RTP Congestion Control and RTCWEB Application Feedback

Ted Hardie
June 2012

One of the principal recent drivers of work on RTP congestion control has been the RTCWEB work, which will enable web-based platforms to exchange real-time media without plugins. In that context, it may be useful to describe how RTP congestion control mechanisms might feedback to an RTCWEB application, so that it could prioritize flows, lower offered load on the network, or change FEC parameters to handle loss.

Unfortunately, the first and most basic fact here is that things are much worse than they may seem. The RTCWEB application architecture sees the browser as a platform for downloaded web applications, so any feedback which will result in updated application preferences must pass from the browser platform to the Javascript-based application. The likelihood that the javascript application programmer will understand the full set of RTP feedback mechanisms is unfortunately small, and the likelihood that they will be fully conversant in the details of alterations which could be made in media parameters or FEC to handle changing network conditions is equally small. The threading model of Javascript in browsers also risks blocking that may affect (or be affected by) updates from congestion control routines, which makes timely response problematic. Further, the downloaded application is fundamentally untrusted by the overall system, so certain aspects of its behavior will be constrained so as to protect against attacks by malicious javascript. While this is useful from a security perspective, it means that there is no single locus of control for managing any feedback. A javascript application may be able to express preferences or constraints, but it is the browser (or its equivalent in a mobile platform) which must perform. Lastly, the basic mode of operation presumes that there may be flows coming to an application from multiple endpoints. This means that some indications of congestion related to a single flow will be early indications of a problem that will impact them all, while others will be restricted to a network segment used by only a single flow. Generalized responses that impact all flows may thus needlessly impact flows which are actually performing well.

It is tempting in that context to give up entirely on feedback mechanisms, in favor of flow-based fairness algorithms that do not require application intervention. Unfortunately, it is ultimately only the application which can determine how best to manage offered load. As an example, imagine a poker application which has both a main video stream for the player currently a betting, a series of thumbnail streams for other players at the virtual table, and data streams which go to the other players and a house server. If a local network gets congested, the application might decide that it is best to request most of its peers to lower the fidelity of the thumbnail streams to static or near-static pictures, while maintaining good fidelity for the current player and a reasonable fidelity for the next up in table order. A flow-based fairness algorithm, in contrast, might over-allocate resources to the thumbnail flows while letting the betting player's video degrade to the point where all the other players miss the "tell". Note as well that the constant shift in which player bets makes it critical which period is used to calculate fair use of resources. If the calculation period does not match the application's timing, the result may both have a larger than appropriate impact on game play and have a less than appropriate impact on congestion.

This hints, however, at a potential approach that may fit both the RTCWEB application context and the need for effective congestion control. Rather than attempting to feedback data on congestion and request application adjustments, the overall system should be constructed with the assumption that the Javascript application feeds forward data about priorities and

preferences to the underlying browser platform (or its mobile equivalent). At application initiation it would signal the priorities and constraints or preferences related to each of the tracks; it could then adjust those in messages to the browser as the application conditions changed.

To return to our poker example, when the betting player submitted the new bet, the receipt of that information at the application would enable it to tell the browser which tracks were now the highest priority. In essence, I believe we want the application to use something it knows ("Play has shifted so that *this* is now the betting player") to adjust the priorities consistently, rather than have it hear something about the network ("This flow got the following RTCP feedback") and react. Reactive processing may both be difficult for the javascript application and not as timely as needed for changing conditions. On an overprovisioned network, these updates from the Javascript application to the browser may be un-needed, and applications without flow priority requirements may choose to run without providing this additional data. When present, though, the browser can combine the data from the application (and any other applications running within its context) with its knowledge of network conditions in order to manage the priorities, offered load, and FEC parameters.

Moving this up one layer of indirection leaves, however, one very basic problem unsolved: what sort of fairness does the browser (or mobile platform) use as a model to combine the application preference with its knowledge of network conditions? Essentially, I believe the browser should treat the application's priority and constraint information as if it were QoS instructions for a small network within the browser itself. The priority and constraints expressed from Javascript result in QoS-like buckets inside the browser's "internal network"; if it gets ECN messages or other indications of congestion, it reacts by managing flows in much the same way a router with those QoS buckets would. These QoS instructions also influence how it then offers load to the successor network off the browser, but obviously its control is less there; an enterprise network may assign very different priorities to poker traffic than the browser would.

With an application like our poker example, we already have small real time packets (audio), large real time packets (video), web traffic, and peer-to-peer to peer game play. While I have no data to support this intuition, my personal guess is that per-bit congestion models may be a better fit for these applications and networks than per-packet congestion models. The work in Byte and Packet Congestion notification highlights some of the impact may have on fairness, as does RFC 4828's description of TRFC-SP. Further research is certainly warranted on this point. In particular, if there is some way for the browser to infer the likely type of congestion from signals sent by other participants, its internal equivalent to AQM may manage the overall set of flows much more effectively.