Report on MegePipe

MegaPipe: A New Programming Interface for Scalable Network I/O

这篇论文是按照上周课上的要求，看完和Affinity Accept（AA）的方法进行了对比。

本文主要阐述了在多核系统的通信中，如何提高进程间通信（包括基于文件系统的通信和基于网络端口的通信）接收和处理效率的问题。

和AA相比，本文在立意上不局限于TCP端口，比AA适用性更广。但实际上，由于本文针对的通信是高接通率（high connection rate）和短消息（short message）的情况。因此还是以网络通信为主的。

**本文主要创新点如下：**

（1）设计了MegaPipe的进程间通信架构。

（2）顶层：

1. 分块（Partitioning）：对listening socket进行分块来利用多核的优势，提高性能。

2. 轻量级（lwsocket）：对于socket进行轻量级处理。

3. 批处理（Batching）：对系统调用批处理化，实质上是指，处理器的请求和数据的读入都实现复用和批处理

（3）底层：

1. 综合AA中的每个核一个通道的概念，实现了每个application一个通道。

2. 实现了双向管道。

（4）实验验证了系统可实现性，并提高了吞吐率（29%~582%），提高了性能（15%~320%）。对于一个实际的HTTP程序，吞吐率提升了75%。

**本文的解决思路如下：**

对于进程间通信，文章提出面对高接通率（high connection rate）和短消息（short message）时，现有的方法效率很低，适应性很差。文章分析认为主要因素如下：

（1）只有一个Accept Queue，因此处理器需要串行的访问；只有单一的锁对socket进行保护，并发性能很差。

（2）对于连接（connection）的相关性缺乏利用很差。由于发送和接收不在同一个核上。

这两个因素和AA分析的基本一致，AA还增加关于调度的代价的分析。

（3）由于传统socket数据结构的限制。传统的socket的处理方式是归类为文件处理。即，一个socket被认为是一个文件。因此存在两个问题：

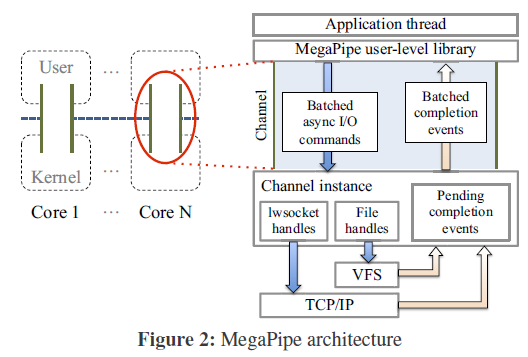
1. socket产生的时候，需要归入文件类，由File Descriptors来给出一个编号。这个编号是现存可用的最小的整数。

2. socket的读写和处理都是通过虚拟文件系统来完成的，由于一般的文件系统存在由虚拟存储到硬盘的多级映射来保证文件的安全和共享。这个过程的代价比较大。

上述的两个情况对于低频率文件产生的情况是很实用的。但是对于快速的文件更新，并且文件长度短，寿命短的情况，这两个的代价就成为了影响性能的主要因素。

（4）系统调用代价过大。这是由于connection的数量多，频繁调用导致系统调用的代价也变得难以承受。

面对这样的情况，传统的解决方案往往是基于现有的BSD socket架构和API的修改，对于性能的提升有限。于是文章提出了一个彻底重建socket架构的MegaPipe架构。



如图是MegaPipe的主要结构。MP主要实现的应用端User和处理器端之间的通道。其中socket架构包含在了处理器端的设计内部。

User端主要是应用程序的线程。与传统的同步调用方式不同，MP中的线程采用异步调用，即轮询+非阻塞函数的形式实现对于进程间数据的读入。在这个基础上，channel中通过复用的方式（文中没有看到，但是感觉应该是时分复用）来批处理异步的 I/O调用和数据读入。因此在API层次上，MP实现了新的结构。对于User而言，这一部分的I/O是透明的。

Core端主要是socket和socket的调用和分配（感觉类似于AA中的NIC功能，但这两个概念是不一样的）。这里的socket部分基本采用了AA中的socket结构，也就是将socket的单一锁拆分成更细粒度的锁，每一个锁都针对一个特定的应用（application）保护一个轻量级的lwsocket，同时原本是一个大的Accept Queue也被分块成多个小的Accept Queue来面向不同的应用。

与AA不同的是，AA中的socket是静态的，每一个核都对应于一个accept queue，可以认为是对应于一个小的socket。但是在MP中，socket是动态产生的，User端发出一个请求，Core端就会产生一个新的socket（lwsocket）。因此这个socket不是面向Core的，而是面向Application的。在这方面，MP的设计更加的灵活。但是相对的，也增加I/O并行（批处理）操作的困难度。

不同之处在本文中增加了对于处理器调用socket的讨论和分析。除了对于accept()函数进行非阻塞+批处理化的修改以外，本文中重点讨论了两部分的内容，也就是对应于前文中的两个主要的挑战：文件编号，socket的多级调用（多级指针）。为了解决这两个问题，本文提出了轻量级接口（lwsocket，light weight socket）的概念。轻量级接口依然保留了文件类型，但是在文件编号和调用方面都进行了简化。

（1）对于文件编号，MP对于lwsocket进行了单独的分类，即用一个独立于文件系统的数据结构来保存lwsocket的列表。在这个列表中，每一个轻量级的socket由一个随机数来表征其文件编号。我认为：由于socket的数据少，寿命短，更新频繁，因此可以一定程度上保证随机数的鲁棒性。这相比于一般的文件，需要检索当前文件的全部编号，并找到最小的可使用编号，要降低很多的复杂度，尤其是socket的生成随着connection的频繁建立而变得更加频繁的时候。

（2）关于文件调用。对于一般的文件，虚拟文件系统（VFS）会对程序要处理的文件建立一个虚拟的映射。由于文件还存在共享的情况，因此虚拟文件系统还对在这个虚拟映射的基础上再增加一级虚拟映射。从而保证文件安全性和共享性。但是对于socket而言，由于socket的数据少，内容短，同时更是只读的，所以其安全性隐患很小；另一方面，由于每一个socket都是对应于同一个connection的，因此其共享性需求也很低。所以，经典文件的两级调用方式对于socket而言是有着极大的冗余的。因此，在MP中，基于已经建立起来的独立于其他文件类型的数据结构，VFS单独对于每一个socket设定一个直接的调用关系，从而可以极大的降低调用代价。

和AA中简单的讨论accept()系统调用的对象相比，MP中对于lwsocket的讨论无疑更加全面而深入。

最终，针对于MP现有的设计架构，文章给出了相关的API函数结构以及代码（貌似是2200行？）。

AA中提及的关于scheduling的讨论，本文中进行了简要的总结，但没有深入的讨论，只是表示是未来的工作方向。