讨论课 01

1352896 王思尧

用户登录后始终在线，考虑低带宽 / 不稳定网络

一 长连接心跳机制

引导实例－MQTT协议

长连接心跳机制典型的应用是互联网推送服务，手机每天都会收到很多推送消息，这些推送服务的原理都是维护一个长连接，而维护任何一个长连接都需要心跳机制，客户端发送一个心跳给服务器，服务器给客户端一个心跳应答，从而形成客户端服务器的一个完整握手。如果超过一个时间的阈值，客户端没有收到服务器的应答，或者服务器没有收到客户端的心跳，那么对客户端来说则断开与服务器的连接重新建立一个连接，对服务器来说只要断开这个连接即可。

PS：iOS长连接是由系统来维护的，也就是说苹果的iOS系统在系统级别维护了一个客户端和苹果服务器的长链接，iOS上的所有应用上的推送都是先将消息推送到苹果的服务器然后将苹果服务器通过这个系统级别的长链接推送到手机终端上。Android的长连接是由每个应用各自维护的，但是google也推出了和苹果技术架构相似的推送框架，但是由于种种原因google的服务器不在中国境内，所以导致这个推送无法使用。

应用层心跳包

所谓的心跳包就是客户端定时发送简单的信息给服务器端告诉它我还在而已。代码就是每隔几分钟发送一个固定信息给服务端，服务端收到后回复一个固定信息如果服务端几分钟内没有收到客户端信息则视客户端断开。比如有些通信软件长时间不使用，要想知道它的状态是在线还是离线就需要心跳包，定时发包收包。发包方：可以是客户也可以是服务端，看哪边实现方便合理。一般是客户端。服务器也可以定时轮询发心跳下去。心跳包之所以叫心跳包是因为：它像心跳一样每隔固定时间发一次，以此来告诉服务器，这个客户端还活着。事实上这是为了保持长连接，至于这个包的内容，是没有什么特别规定的，不过一般都是很小的包，或者只包含包头的一个空包。

TCP协议长连接

TCP协议中有长连接和短连接之分。短连接在数据包发送完成后就会自己断开，长连接在发包完毕后，会在一定的时间内保持连接，即我们通常所说的Keepalive（存活定时器）功能。默认的Keepalive超时需要7,200,000 milliseconds，即2小时，探测次数为5次。它的功效和用户自己实现的心跳机制是一样的。开启Keepalive功能需要消耗额外的宽带和流量，尽管这微不足道，但在按流量计费的环境下增加了费用，另一方面，Keepalive设置不合理时可能会因为短暂的网络波动而断开健康的TCP连接。

keepalive并不是TCP规范的一部分。
在Host Requirements RFC罗列有不使用它的三个理由：
（1）在短暂的故障期间，它们可能引起一个良好连接（good connection）被释放（dropped）
（2）它们消费了不必要的宽带
（3）在以数据包计费的互联网上它们（额外）花费金钱。然而，在许多的实现中提供了存活定时器。

一些服务器应用程序可能代表客户端占用资源，它们需要知道客户端主机是否崩溃。存活定时器可以为这些应用程序提供探测服务。Telnet服务器和Rlogin服务器的许多版本都默认提供存活选项。
个人计算机用户使用TCP/IP协议通过Telnet登录一台主机，这是能够说明需要使用存活定时器的一个常用例子。如果某个用户在使用结束时只是关掉了电源，而没有注销（log off），那么他就留下了一个半打开（half-open）的连接。如果客户端消失，留给了服务器端半打开的连接，并且服务器又在等待客户端的数据，那么等待将永远持续下去。存活特征的目的就是在服务器端检测这种半打开连接。也可以在客户端设置存活器选项，且没有不允许这样做的理由，但通常设置在服务器。如果连接两端都需要探测对方是否消失，那么就可以在两端同时设置（比如NFS）。

keepalive工作原理：
若在一个给定连接上，两小时之内无任何活动，服务器便向客户端发送一个探测段。（我们将在下面的例子中看到探测段的样子。）客户端主机必须是下列四种状态之一：
1) 客户端主机依旧活跃（up）运行，并且从服务器可到达。从客户端TCP的正常响应，服务器知道对方仍然活跃。服务器的TCP为接下来的两小时复位存活定时器，如果在这两个小时到期之前，连接上发生应用程序的通信，则定时器重新为往下的两小时复位，并且接着交换数据。
2) 客户端已经崩溃，或者已经关闭（down），或者正在重启过程中。在这两种情况下，它的TCP都不会响应。服务器没有收到对其发出探测的响应，并且在75秒之后超时。服务器将总共发送10个这样的探测，每个探测75秒。如果没有收到一个响应，它就认为客户端主机已经关闭并终止连接。
3) 客户端曾经崩溃，但已经重启。这种情况下，服务器将会收到对其存活探测的响应，但该响应是一个复位，从而引起服务器对连接的终止。
4) 客户端主机活跃运行，但从服务器不可到达。这与状态2类似，因为TCP无法区别它们两个。它所能表明的仅是未收到对其探测的回复。

总结：考虑到低带宽网络不稳定情况，长连接占用资源比较大，用自己实现的心跳包机制比较好。

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
http://qiita.com/lucifdch/items/bfb98074636a8222ddcc

二 消息不遗漏

本组的项目中使用了rabbitmq，其通过消息确认机制在一定程度上确保了消息不遗漏。

一项任务可能会占用数秒的时间。如果某个消费者启动一个长时间的任务，然后还没做完任务就死掉了，一旦RabbitMQ将某条消息传递给消费者之后，它就会立即从内存中删除这条消息，在这种情况下，如果杀死一个工作者的话，便会丢失它刚才正在处理的消息。并且，所有那些已被分发给该工作者却没有被处理的消息，也会丢失。然而，我们并不想丢失任何任务。如果某个工作者死掉了，那么，我们希望该任务被重新分发给另一个工作者。

为了确保不遗漏任何一条消息， RabbitMQ支持消息的确认 。回执ack(nowledgement)是由消费者回发的，用来告知 RabbitMQ ，某条特定的消息已经被接收到 、并且处理完毕了，从此 RabbitMQ 可以随意删除它了。
如果某个消费者死掉了而没有发送回执，那么，RabbitMQ就会知道，某条消息并没有被完整地处理，因而会将该条消息重新分发给另一个消费者。那样，就可以确保，即使工作者偶尔死掉，也不会有消息丢失。
没有任何的消息超时：RabbitMQ只会在工作者的连接断开的时候重新分发消息。即使消息的处理过程要占用狠长狠长的时间，也没有问题。

——————————————————————————————

http://stupidbeauty.com/Blog/article/1672/RabbitMQ%20Java%E6%95%99%E7%A8%8B%E7%BF%BB%E8%AF%91%EF%BC%9A%E5%B7%A5%E4%BD%9C%E9%98%9F%E5%88%97%EF%BC%8CWork%20Queues

三 消息不重复

在rabbitmq中，生产者的消息确认叫做confirm，confirm确保消息已经发送到MQ中。当connection或channel异常时，会重新发送消息，如果消息是持久的，并不能一定保证消息持久化到磁盘中，因为消息可能存在与磁盘的缓存中。
当生产者收不到消息确认的时候，消息可能会重复，所以如果消息不允许重复，则消费者需要自己实现消息去重。

消费者去重可以使用给消息添加序列号即标示id的方法，在客户端对比收到的消息的序列号即可判断收到的消息是否位重复消息。

PS:阿里云消息队列负责人表示，产品团队正在攻克“消息不重复”这个难题，该功能最快将于2016年中旬推出。

——————————————————————————————

http://dc0127.blog.163.com/blog/static/11217896201322394228794/

四 消息压缩

低带宽的情况下消息压缩尤为重要。
RabbitMq的序列化是指Message的body属性，即我们真正需要传输的内容。
RabbitMq抽象出一个MessageConvert接口处理消息的序列化，其实现SimpleMessageConverter、Jackson2JsonMessageConverter等，其中默认的序列化类为SimpleMessageConverter。只有调用了convertAndSend方法才会使用相应的MessageConvert进行消息的序列化与反序列化。
SimpleMessageConverter对于要发送的消息体body为字节数组时，不进行处理。对于如果是String，则将String转成字节数组。对于如果是Java对象，则使用jdk序列化将消息转成字节数组，转出来的结果较大，含class类名，类相应方法等信息。因此性能较差。
当使用RabbitMq作为中间件时，数据量比较大，此时就要考虑使用类似Jackson2JsonMessageConverter，hessian等序列化形式。以此提高性能。

——————————————————————————————

http://www.it610.com/article/3907476.htm

知乎上的栗子

很多人提到keepalive，TCP无法感知网络中断这些问题。。。这个算是TCP一个容易踩的坑，但这并不能说明UDP就比TCP好（或者说解释为何要使用UDP）。因为在UDP上面一样需要面对这些问题，而解决这类问题的方法和在TCP上面进行应用层心跳的方法其实没有本质上的区别。而这就是为什么没有接触过这类问题的人会有题主提出的疑惑。

那么为什么呢？最本质上UDP的优势还是带宽的利用。这一切要回归到99~03年的网络状况，当时网络的特点就是接入带宽很窄而且抖动特别厉害。所谓抖动可能是多方面的，例如延时突发性地暴

增、也有可能是由于路由层面的变化突然导致路由黑洞，还各种等等等等的问题。TCP因为拥塞控制、保证有序等原因，在这种网络状态上对带宽的利用是非常低的。而且因为网络抖动的原因，应用层心跳超时（一般不依靠keepalive）应用层主动断掉socket之后TCP需要三次握手才能重新建立链接，一旦出现频繁的小抖动就会使得带宽利用更低。而等待四次挥手的时间，也会占用服务器上宝贵的资源。

总结来说，当网络差到一定程度了，TCP的优势反而会成为劣势。

这时候我们再看看UDP在这种情况下的表现。使用UDP对抗网络抖动，说到底就是在应用层比TCP更快地探测和重传，一旦超过一定的时间没有收到回复，客户端可以选择马上重试或者换一个IP:PORT重试（假如你的服务像QQ一样有多个接入），在服务器端则可以果断地断掉socket。而可以应用UDP的时候，往往是你的应用层协议本身已经具备了一定的面向连接的特性。如果你应用层的协议已经达到了一定程度的消息幂等，客户端可以几乎无脑地进行重传，这样就可以尽可能地降低网络抖动的影响，同时也可以尽可能地利用整个带宽。而刚好QQ的协议，就具备类似的特点。

简单来说就是我们可以使用UDP实现一个面向连接协议，这个协议可以很好地适应当时的网络状况和QQ本身的业务。但凡事都有成本，成本就是你的应用层协议本身需要去实现抵抗网络异常带来的问题。例如乱序、例如业务数据的分片和重组、例如网络状态探测等等等等。。。

而现在UDP也应用在很多跨运营商、跨地域、跨机房之间的服务调用当中。原因无它，就是网络烂到一定程度了。

作者：hailiang huang
链接：http://www.zhihu.com/question/20292749/answer/85286488
来源：知乎
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。