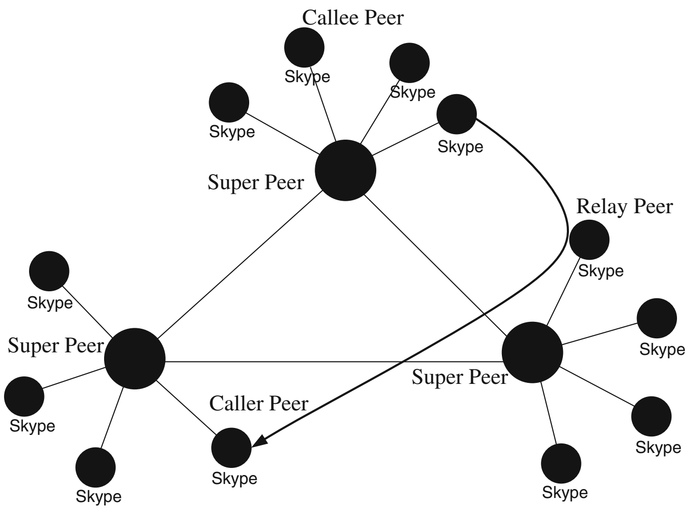

Fig. 16.26 An illustration of the Skype peer-to-peer network

5. Discuss the similarities and differences between a CDN for web content distribution and that for multimedia streaming. What is the role of *reflectors* in Akamai's streaming CDN?
6. For Staggered broadcasting, if the division of the bandwidth is equal among all K logical channels ($K \geq 1$), show that the access time is independent of the value of K.
7. Given the available bandwidth of each user, $b_1, b_2, \ldots, b_N$, in a multicast session of N users, and the number of replicated video streams, M, develop a solution to allocate the bitrate to each stream, B_i, $i = 1, 2, \ldots, M$, so that the average *inter-receiver fairness* is maximized. Here, the inter-receiver fairness for user j is defined as $\max \frac{B_k}{b_j}$ where $B_k \leq b_j$, $k = 1, 2, \ldots, M$, i.e., the video stream of the highest rate that user j can receive.
8. In Receiver-driven Layer Multicast (RLM), why is shared learning necessary? If IntServ or DiffServ is deployed in the network, will RLM still need shared learning?
9. In a multicast scenario, too many receivers sending feedback to the sender can cause a *feedback implosion* that would block the sender. Suggest two methods to avoid the implosion and yet provide reasonably useful feedback information to the sender.
10. To achieve TCP-Friend Rate Control (TFRC), the Round-Trip Time (RTT) between the sender and the receiver must be estimated (see Sect. 15.3.2). In the unicast TFRC, the sender generally estimates the RTT and hence the TCP-friendly throughput, and accordingly controls the sending rate. In a multicast scenario, who should take care of this and how? Explain your answer.
11. In this question, we explore the scalability of peer-to-peer, as compared to client/server. We assume that there is one server and N users. The upload

bandwidth of the server is S bps, and the download bandwidth of user i is D_i bps, $i = 1, 2, \ldots, N$. There is a file of size M bits to be distributed from the server to all the users.

(a) Consider the client/server architecture. Each user is a now a client that is directly served by the server. Calculate the time to distribute the file to all the users.
(b) Now consider the peer-to-peer architecture. Each user is now peer, who can either download directly from the server or from other peers. Assume that the upload bandwidth of user i for serving other peers is U_i bps, $i = 1, 2, \ldots, N$. Calculate the time to distribute the file to all the users.
(c) Using the results, explain in what conditions will peer-to-peer scale better (with more users) than client/server. Are these conditions naturally satisfied in the Internet?

12. Discuss the similarities and differences between peer-to-peer file sharing and peer-to-peer live streaming. How will such differences affect the implementation of a peer-to-peer living streaming? And how will they affect the calculation in the previous question.
13. Consider tree-based and mesh-based overlays for peer-to-peer streaming.

(a) Discuss the pros and cons of each of them.
(b) Why is the pull operation used in mesh-based overlays?
(c) Propose a solution (other than those introduced in the book) to combine them toward a hybrid overlay. You may target different application scenarios, e.g., for minimizing delay or for multi-channel TV broadcast where some users may frequently change channels.

14. Consider Skype, a popular Voice-over-IP (VoIP) application using peer-to-peer communication. The peers in Skype are organized into a hierarchical overlay network, with the peers being classified as *super peers* or *ordinary peers*, as illustrated in Fig. 16.26. When two Skype users (caller and callee) need to set up a call, both the ordinary peers and the super peers can serve as relays.

(a) Skype generally uses UDP for audio streams but TCP for control messages. What kind of control messages are necessary for Skype peers, and why is TCP used?
(b) Explain the benefit of distinguishing super peers and ordinary peers.
(c) Besides one-to-one calls, Skype also supports multi-party conferences. How many copies of audio streams would be delivered in an N user conference, if each user needs to send its copy of stream to all others?
(d) Note that this number can be high. Skype reduces it by asking each user to send its stream to the conference initiator, who will combine the streams into one stream and then forward to each of the other users. How many streams are to be forwarded in the whole conference now? Discuss the pros and cons of this solution, and suggest improvements.

15. One important reason that HTTP was not traditionally used for media streaming is that the underlying TCP has highly fluctuated transmission rate (the *saw-tooth*

behavior), and during severe congestion or channel errors, it may persistently block the data pipe. Explain how DASH addresses these problems. Also discuss other supports for streaming that are missing in the basic HTTP but addressed in DASH.

References

1. B. Li, Z. Wang, J. Liu, W. Zhu, Two decades of internet video streaming: a retrospective view. ACM Trans. Multimedia Comput. Commun. Appl. **9**(1s):1–33 (2013)
2. D. Wu, Y.T. Hou, W. Zhu, Y.-Q. Zhang, J.M. Peha, Streaming video over the internet: approaches and directions. IEEE Trans. Circuits Syst. Video Technol. **11**(3):282–300 (2001)
3. D. Wu, Y.T. Hou, Y.-Q. Zhang, Transporting real-time video over the internet: challenges and approaches. Proc. IEEE **88**(12):1855–1877 (2000)
4. S. Chen, B. Shen, S. Wee, X. Zhang, Designs of high quality streaming proxy systems. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, vol. 3, pp. 1512–1521 (2004)
5. X. Jianliang, J. Liu, B. Li, X. Jia, Caching and prefetching for web content distribution. IEEE Comput. Sci. Eng. **6**(4), 54–59 (2004)
6. J. Liu, X. Jianliang, Proxy caching for media streaming over the internet. IEEE Commun. Mag. **42**(8), 88–94 (2004)
7. R. Tewari, H.M. Vin, A. Dany, Y.D. Sitaramy, Resource-based caching for web servers, in *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, pp. 191–204 (1998)
8. S. Chen, B. Shen, Y. Yan, S. Basu, X. Zhang, SRB: shared running buffers in proxy to exploit memory locality of multiple streaming media sessions, in *Proceedings of 24th International Conference on Distributed Computing Systems*, pp. 787–794 (2004)
9. S. Sen, J. Rexford, D. Towsley, Proxy prefix caching for multimedia streams, in *Proceedings of IEEE INFOCOM'99*, vol. 3, pp. 1310–1319 (1999)
10. H. Fabmi, M. Latif, S. Sedigh-Ali, A. Ghafoor, P. Liu, L.H. Hsu, Proxy servers for scalable interactive video support. Computer **34**(9), 54–60 (2001)
11. K.-L. Wu, P.S. Yu, J.L. Wolf, Segment-based proxy caching of multimedia streams, in *Proceedings of the 10th International Conference on World Wide Web*, ACM, pp. 36–44 (2001)
12. Z.-L. Zhang, Y. Wang, D.H.C. Du, D. Shu, Video staging: a proxy-server-based approach to end-to-end video delivery over wide-area networks. IEEE/ACM Trans. Netw. **8**(4), 429–442 (2000)
13. J.D. Salehi, Z.L. Zhang, J.F. Kurose, D. Towsley, Supporting stored video: reducing rate variability and end-to-end resource requirements through optimal smoothing. ACM SIGMETRICS **24**(1):222–231 (1996)
14. E. Nygren, R.K. Sitaraman, J. Sun, The Akamai network: a platform for high-performance internet applications. SIGOPS Oper. Syst. Rev. **44**, 2–19 (2010)
15. A. Hu, Video-on-demand broadcasting protocols: a compreshensive study, in *Proceedings of IEEE INFOCOM* (2001)
16. S. Viswanathan, T. Imielinski, Pyramid broadcasting for video on demand service. IEEE Conference on Multimedia Computing and Networking, pp. 66–77 (1995)
17. K.A. Hua, S. Sheu, Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems, in *Proceedings of ACM SIGCOMM*, pp. 89–100 (1997)
18. L. Juhn, L. Tseng, Harmonic broadcasting for video-on-demand service. IEEE Trans. Broadcast **43**(3), 268–271 (1997)

19. D. Eager, M. Vernon, J. Zahorjan, Minimizing bandwidth requirements for on-demand data delivery. IEEE Trans. Knowl. Data Eng. **13**(5), 742–757 (2001)
20. B. Li, J. Liu, Multirate video multicast over the internet: an overview. IEEE Netw. **17**(1), 24–29 (2003)
21. S.Y. Cheung, M.H. Ammar, Using destination set grouping to improve the performance of window-controlled multipoint connections, in *Proceedinf of Fourth International Conference on Computer Communications and Networks* (1995), pp. 388–395
22. S. McCanne, V. Jacobson, M. Vetterli, Receiver-driven layered multicast, In *Conference Proceeding on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '96* (1996), pp. 117–130
23. L. Vicisano, J. Crowcroft, L. Rizzo, Tcp-like congestion control for layered multicast data transfer, in *Proceedings of IEEE INFOCOM'98*, vol. 3 (1998), pp. 996–1003
24. D. Sisalem, A. Wolisz, Mlda: a tcp-friendly congestion control framework for heterogeneous multicast environments, in *2000 Eighth International Workshop on Quality of Service, IWQOS*, pp. 65–74, 2000
25. C. Diot, B.N. Levine, B. Lyles, H. Kassem, D. Balensiefen, Deployment issues for the ip multicast service and architecture. IEEE Netw. **14**(1), 78–88 (2000)
26. S. Sheu, K.A. Hua, W. Tavanapong, Chaining: a generalized batching technique for video-on-demand systems, in *Proceeding of IEEE International Conference on Multimedia Computing and Systems* (1997)
27. Y.-H. Chu, S.G. Rao, H. Zhang, A case for end system multicast, in *Proceeding of ACM SIGMETRICS* (2000)
28. M. Hosseini, D.T. Ahmed, S. Shirmohammadi, N.D. Georganas, A survey of application-layer multicast protocols. IEEE Commun. Surv. Tutorials **9**(3), 58–74 (2007)
29. J. Liu, S.G. Rao, B. Li, H. Zhang, Opportunities and challenges of peer-to-peer internet video broadcast. Proc. of the IEEE **96**(1), 11–24 (2008)
30. Y.-H. Chu, S.G. Rao, H. Zhang, A case for end system multicast. IEEE J. Sel. A. Commun. **20**(8), 1456–1471 (2006)
31. V.N. Padmanabhan, H.J. Wang, P.A. Chou, K. Sripanidkulchai, Distributing streaming media content using cooperative networking, in *Proceeding of the 12th International workshop on Network and Operating Systems Support for Sigital Audio and Video*, NOSSDAV '02, ACM, New York (2000), pp. 177–186
32. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, Splitstream: high-bandwidth multicast in cooperative environments, in *Proceeding of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM, New York (2003), pp. 298–313
33. V. Venkataraman, K. Yoshida, P. Francis, Chunkyspread: heterogeneous unstructured tree-based peer-to-peer multicast, in *Proceeding of 5th International Workshop on Peer-to-Peer Systems (IPTPS)* (2006), pp. 2–11
34. N. Magharei, R. Rejaie, Y. Guo, Mesh or multiple-tree: a comparative study of live P2P streaming approaaches, in *Proceeding of IEEE INFOCOM* (2007)
35. P.T. Eugster, R. Guerraoui, A.M. Kermarrec, L. Massoulié, From epidemics to distributed computing. IEEE Comput. **37**, 60–67 (2004)
36. X. Zhang, J. Liu, B. Li, T.P. Yum, Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming, in *Proceedings of IEEE INFOCOM*, vol. 3 (2005), pp. 2102–2111
37. Z. Liu, C. Wu, B. Li, S. Zhao, UUSee: large-scale operational on-demand streaming with random network coding, in *Proceeding of IEEE INFOCOM* (2010)
38. M. Wang, B. Li, R^2: random push with random network coding in live peer-to-peer streaming. IEEE J. Sel. Areas Commun. **25**, 1678–1694 (2007)
39. V. Venkataraman, K. Yoshida, P. Francis, Chunkyspread: heterogeneous unstructured tree-based peer-to-peer multicast, in *Proceeding of the 14th IEEE International Conference on Network Protocols, ICNP '06* (2006), pp 2–11

40. F. Wang, Y. Xiong, J. Liu, Mtreebone: a collaborative tree-mesh overlay network for multicast video streaming. IEEE Trans. Parallel Distrib. Syst. **21**(3), 379–392 (2010)
41. G. Pallis, A. Vakali, Insight and perspectives for content delivery networks. Commun. ACM **49**(1), 101–106 (2006)
42. ISO/IEC JTC 1/SC 29/WG 11 (MPEG). Dynamic adaptive streaming over HTTP (2010)
43. S. Gouache, G. Bichot, A. Bsila, C. Howson, Distributed and adaptive HTTP streaming, in *Proceeding of IEEE ICME* (2011)
44. S. Akhshabi, A.C. Begen, C. Dovrolis, An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP, in *Proceeding of ACM MMSys* (2011)