Network Working Group
Request for Comments: 3124
Category: Standards Track

H. Balakrishnan
MIT LCS
S. Seshan
CMU

June 2001

The Congestion Manager

Диспетчер контроля перегрузок

Статус документа

В этом документе содержится проект стандартного протокола, предложенного сообществу Internet. Документ служит приглашением к дискуссии в целях развития и совершенствования протокола. Текущее состояние стандартизации протокола вы можете узнать из документа “Internet Official Protocol Standards” (STD 1). Документ может распространяться без ограничений.

Авторские права

Аннотация

В документе описан менеджер контроля перегрузок (Congestion Manager или CM) - модуль конечной системы, который:

(i) позволяет предотвращать и контролировать перегрузки для нескольких одновременных потоков от источника к одному получателю, имеющих одинаковые свойства в части перегрузок;

(ii) обеспечивает приложениям лёгкую адаптацию к перегрузкам в сети.

1. Соглашения и термины

Ключевые слова **необходимо** (MUST), **недопустимо** (MUST NOT), **требуется** (REQUIRED), **нужно** (SHALL), **не следует** (SHALL NOT), **следует** (SHOULD), **не нужно** (SHOULD NOT), **рекомендуется** (RECOMMENDED), **возможно** (MAY), **необязательно** (OPTIONAL) в данном документе интерпретируются в соответствии с RFC-2119 [Bradner97].

STREAM - поток
Группа пакетов с одинаковыми IP-адресами отправителя и получателя, типом обслуживания IP (TOS), транспортным протоколом, и номерами транспортных портов у отправителя и получателя.

MACROFLOW - макропоток
Группа поддерживающих CM потоков с одинаковыми алгоритмами контроля перегрузок и планирования, а также общими данными о состоянии перегрузки. В настоящее время потоки с разными получателями относятся к разным макропотокам и потоки для одного получателя также **могут** относиться к разными макропотокам. При использовании диспетчера контроля перегрузок потоки с идентичным поведением при перегрузке, использующие один механизм контроля перегрузок **следует** относить к одному макропотоку.

APPLICATION - приложение
Любой программный модуль, использующий CM, включая пользовательские приложения, такие как Web-серверы и серверы аудио-видео, а также встроенные в ядро протоколы, такие как TCP [Postel81], использующие CM для контроля перегрузок.

WELL-BEHAVED APPLICATION - приложение с корректным поведением
Приложение, передающее данные только с разрешения CM и точно учитывающее все отправленные получателю данные, информируя о них CM через CM API.

PATH MAXIMUM TRANSMISSION UNIT (PMTU) - максимальный блок передаваемых по пути данных
Размер наибольшего пакета, который можно передать без фрагментирования по пути к получателю, включая все данные и заголовки, кроме заголовка IP.

CONGESTION WINDOW (cwnd) - окно перегрузки
Переменная состояния CM, управляющая количеством остающихся в сети данных между отправителем и получателем.

OUTSTANDING WINDOW (ownd) - окно оставшегося
Число байтов, которые были переданы отправителем, но для них ещё нет сведений о приёме получателем или потере в сети.

INITIAL WINDOW (IW) - начальное окно
Размер окна перегрузки у отправителя в начале макропотока.

DATA TYPE SYNTAX - синтаксис типа данных
Используется обозначение u64 для 64-битовых чисел без знака, u32 для 32-битовых чисел без знака, u16 для 16-битовых чисел без знака, u8 для 8-битовых чисел без знака, i32 для 32-битовых чисел со знаком, i16 для 16-битовых чисел со знаком и float для значений IEEE с плавающей точкой. Тип void указывает, что от вызова не ожидается возврата значения. Для указателей применяется обозначение * в соответствии с синтаксисом языка C.
Следует подчеркнуть, что все описанные в документе функции API являются «абстрактными» вызовами и соответствующие CM реализации могут отличаться в деталях.

2. Введение

Описываемая в этом документе модель встраивает контроль перегрузок во все приложения и транспортные протоколы. CM поддерживает параметры контроля перегрузок (доступная пропускная способность по потокам и в

целом, время кругового обхода по потокам и т. п.) и обеспечивает интерфейс API, позволяющий приложениям узнавать характеристики сети, передавать сведения в CM, совместно использовать данные о перегрузках и планировать передачу данных. Документ ориентирован на приложения и транспортные протоколы с их собственными независимыми данными о порядковых номерах для пакетов и байтов и не требует изменения в стеке протоколов получателя. Однако принимающее приложение должно обеспечивать обратную связь с передающей стороной, сообщая той о принятых и потерянных пакетах. Предполагается, что отправитель будет использовать CM API для обновления состояния CM. В документе не рассматриваются сети с резервированием или дифференцированными услугами.

CM - это модуль конечной системы, позволяющий стабильно предотвращать перегрузки для набора одновременных потоков. Он также позволяет приложениям легко приспосабливать свою передачу к преобладающим в сети условиям. Модуль поддерживает параметры контроля перегрузки (доступная пропускная способность по потокам и в целом, время кругового обхода по потокам и т. п.) и обеспечивает интерфейс API, позволяющий приложениям узнавать характеристики сети, передавать сведения в CM, совместно использовать данные о перегрузках и планировать передачу данных. При использовании CM передача данных, подлежащих контролю перегрузки, должна выполняться только с явного согласия CM через этот API для обеспечения надлежащего поведения при перегрузках.

Системы **могут** использовать CM и в этом случае они **должны** следовать данной спецификации.

Этот документ относится к сетям, в которых выполняются указанные ниже условия.

1. Приложения ведут себя корректно, имеют свою независимую нумерацию байтов и пакетов и применяют CM API для обновления внутреннего состояния в CM.
2. Сети работают по принципу best-effort без дискриминации и резервирования. В частности, не рассматриваются ситуации, где разные потоки между одной парой проходят по путям с разными характеристиками.

Модель CM можно расширить для поддержки приложений, не обеспечивающих обратной связи, и сетей с дифференцированными услугами. Эти расширения будут описаны в последующих документах. Разработка CM преследовала две основных цели, указанные ниже.

(i) Эффективное мультиплексирование. В Internet все чаще возникают ситуации, когда отправители индивидуальных (unicast) данных (например, Web-серверы) передают получателям разнотипные данные - от доставки потокового содержимого без гарантий в реальном масштабе времени до гарантированной доставки Web-страниц и приложений (applet). В результате множество логически различающихся потоков проходят по одному пути между отправителем и получателем. Для сохранения стабильности Internet каждый из таких потоков должен включать протоколы управления, безопасно проверяющие наличие свободной пропускной способности и реагирующие на перегрузку. К сожалению эти одновременные потоки обычно конкурируют в части доступа к ресурсам сети вместо их эффективного совместного использования. Кроме того, они не узнают один от другого состояния сети. Даже при реализации в каждом потоке независимого контроля перегрузки (например, в группе соединений TCP, реализующих алгоритмы [Jacobson88, Allman99]), совокупность потоков обычно будет более агрессивной при перегрузке, нежели одно соединение TCP со стандартным для TCP контролем и предотвращением перегрузок [Balakrishnan98].

(ii) Адаптация приложений к перегрузкам. Все чаще популярные потоковые приложения в реальном масштабе времени работают по протоколу UDP, применяя свой транспортный протокол пользовательского уровня для обеспечения высокой производительности приложений, но в большинстве случаев без должной адаптации и реагирования на перегрузки в сети. За счёт реализации стабильного алгоритма управления и предоставления адаптационного интерфейса API в CM обеспечивается адаптация приложений к перегрузкам с учётом текущего состояния сети.

Модель CM основана на недавних работах по совместному использованию блока управления TCP [Touch97], интегрированному контролю перегрузок в TCP (TCP-Int) [Balakrishnan98] и сессиям TCP [Padmanabhan98]. В [Touch97] отстаивается идея совместного использования состояния в блоке управления TCP для краткосрочного повышения производительности транспорта и описано совместное использование блока группой соединений TCP. В работах [Balakrishnan98], [Padmanabhan98], [Eggert00] описано несколько экспериментов по количественной оценке преимуществ совместного использования сведений о перегрузке, включая повышение стабильности при перегрузке и лучшее восстановление потерь. Интеграция восстановления потерь в одновременных соединениях существенно повышает производительность, поскольку потери в соединении можно обнаружить по получению и подтверждению более поздних данных в другом соединении. Модель CM расширяет эти идеи двумя важными способами: (i) контроль перегрузок в потоках, не относящихся к TCP, которые получают все большее распространение и зачастую не включают должного контроля перегрузок, и (ii) наличие API для приложений, позволяющего адаптировать их передачи к текущим условиям в сети. Расширенное обсуждение мотивов разработки и архитектуры CM, API и алгоритмов приведено в работе [Balakrishnan99], а описание реализации и влияния на производительность - в [Andersen00].

Разработанная архитектура протокола на конечном узле показана на рисунке 1. CM помогает обеспечить стабильность сети, реализуя стабильные алгоритмы предотвращения и контроля перегрузок, которые дружественны к TCP [Mahdavi98], на основе алгоритмов, описанных в [Allman99]. Однако в модели не предпринимается попыток обеспечить корректное поведение при перегрузке всех приложений (но не исключается наличие средств применения правил на хосте, выполняющем эту задачу). Хотя применение правил на конечном хосте может включать использование CM, сеть должна быть защищена от компрометации CM и средств применения правил на конечном хосте, а для этого требуется оборудования маршрутизатора [Floyd99a]. В данном документе этот вопрос не рассматривается.

Ключевыми компонентами модели CM являются (i) API, (ii) контроллер перегрузок и (iii) планировщик. Включение API (частично) обусловлено требованиями кадрирования на прикладном уровне (application-level framing или ALF) [Clark90], как описано в разделе 3. CM (раздел 4) включает контроллер перегрузок (параграф 4.1) и планировщик для организации передачи данных одновременных потоков в одном макропотоке (параграф 4.2). Контроллер перегрузок регулирует суммарную скорость передачи от отправителя к получателю на основе оценки перегрузок в сети, получая обратную связь о предшествующих передачах от самих приложений через API. Планировщик распределяет доступную пропускную способность между потоками в каждом макропотоке и уведомляет приложения о возможности передачи данных. В этом документе рассматриваются приложения с корректным поведением, а в будущем будет описан протокол и форматы заголовков для работы с приложениями, не обеспечивающими обратной связи с CM.

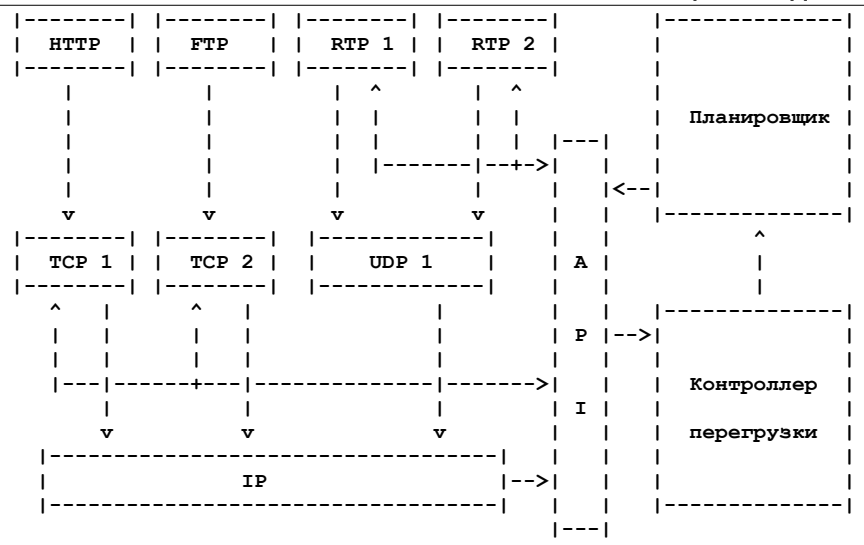

Рисунок 1.

3. CM API

По традиции IETF не рассматривает интерфейсы прикладных программ (Application Programming Interface или API) как стандарты. Однако считается важным собрать требования к CM API и алгоритму CM в одном согласованном документе. В последующих параграфах, посвящённых CM API, используются термины **должно**, **следует** и т. п., но эти термины применяются в контексте CM API. Раздел не относится к реализациям контроля перегрузок в целом и связан лишь с реализациями, предоставляющими CM API.

Используя CM API, потоки могут определять свою долю доступной пропускной способности, запрашивать и планировать передачу данных, информировать CM об успешных передачах и получать от CM оценку изменения пропускной способности. Таким образом, CM освобождает приложения от необходимости хранить сведения о состоянии перегрузки и доступной на пути пропускной способности.

Представленные ниже прототипы используют стандартные соглашения языка C. Следует подчеркнуть, что функции API являются абстрактными вызовами и соответствующие этому документу реализации CM могут отличаться в деталях при условии обеспечения эквивалентной функциональности.

При создании приложением нового потока некоторая информация передаётся в CM через вызов cm_open(stream_info). В настоящее время stream_info включает (i) IP-адрес источника, (ii) порт источника, (iii) IP-адрес получателя, (iv) порт получателя и (v) номер протокола IP.

3.1 Поддержка состояний

Open

Все приложения перед использованием CM API **должны** вызывать функцию cm_open(stream_info), которая возвращает дескриптор cm_streamid, используемый приложением при всех последующих вызовах CM API для этого потока. Возврат cm_streamid = -1 говорит об отказе cm_open() и невозможности использовать CM.
Во всех последующих вызовах CM для потока используется значение cm_streamid, возвращённое cm_open().

Close

При завершении потока приложению **следует** вызывать функцию cm_close(cm_streamid) для информирования CM о завершении потока.

Размер пакетов

Вызов cm_mtu(cm_streamid) возвращает значение PMTU на пути между отправителем и получателем. Внутри эти сведения **следует** получать через определение MTU для пути [Mogul90]. Значение **можно** задать статически при отсутствии механизма для его определения.

3.2 Передача данных

CM принимает два типа адаптивных отправителей, позволяя приложениям динамически менять своё содержимое в зависимости от преобладающих в сети условий и поддерживая приложения на основе кадрирования прикладного уровня (ALF).

1. Передача на основе обратного вызова (callback). API для такой передачи предоставляет потоку самостоятельно выбирать, что передавать в данный момент. Для этого CM не буферизует данных, а предоставляет потоку возможность адаптироваться к неожиданным изменениям в сети в последний момент. Это позволяет потоку «выталкивать» и перепаковывать данные, узнав о любом изменении скорости, что было бы сложно сделать при буферизации данных. В CM должен быть реализован вызов cm_request(i32 cm_streamid) для потоков, желающих передавать данные таким способом. Через некоторое время, в зависимости от скорости, CM **должен** выполнить обратный вызов с помощью cmapp_send(), что позволяет потоку передать до PMTU байтов. API с обратными вызовами рекомендуется для потоков на основе ALF. Следует отметить, что cm_request() не принимает в качестве аргумента числа байтов или блоков размером MTU, каждый вызов cm_request() является неявным запросом на отправку до PMTU байтов. CM **может** предоставлять дополнительный интерфейс cm_request(int k). Обратный вызов cmapp_send для такого запроса даёт право на передачу до k сегментов размером PMTU. В параграфе 3.3 рассматривается продолжительность действия такого права, а в параграфе 4.2 описано планирование запросов и выполнение обратных вызовов.

2. Синхронная передача. Упомянутый выше API на основе обратных вызовов позволяет использовать класс потоков ALF, являющихся «асинхронными». Асинхронный отправитель не передаёт данные периодически на основе часов, а делает это по событиям, таким как чтение файла или получение кадров. Существует много

потков, являющихся «синхронными» отправителями и передающих данные по своим внутренним таймерам (например, отправители звукового потока с постоянной скоростью выборки). Хотя обратные вызовы CM можно настроить на периодические прерывания для таких передатчиков, цикл передачи таких приложений меньше пострадает при использовании своей периодичности на основе внутренних таймеров. Кроме того, выражение потоком периодичности и гранулярности обратных вызовов усложнит CM API. Поэтому CM должен экспортировать интерфейс API, позволяющий информировать потоки об изменении скорости с помощью обратного вызова cmapp_update(u64 newrate, u32 srtt, u32 rttdev), где newrate указывает новую скорость (бит/сек), srtt - текущее сглаженное время кругового обхода (мксек), rttdev - сглаженное линейное отклонение при оценке времени кругового обхода, рассчитанное по алгоритму TCP [Paxson00]. Значение newrate сообщает рассчитанную мгновенную скорость, например, по отношению cwnd к srtt, поделённому на долю пропускной способности, выделенную потоку.

В ответ поток **должен** изменить размер пакетов или интервал их передачи в соответстви с разрешённой скоростью (не более). Отметим, что CM не является частью пути фактической передачи данных.

Во избежание ненужных обратных вызовов cmapp_update(), которые приложение просто будет игнорировать, CM **должен** поддерживать функцию cm_thresh(float rate_downthresh, float rate_upthresh, float rtt_downthresh, float rtt_upthresh), которую поток может использовать на любом этапе своего существования. В ответ диспетчеру CM **следует** использовать обратный вызов только при снижении скорости не менее, чем в (rate_downthresh * lastrate) раз или росте более, чем в (rate_upthresh * lastrate) раз, где lastrate - последняя скорость, сообщённая потоку, или при изменении времени кругового обхода в соответствии с заданными предварительно порогами. Эти сведения служат рекомендациями CM в части вызова cmapp_update() даже при невыполнении указанных условий.

В CM **должна** быть реализована функция cm_query(i32 cm_streamid, u64* rate, u32* srtt, u32* rttdev), позволяющая приложению запросить текущее состояние CM. При этом параметр rate указывает текущую оценку скорости (бит/сек), srtt - текущую сглаженную оценку времени кругового обхода (мксек), а rttdev - среднее линейное отклонение. Если у CM нет действительных оценок для макропотока, в параметрах rate, srtt, rttdev указываются отрицательные значения.

Отметим, что поток может использовать одновременно оба указанных выше API передачи. В частности, знание установившейся скорости полезно как для асинхронных потоков, так и для синхронных. Например, асинхронный Web-сервер, распространяющий изображения по протоколу TCP, может использовать cmapp_send() для планирования передачи и cmapp_update() для решения вопроса о выдаче изображений с высоким или низким разрешением. Описанная в параграфе 5.1.1 реализация TCP с использованием CM показывает преимущество API обратного вызова cm_request() для TCP.

Читатель может заметить, что базовый CM API не включает интрефейс для буферизованной передачи с контролем перегрузок. Это сделано намеренно, поскольку этот режим можно организовать с помощью примитива на основе обратного вызова. В параграфе 5.1.2 описана реализация сокетов UDP с контролем перегрузки, использующих CM API.

3.3 Уведомления от приложений

При получении потоком обратной связи от приёмников он **должен** вызвать функцию cm_update(i32 cm_streamid, u32 nrecd, u32 nlost, u8 lossmode, i32 rtt) для передачи CM таких сведений, как потери из-за перегрузки, успешный приём, тип потерь (там-аут, ECN[1] [Ramakrishnan99] и т. п.) и выборки времени кругового обхода. Параметр nrecd указывает число успешно полученных приёмником байтов с момента последнего вызова cm_update call, а nlost[2] - число потерянных за тот же интервал времени байтов. Значение rtt указывает время кругового обхода измеренное в процессе передачи указанных байтов. При отсутствии у приложения действительной выборки rtt следует указывать rtt = -1. Параметр lossmode указывает способ обнаружения потерь и **должен** указываться вектором битов, соответствующих CM_NO_FEEDBACK, CM_LOSS_FEEDBACK, CM_EXPLICIT_CONGESTION и CM_NO_CONGESTION. CM_NO_FEEDBACK говорит, что приложение не получило обратной связи от приёмника для остающихся данных и сообщает об этом CM. Например, при возникновании тайм-аута TCP это значение уведомит об этом CM. CM_LOSS_FEEDBACK указывает, что приложение столкнулось с потерями, которые, на его взгляд, связаны с перегрузкой но потеряны были не все остающиеся данные. Например, потеря сегмента TCP, обнаруженная по дубликату (селективного) подтверждения или иным методом, относится к этой категории. CM_EXPLICIT_CONGESTION указывает, что получатель отразил (echo) явное уведомление о перегрузке. CM_NO_CONGESTION говорит об отсутствии связанных с перегрузкой потерь. Отметим, что при потерях, не связанных с перегрузкой на других каналах (путях) приложению **следует** информировать CM о потерях установкой бита CM_NO_CONGESTION.

Функция cm_notify(i32 cm_streamid, u32 nsent) **должна** вызываться при передаче данных с хоста (например, в выходной процедуре IP) для информирования CM об отправке nsent байтов в данный поток. Это позволяет CM обновить свою оценку числа остающихся байтов для макропотока и потока.

Право cmapp_send(), предоставленное CM приложению, действительно лишь в течение времени, равного большему из значений интервала кругового обхода и зависящего от реализации порога, переданного как аргумент обратного вызова cmapp_send(). Приложению **недопустимо** передавать данные на основе вызова cmapp_send() по истечении этого времени. Если же приложение решит не передавать данных после этого вызова, ему **следует** вызвать cm_notify(stream_info, 0), чтобы позволить CM разрешать передачу данных другим потокам этого макропотока. Контроллер перегрузок CM **должен** быть устойчив к приложениям, забывающим корректно вызывать cm_notify(stream_info, 0), а также приложениям, завершающим работу аварийно или исчезающим после вызова cm_request().

3.4 Запросы

Если приложения хотят узнать доступную пропускную способность и время кругового обхода по потокам, они могут воспользоваться вызовом CM cm_query(i32 cm_streamid, i64* rate, i32* srtt, i32* rttdev), который возвращает нужные значения. Если у CM нет действительных оценок для макропотока, указываются отрицательные значения для rate, srtt и rttdev.

[1] Explicit Congestion Notification - явное уведомление о перегрузке.
[2] В оригинале ошибочно сказано nrecd, см. https://www.rfc-editor.org/errata/eid7818. *Прим. перев.*

3.5 Гранулярность обобществления

Одним из решений, которые нужно принимать CM, является гранулярность организации макропотока - выбор потоков, включаемых в него и обобществляющих сведения о перегрузках. API предоставляет две функции, позволяющие приложениям решить, какие потоки следует включать в один макропоток. Функция cm_getmacroflow(i32 cm_streamid) возвращает уникальный идентификатор макропотока (i32), а cm_setmacroflow(i32 cm_macroflowid, i32 cm_streamid) относит поток cm_streamid к макропотоку cm_macroflowid. Если при вызове cm_setmacroflow() передано значение cm_macroflowid = -1, создаётся новый макропоток, идентификатор которого возвращается вызывающей стороне. Каждый вызов cm_setmacroflow() переопределяет для потока макропоток, к которому он относится (при наличии).

По умолчанию предполагается агрегирование по IP-адресу получателя, т. е. все потоки на один адрес агрегируются в один макропоток. В некоторых случаях такое агрегирование будет неоптимально даже в сетях best-effort. Например, если группа получателей находится за шлюзом NAT, отправитель будет видеть для всех один адрес получателя. Если эти узлы за NAT фактически подключены через разные узкие места (bottleneck), работа некоторых из них может ухудшиться в результате. Такие узлы можно обнаружить путём оценки задержки и потерь, но конкретные механизмы такой оценки выходят за рамки этого документа. Вызовы cm_getmacroflow() и cm_setmacroflow() позволяют изменить принятое по умолчанию поведение.

Целью данного интерфейса является настройка групп обобществления, а не политики совместного использования относительных весов потоков в макропотоке, для которой от планировщика требуется интерфейс установки правил совместного использования. Поскольку нужно поддерживать множество разных планировщиков (каждому из которых можгу требоваться разные данные для установки правил), здесь не указывается полный интерфейс API для планировщика (см. параграф 5.2). Предполагается, что в одном из последующих документов будет описано несколько простых планировщиков (например, круговой перебор с учётом весов, иерархическое планирование) и предоставляемых ими API.

4. Устройство CM

В этом разделе описываются внутренние компоненты CM - контроллер перегрузок (Congestion Controller) и планировщик (Scheduler) с чётко определёнными абстрактными интерфейсами, которые они экспортируют.

4.1 Контроллер перегрузок

С каждым макропотоком связан алгоритм контроля перегрузок, а совокупность этих алгоритмов образует контроллер перегрузок CM. Алгоритм контроля решает, когда и сколько можно передать данных через макропоток. Алгоритм использует уведомления приложений (параграф 4.3) из одновременных потоков одного макропотока для сбора сведений о состоянии перегрузки на пути макропотока через сеть.

Контроллер перегрузок **должен** реализовать дружественный к TCP (TCP-friendly) [Mahdavi98] алгоритм контроля перегрузок. Несколько макропотоков **могут** (зачастую, будут) использовать один алгоритм контроля перегрузок, но каждый макропоток поддерживает своё состояние для сети, используемой его потоками.

Модуль контроля перегрузок **должен** реализовать указанные ниже абстрактные интерфейсы, которые напрямую не видны приложениям, размещаются в контексте макропотока и отличаются от функций CM API, описанных в разделе 3.

void query(u64 *rate, u32 *srtt, u32 *rttdev)
: Возвращает оценку скорости (бит/сек) и сглаженное время кругового обхода (мксек) для макропотока.

void notify(u32 nsent)
: Эта функция **должна** применяться для уведомления модуля контроля перегрузок при каждой отправке данных приложением. Параметр nsent указывает число байтов, переданных приложением.

void update(u32 nsent, u32 nrecd, u32 rtt, u32 lossmode)
: Эта функция вызывается всякий раз, когда любой из потоков CM, связанных с макропотоком, видит, что данные были получены приёмником или потеряны в пути. Параметр nrecd указывает число байтов, достигших получателя, nsent - сумму числа пришедших к получателю и потерянных в сети байтов, rtt указывает оценку времени кругового обхода (мксек) в процессе передачи, а lossmode служит индикатором способа обнаружения потерь (параграф 3.3).

Хотя эти интерфейсы не видны приложениям, контроллер перегрузок **должен** реализовать абстрактные интерфейсы для функциональной совместимости модулей с независимо разработанными планировщиками. Модуль планировщика **должен** вызывать функцию расписания соответствующего планировщика (параграф 5.2), когда он считает, что текущее состояние позволяет передать пакет размером MTU.

4.2 Планировщик

Модуль контроля перегрузкок отвечает за определение момента и числа передаваемых данных, а планировщик макропотока определяет, каким потокам можно разрешить передачу данных. Планировщик **должен** реализовать указанные ниже интерфейсы.

void schedule(u32 num_bytes)
: Когда модуль контроля перегрузок решает, что можно передать данные, **должна** вызываться процедура schedule() с указанием числа байтов не больше разрешённого для отправки. Планировщик **может** вызвать функцию cmapp_send(), которую должны предоставлять приложения CM.

float query_share(i32 cm_streamid)
: Возвращает долю указанного потока в общей пропускной способности макропотока и в сочетании с запросом контроллера перегрузок предоставляет сведения для выполнения запроса приложения cm_query().

void notify(i32 cm_streamid, u32 nsent)
: Служит для уведомления модуля планирования о каждой отправке данных приложением CM. Параметр nsent указывает число переданных приложением байтов.

Планировщик **может** реализовать дополнительные интерфейсы. По мере накопления опыта работы с планировщиками CM в будущих документах могут вноситься изменения и дополнения в некоторые части API планировщика.

5. Примеры

5.1 Примеры приложений

В этом разделе описаны три возможных варианта примерения CM API приложениями - два асинхронных (отправитель TCP и сокеты UDP с контролем перегрузки) и одно синхронное (потоковых звуковой сервер). Более подробно эти приложения, а также оптимизация реализаций CM для эффективной работы описаны в [Andersen00].

Все приложения, использующие CM, **должны** обеспечивать отклики от получателя. Например, источник должен периодически (обычно 1 или 2 раза за интервал кругового обхода) определять, сколько его пакетов прибыло к получателю. Когда источник получает такую обратную связь, он **должен** вызывать функцию cm_update() для передачи CM новых сведений. Это ведёт к обновлению диспетчером CM значения ownd и может приводить к смене CM своих оценок и вызову cmapp_update() для потоков макропотока.

Указанные ниже протоколы являются примерами и предложениями для реализации, а не требованиями.

5.1.1 TCP

Реализации TCP с поддержкой CM следует использовать функцию cmapp_send() API обратных вызовов. TCP определяет, какие данные следует передать, лишь после получения подтверждения или завершения отсчёта таймера. Нужен жёсткий контроль над передачей новых данных и повтором передачи.

Когда TCP подключается к удалённому хосту или принимает соединение, выполняется вызов cm_open() для привязки соединения TCP к cm_streamid. После организации соединения CM используется для контроля передачи исходящих данных. CM избавляет от необходимости отслеживания и реагирования на перегрузку в TCP, поскольку CM и его API передачи обеспечивают корректное поведение при перегрузке. Восстановлением потерь по-прежнему занимается TCP на основе ускоренного повтора передачи, восстановления и тайм-аутов. Кроме того, в TCP вносятся изменения для сасооценки окна остающихся данных (tcp_ownd). При каждой передаче сегментов данных путём вызова cmapp_send() TCP обновляет значение tcp_ownd. Переменная ownd обновляется также после каждого вызова cm_update(). TCP также подсчитывает число остающихся сегментов (pkt_cnt). В любой момент TCP может рассчитать средний размер пакета (avg_pkt_size) как tcp_ownd/pkt_cnt. Значение avg_pkt_size используется в TCP при оценке объёма оставшихся данных. Отметим, что это не требуется при использовании в соединении опции SACK, поскольку сведения доступны явно.

Изменения в выходных процедурах TCP указаны ниже.

1. Исключаются все проверки окна перегрузки (cwnd).
2. При доступности данных приложения выходные процедуры TCP выполняют все проверки, не связанные с перегрузкой (алгоритм Nagle, анонсированное получателем окно и т. п.). При положительном результате выходная процедура помещает данные в очередь и вызывает cm_request() для потока.
3. Если по входящим данным или таймерам фиксируются потери, данные для повторной передачи помещаются в очередь и вызывается функция cm_request() для потока.
4. В выходной процедуре TCP устанавливается обратный вызов cmapp_send(). При наличии в очереди данных для повторной передачи выполняется повтор отправки, иначе процедура вывода отправляет новые данные в объёме, разрешенном для соединения. Функция cmapp_send() никогда не передаёт более одного сегмента на вызов. Эта процедура организует другие расчёты для вывода, такие как создание заголовка и опций.

Выходная процедура IP на хосте вызывает cm_notify(), когда пакеты фактически переданы наружу. Поскольку она не знает cm_streamid для пакета, он указывается аргументом stream_info (содержимое stream_info указано в разделе 3). Поскольку cm_notify() сообщает размер данных IP (payload), TCP отслеживает общий размер заголовка и учитывает эти обновления.

Изменения во входных процедурах TCP указаны ниже.

1. Оценка RTT выполняется как обычно по временным меткам или алгоритму Karn. Значение rtt передаётся CM через вызов cm_update.
2. Все обновления cwnd порога замедленного старта (ssthresh) исключаются.
3. При поступлении подтверждения для новых данных TCP рассчитывает значение in_flight (размер находящихся в сети данных) как snd_max-ack-1 (MAX Sequence Sent - Current Ack - 1), после чего вызывает cm_update(streamid, tcp_ownd - in_flight, 0, CM_NO_CONGESTION, rtt).
4. При получении дубликата подтверждения TCP проверяет свой счётчик дубликатов (dup_acks) для определения действия. Если dup_acks < 3, TCP не делает ничего. При dup_acks == 3 предполагается, что пакет был потерян и генерации дубликата потверждения вызвана прибытием не менее 3 пакетов. Поэтому вызывается функция cm_update(streamid, 4 * avg_pkt_size, 3 * avg_pkt_size, CM_LOSS_FEEDBACK, rtt). Средний размер пакета используется потому, что подтверждения не указывают точно размер прибывших к получателю данных. Большинство реализаций TCP считают дубликат ACK индикацией прибытия к получателю полного MSS. После приёма нового ACK такие реализации отправителя TCP могут повторить синхронизацию с получателем TCP. CM API не предоставляет TCP механизма передачи сведений о ресинхронизации, поэтому TCP может лишь сделать вывод о поступлении avg_pkt_size данных из приёма каждого дубликата подтверждения. TCP помещает потерянный сегмент в очередь повторной передачи и вызывает cm_request(). При dup_acks > 3 TCP предполагает, что пакет прибыл к получателю и вызвал отправку подтверждения. В результате вызывается функция cm_update(streamid, avg_pkt_size, avg_pkt_size, CM_NO_CONGESTION, rtt).
5. При получении частичного подтверждения (не превышающего наибольший сегмент переданный на момент потери, как указано в [Floyd99b]) TCP считает, что пакет был потерян, а переданный повторно пакет пришел к получателю. Поэтому вызывается функция calls cm_update(streamid, 2 * avg_pkt_size, avg_pkt_size, CM_NO_CONGESTION, rtt). Использование CM_NO_CONGESTION обусловлено тем, что период потерь уже указан. Потерянный сегмент помещается в очередь повторной передачи и вызывается функция cm_request().

По завершении отсчёта таймера повторной передачи TCP отправитель считает, что пакет был потерян и вызывает функцию cm_update(streamid, avg_pkt_size, 0, CM_NO_FEEDBACK, 0) для обозначения отсутствия отклика получателя и «ухода одного сегмента из трубы». Потерянный сегмент сегмент помещается в очередь повторной передачи и вызывается функция cm_request().

5.1.2 UDP с контролем перегрузок

UDP с контролем перегрузки - это полезное применение CM, описываемое здесь в контексте сокетов Berkeley [Stevens94]. Они обеспечивают функциональность стандартных сокетов Berkeley UDP, но вместо незамедлительной передачи пакетов из очереди ядра нижележащим уровням для отправки реализация буферизованного сокета вызывает функцию API, экспортируемую CM внутри ядра и получает обратный вызов от CM. При создании сокета CM UDP он связывается с определенным потоком, а позднее при добавлении данных в очередь пакетов вызывается функция cm_request() для связанного с сокетом потока. При планировании CM передачи в этом потоке вызывается функция udp_ccappsend() в модуле UDP, которая передаёт из очереди пакетов MTU данных и планирует передачу любых остающихся в очереди пакетов. Реализации CM UDP API не следует требовать дополнительных копий данных и следует поддерживать все стандартные опции UDP. Изменение имеющихся приложений для использования UDP с контролем перегрузок требует реализации новой опции сокетов. Для корректной работы отправитель должен получать сведения о перегрузке. Это можно сделать двумя путями - (i) отклики от принимающего приложения UDP с обратной связью для отправителя или (ii) обратная связи от приёмного модуля UDP к отправителю UDP. Отметим, что второй вариант требует изменений в сетевом стеке получателя, а отправитель UDP не может узнать о поддержке этого без явного согласования.

5.1.3 Звуковой сервер

Типичное голосовое приложение зачастую имеет доступ к выборкам с разной частотой и качеством. Задача приложения состоит в обеспечении клиентам максимально возможного качества звука (обычно с наибольшей частотой). Выбор варианта звука для передачи следует основывать на текущем состоянии перегрузки в сети. Кроме того, источник будет хотеть доставлять звуковой поток получателям с постоянной частотой выборки. В результате он должен передавать данные с регулярной скоростью, минимизируя задержки при передаче и сокращая буферизацию перед воспроизведением. Для выполнения этих требований приложение может использовать API синхронной отправки (параграф 3.2).

При первом запуске источника он использует вызов cm_query() для получения начальной оценки пропускной способности и задержки в сети. Если уже активны другие потоки того же макропотока, источник получает действительную начальную оценку, иначе он получит отрицательные значения, которые игнорируются. Затем источник выбирает кодирование, не выходящее за пределы оценок (или заданное приложением по умолчанию в случае недействительной оценки), и начинает передачу данных. Приложение также реализует обратный вызов cmapp_update(). Когда CM видит изменение характеристик сети, вызывается функция приложения cmapp_update() с передачей новой оценки скорости и времени кругового обхода. Приложение должно изменить кодирование звука, чтобы не выйти за вновь указанные оценки.

5.2 Пример модуля контроля перегрузок

Для иллюстрации обязанностей модуля контроля перегрузок ниже описаны некоторые действия простого модуля контроля перегрузок в стиле TCP, реализующего аддитивное увеличение и мультипликатовной снижение (Additive Increase Multiplicative Decrease congestion control или AIMD_CC):

query()

AIMD_CC возвращает текущее окно перегрузки (cwnd), делённое на сглаженное значение rtt (srtt) в качестве оценки пропускной способности. Сглаженная оценка rtt возвращается как srtt.

notify()

AIMD_CC добавляет число переданных байтов к окну остающихся данных (ownd).

update()

AIMD_CC вычитает nsent из ownd. Если значение rtt отлично от 0, AIMD_CC обновляет srtt, используя расчёт TCP srtt. Если обновление говорит о потере данных, AIMD_CC устанавливает для cwnd значение 1 MTU при loss_mode = CM_NO_FEEDBACK и cwnd/2 (не менее 1 MTU) при loss_mode = CM_LOSS_FEEDBACK или CM_EXPLICIT_CONGESTION. AIMD_CC также устанавливает для внутренней переменной ssthresh значение cwnd/2. Если потерь не возникает, AIMD_CC имитирует замедленный старт TCP и линейный рост. Значение cwnd увеличивается на nsent, когда cwnd < ssthresh (ограничено максимум ssthresh-cwnd) и на nsent * MTU/cwnd при cwnd > ssthresh.

Когда cwnd и ownd обновляются и указывают возможность передачи по меньшей мере MTU данных, AIMD_CC вызывает CM для планирования передачи.

5.3 Пример модуля планировщика

Для пояснения обязанностей планировщика ниже описаны некоторые действия простого модуля планирования с круговым обходом (round robin scheduler, RR_sched).

schedule()

RR_sched планирует число потоков, с которыми можно работать в режиме перебора по кругу.

query_share()

RR_sched возвращает значение 1/(число потоков в макропотоке).

notify()

RR_sched не делает ничего. На планирование с перебором по кругу не влияет размер переданных данных.

6. Вопросы безопасности

CM предоставляет многие из услуг, обеспечиваемых контролем перегрузки в TCP, и имеет такие же проблемы безопасности. Например, некорректные сведения о потерях и передачах будут давать CM некорректное представление состояния перегрузок в сети. Давая CM завышенную оценку перегрузки, злоумышленник может негативно повлиять на производительность приложений. Например, поток хоста может произвольно замедлить любой другой поток того же макропотока, что является формой отказа в обслуживании. Более опасные атаки становятся возможными при

заниженной оценке перегрузки, предоставленной приложениями CM, что заставит CM быть излишне агрессивным и передавать данные быстрее, чем это допускают разумные правила контроля перегрузок.

В [Touch97] рассмотрен ряд проблем безопасности, возникающих при обобществлении сведений о перегрузках. Дополнительная уязвимость, не отмеченная в [Touch97], возникает из-за доступа приложений через CM API к управлению общим состоянием перегрузки, влияющему на другие приложения на том же компьютере. Например, плохо спроектированное, скомпрометированное или вредоносное приложение UDP может злоупотреблять вызовом cm_update() для создания препятствий другим потокам макропотока.

7. Литература

[Allman99] Allman, M. and Paxson, V., "TCP Congestion Control", RFC 2581, April 1999.

[Andersen00] Balakrishnan, H., System Support for Bandwidth Management and Content Adaptation in Internet Applications, Proc. 4th Symp. on Operating Systems Design and Implementation, San Diego, CA, October 2000. Available from http://nms.lcs.mit.edu/papers/cm-osdi2000.html[1]

[Balakrishnan98] Balakrishnan, H., Padmanabhan, V., Seshan, S., Stemm, M., and Katz, R., "TCP Behavior of a Busy Web Server: Analysis and Improvements," Proc. IEEE INFOCOM, San Francisco, CA, March 1998.

[Balakrishnan99] Balakrishnan, H., Rahul, H., and Seshan, S., "An Integrated Congestion Management Architecture for Internet Hosts," Proc. ACM SIGCOMM, Cambridge, MA, September 1999.

[Bradner96] Bradner, S., "The Internet Standards Process --- Revision 3", BCP 9, RFC 2026, October 1996.

[Bradner97] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[Clark90] Clark, D. and Tennenhouse, D., "Architectural Consideration for a New Generation of Protocols", Proc. ACM SIGCOMM, Philadelphia, PA, September 1990.

[Eggert00] Eggert, L., Heidemann, J., and Touch, J., "Effects of Ensemble TCP," ACM Computer Comm. Review, January 2000.

[Floyd99a] Floyd, S. and Fall, K.," Promoting the Use of End-to-End Congestion Control in the Internet," IEEE/ACM Trans. on Networking, 7(4), August 1999, pp. 458-472.

[Floyd99b] Floyd, S. and T. Henderson,"The New Reno Modification to TCP's Fast Recovery Algorithm," RFC 2582, April 1999.

[Jacobson88] Jacobson, V., "Congestion Avoidance and Control," Proc. ACM SIGCOMM, Stanford, CA, August 1988.

[Mahdavi98] Mahdavi, J. and Floyd, S., "The TCP Friendly Website," http://www.psc.edu/networking/tcp_friendly.html

[Mogul90] Mogul, J. and S. Deering, "Path MTU Discovery," RFC 1191, November 1990.

[Padmanabhan98] Padmanabhan, V., "Addressing the Challenges of Web Data Transport," PhD thesis, Univ. of California, Berkeley, December 1998.

[Paxson00] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.

[Postel81] Postel, J., Editor, "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[Ramakrishnan99] Ramakrishnan, K. and Floyd, S., "A Proposal to Add Explicit Congestion Notification (ECN) to IP," RFC 2481, January 1999.

[Stevens94] Stevens, W., TCP/IP Illustrated, Volume 1. Addison-Wesley, Reading, MA, 1994.

[Touch97] Touch, J., "TCP Control Block Interdependence", RFC 2140, April 1997.

8. Благодарности

Спасибо David Andersen, Deepak Bansal и Dorothy Curtis за их работу по проектированию и реализации CM. Спасибо Vern Paxson за подробные комментарии, отклики и терпение, а также Sally Floyd, Mark Handley, Steven McCanne за полезные отклики по архитектуре CM. Allison Mankin и Joe Touch предоставили ценные замечания к черновым вариантам документа.

9. Адреса авторов

Hari Balakrishnan
Laboratory for Computer Science
200 Technology Square
Massachusetts Institute of Technology
Cambridge, MA 02139
EMail: hari@lcs.mit.edu
Web: http://nms.lcs.mit.edu/~hari/

Srinivasan Seshan
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213
EMail: srini@cmu.edu
Web: http://www.cs.cmu.edu/~srini/

[1]Приведённая ссылка утратила актуальность. Документ в формате PDF доступен по ссылке. *Прим. перев.*

Полное заявление авторских прав

Подтверждение

Финансирование функций RFC Editor обеспечено Internet Society.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru