第 34 卷 第 5 期 计 算 机 工 程 2008 年 3 月
Vol.34 No.5 Computer Engineering March 2008
·网络与通信· 文章编号：1000—3428(2008)05—0128—03 文献标识码：A 中图分类号：TP393.02

即时通信信息还原并行处理模型

余壮辉，黄永忠，周 蓓
(信息工程大学信息工程学院，郑州 450002)

摘 要：在即时通信信息还原过程中，引入分布式并发处理的思想，提出基于元组空间的并行处理模型。该模型使用细化任务粒度和主动提取任务相结合的方法，实现自适应的动态负载均衡。通过对任务提取算法的改进，分析还原的可靠性。理论分析和实验测试证明，该模型有效地提高了信息还原的速度。

关键词：即时通信；信息还原；并行化；元组空间

Parallel Model for Presence and Instant Messaging Information Reversion

YU Zhuang-hui, HUANG Yong-zhong, ZHOU Bei
(Institute of Information Engineering, Information Engineering University, Zhengzhou 450002)

【Abstract】 A thought of parallel computing is applied to IM information reversion by putting forward a parallel computing model based on tuplespace. Thinking size and picking it automatically, the model realizes dynamic load balancing. It achieves stability of reversion, through an improved task-taking algorithm. Theoretic analysis and experiments demonstrate that the model can improve the performance effectively.

【Key words】 presence and instant messaging; information reversion; parallel; tuplespace

目前，即时通信信息监听技术主要针对特定网络，采取单个引擎同步集中处理的方式。随着通信量和网络带宽的增加，单个引擎已不能满足需求，信息还原模块由于处理速度的原因成为了系统瓶颈[1-2]，会有丢包的危险。不管是优化信息还原算法，还是提高单个引擎的硬件配置，都无法从根本上满足监听处理的需求。基于上述原因，本文提出一种基于元组空间的异步还原并行处理模型，异步还原是指：截获数据包后，先将其储存下来，再由信息还原模块进行后续处理，本模型将分布式并行化的思想应用于信息还原，提高了处理效率。

1 信息还原并行处理模型

1.1 元组空间

在并行处理模型中，使用元组空间支持并发处理，分布式通信，持久化存储。

元组空间(tuplespace)可以理解为一个支持并发存取的共享空间(共享区)[3]。在元组空间中，数值对(二元、三元、四元等)可以被存储到空间中，并在需要时读出来。元组空间可以在一个分布式系统的范围内被访问，也就是说，任意引擎都可以存放一个元组(即数值对)供其他引擎使用，以元组的方式来传递信息和数据，可以实现分布式通信。另外，由于元组空间中数据对象的存活时间比产生它们的程序时间长，因此元组空间的存储可以看成是一种持久化存储。

支持元组空间的产品目前主要有 Sun 实验室的 Java Spaces 和 IBM 公司的 Tspaces 等。

1.2 并行化处理模型

1.2.1 单一引擎处理过程

在单一引擎的情况下，原始数据集处理过程如下：(1)将单个网络数据包从原始数据集中提取出来；(2)处理引擎根据IM相关协议还原应用层报文[4-5]；(3)重组并存储还原出来的报文。

1.2.2 并行处理基本思想

并行处理模型主要有 4 个部分组成：提取引擎，处理引擎，整合引擎和元组空间。

模型的并行处理过程为：提取引擎负责提取原始数据集中的网络数据包，并放入元组空间；处理引擎从元组空间中取出数据，实现应用层报文的还原；整合引擎负责取出结果，重组并存储还原出来的报文；显然，各个引擎之间的协同和数据交换通过元组空间来进行。处理过程见图 1。

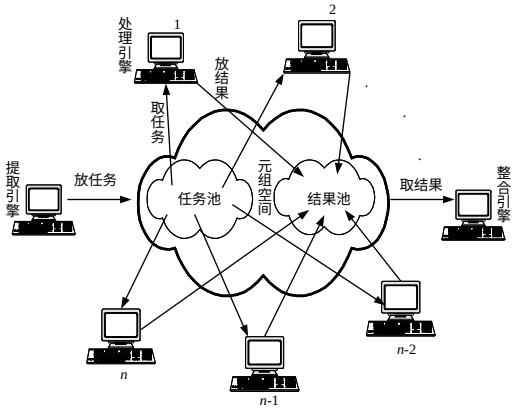

图 1 并行处理过程

在并行处理过程中，引入 2 个元组空间来支持并发和协同，为了方便描述，分别称为任务池和结果池(也可用一个元组空间将任务池和结果池放在一起)；模型采取多个处理引擎

作者简介：余壮辉(1983－)，男，硕士研究生，主研方向：分布式系统；黄永忠，副教授；周 蓓，讲师

收稿日期：2007-08-29 **E-mail：**zhuanghui_yu@163.com

对应用层报文进行还原。

1.2.3 自适应动态负载均衡的实现

提取引擎先将任务粒度细化至单个网络数据包，以单个任务为单位放入任务池。在整个处理过程中，多个处理引擎主动并发地从任务池中取出数据包，并进行处理。对每个处理引擎来说，如果每次只取一个任务，可以以较快的速度完成。在单位时间内，处理能力强的处理引擎可以完成多个任务处理循环，而处理能力较弱的处理引擎就完成较少的任务量，通过各个处理引擎根据自身性能的不同来调整处理任务数目的多少，模型实现了一种自适应的动态负载均衡。

在实际情况中，处理引擎对元组空间进行操作是需要(建立和断开)连接的，如果连接过于频繁，将会过多地占用网络带宽，影响整体性能。因此，为了提高系统的性能，需要尽量降低连接的次数。而连接的频率直接取决于取任务和放结果的频率，降低取放任务的频率有 2 种方法：(1)增大任务粒度；(2)增加每次取任务的数目。如果增大任务粒度，将使提取引擎的处理算法变得复杂(因为每次提取一个网络数据包是最简单易行的)，而且粒度大小的选择需要考虑各个处理引擎的性能，而各个处理引擎的性能又是动态变化的。基于以上原因，本模型使用第(2)种方法，第 m+1 次所取任务数为

$$K_{m+1}=\frac{t_{std}}{t_m}\times\delta+P \tag{1}$$

其中，t_{std}是经验值；t_m是第m次任务所花费的时间；δ是扩放因子，用于调整任务数变化的敏感度；P是任务基数。在最简单的情况下，δ和P都取 1。从式(1)可以看出，K_{m+1}与当前任务完成时间成反比，而当前任务完成的时间一定程度上反映了当前处理引擎的综合处理能力，即反映了当前状况下的负载情况。

2 性能分析

在不影响精确度的情况下，为了便于进行性能分析，特作如下假设。

假设 1 在还原的时候，网络数据包之间不存在依赖关系。

MSN, ICQ, Yahoo和AIM等即时通信软件在通信过程中，因为文本信息是以明文的方式传输的，所以每个网络数据包的还原可以直接进行，不存在先后依赖关系，而国内主流即时通信软件OICQ的文本信息是以加密方式传输的，要想还原其通信内容，需要获得会话密钥(session key)，其还原过程依赖于某些关键数据包的获得[1]。

假设 2 数据包的还原不需要有前后顺序。

在通信过程中，如果某些应用层报文的信息量比较大，就会被分配在不同的数据包中进行传输，这种情况下，在进行数据包的还原时暂时不关心其先后顺序，在重组存储的阶段再根据内容的时间戳进行组合。

假设 3 数据包的存取所需要的时间远小于应用层报文还原的时间。

即假设通过式(1)的平衡，网络访问导致的额外开销可以忽略不计。另外，实际上任务池可以位于提取引擎中，结果池可以位于整合引擎中，这样，放任务和取结果所消耗的时间也可减至最小。在初始状况下，只要任务池中所放置的任务数大于第 1 次各个处理引擎所需取走的任务数，各个处理引擎就可以进行处理，之后的提取引擎放任务是与处理引擎处理数据包相并行的，可以不计其处理时间。

假设 4 每个数据包还原所用的时间都一样。

实际情况中，每个处理引擎的处理能力不同，处理数据包所需要的时间也会有所差别，但该假设并不影响结果的有效性。

基于以上 4 个假设，可得：

(1)单机情况下，采用 m 个引擎并发处理 l 个数据包所需要的时间为

$$T(m,1)=l\times t_{\mathrm{pic}}+\frac{l}{m}t_{\mathrm{rev}}+l\times t_{\mathrm{reg}}$$

其中，t_{pic}是提取一个数据包所耗费的时间；t_{rev}是还原一个数据包所需要的时间；t_{reg}是重组过程所耗费在每个数据包上的平均时间。

(2)采用 n 台处理引擎并发处理所需时间为

$$T(m,n)=l\times t_{\mathrm{pic}}+\frac{l}{m\times n}t_{\mathrm{rev}}+l\times t_{\mathrm{reg}}$$

(3)采用并行处理模型的加速比为

$$S(l,n)=\frac{T(1,1)}{T(m,n)}=\frac{l\times t_{\mathrm{pic}}+l\times t_{\mathrm{rev}}+l\times t_{\mathrm{reg}}}{l\times t_{\mathrm{pic}}+\frac{l}{m\times n}\times t_{\mathrm{rev}}+l\times t_{\mathrm{reg}}}=\frac{t_{\mathrm{rev}}+t_{\mathrm{pic}}+t_{\mathrm{reg}}}{\frac{1}{m\times n}t_{\mathrm{rev}}+t_{\mathrm{pic}}+t_{\mathrm{reg}}}$$

(4)理想状况下， $n\to\infty,1/mn\to 0$ 则

$$\lim_{n\to\infty}S(l,n)=(t_{\mathrm{rev}}+t_{\mathrm{pic}}+t_{\mathrm{reg}})/(t_{\mathrm{pic}}+t_{\mathrm{reg}})=1+t_{\mathrm{rev}}/(t_{\mathrm{pic}}+t_{\mathrm{reg}})$$

(5)由假设 3 可知

$$\lim_{n\to\infty}S(l,n)>>1 \tag{2}$$

由式(2)可知，采用并行处理模型可以提高处理效率。

3 可靠性分析

一个模型可靠与否决定了它的实用价值，因此，论证模型的可靠性是很必要的。假设每个处理引擎的工作状态只有 2 个：正常或出错。定义 $P(X_i(t)=0)$ 为处理引擎 i 在 t 时刻不出错的概率，则整个系统在时刻 t 不出错的概率为

$$P_{sys}(t)=\prod_{i=1}^{n}P(X_i(t)=0) \tag{3}$$

由式(3)可知，在并行处理的情况下，增加处理引擎的数目会降低系统的可靠性，当一个处理引擎取走某个(些)任务后发生故障，将导致该任务无法被正常处理，从而导致系统出错。

本模型使用一种简便易行的方法来提高模型的可靠性。为任务增加一个字段 T，标识任务的处理状态。当处理引擎取任务时，并不真正地从任务池中取走任务，而是修改该任务的 T 字段。任务 j 的 T 字段取值为

$$T_j=\begin{cases}-1\\ \bigcup\limits_{i\in G_j}D_i\end{cases}$$

其中，-1 为T的初值；G_j为处理引擎i的唯一标识(如IP, MAC地址等)；G_j为已取走任务j的处理引擎的集合。

处理引擎 i 取任务的算法如下：

(1)查询任务池，查看是否有符合 $T_j\supset D_i$ 的任务，若有，通知G_j包含的其它引擎该任务已结束，并将该任务从任务池中删除。

(2)如果有T=-1 的任务，就取走(但不删除)，并令$T_j=D_i$。

(3)如果没有T=-1 的任务，就查询T!= -1 的任务，即按T去询问G_j包含的引擎，如果引擎不存在(比如发生故障)，则取走该任务，并令$T_j=D_i$。

(4)如果剩下的每个任务都有处理引擎在执行，则随机取任务并计算，并令 $T_j=T_j\cup D_i$。

(5)如果没有任务，则提示处理结束。

由算法可知，如果某处理引擎取走任务后发生故障，被取走的任务由于有该引擎的标记，因此其他引擎就不会取走，但当所有满足 T=-1 的任务都被处理完，也就是存在空闲引擎的时候，步骤(3)就会被执行，发生故障的引擎起先取走的任务将会被空闲引擎重新取走并处理。步骤(4)显示了一种冗余计算机制，大多数任务已经完成后，使用空闲引擎重复运行未完成的任务，是为了防止某处理引擎处理能力过低而影响整个任务的完成。通过该算法，本模型以较低的代价保证了处理过程的可靠性。

4 测试

(1)标本数据的获得。用同一局域网(共享式)内的 3 台 PC 机在 Windows 环境下以发窗口消息的方法控制 MSN，使其连续发送消息，在其中一台 PC 机上用 Sniffer pro 监听网络，并设置其 IP 和端口过滤等选项，使其集中抓取与 MSN 文字聊天协议族相关的数据包约有 10 万多个，共 15.6 Mb。

(2)测试环境。测试环境按照图 1 构建，使用 tspaces 建立元组空间，由于条件限制，测试时将模型中的提取引擎、任务池、整合引擎、结果池置于同一台机器上(P4 3.0/1 GMb)，采用 P4 2.4/512 Mb 的 PC 机作为处理引擎，测试机之间通过 100 M 局域网相连。

(3)测试过程

1)测试 1

①在单机上进行通信内容的还原，获得单机处理所需要的时间。

②在测试机上设置式(1)中各个参数的值：t_{std}＝5 min, δ＝1, P＝50，启动提取引擎和整合引擎后，逐次增加处理引擎启动的数目。

③获得数据如下：处理引擎数目为 1(未引入模型)、2、4、8 时，耗费时间分别为 59’3, 38’0, 20’1, 11’7。

通过测试 1 可看出，随着处理引擎数目的增加，处理相同数据量所消耗的时间明显下降，说明该模型具有一定的可扩展性，在测试过程中发现，处理引擎的下载任务数K_{m+1}在经过一两次取任务后迅速增长到 1 500 左右，说明处理单个数据包所花费的时间是很少的，如果能够适当增大任务粒度，基本上也能够不影响模型的动态负载平衡，并且可以减少提取引擎和整合引擎的工作量。

2)测试 2

使用 4 台处理引擎，逐次变化式(1)中的t_{std}获得数据，即，t_{std}取值为 10 s, 1min, 3 min, 8 min时，耗费时间分别为 26’7, 22’31, 20’1, 20’0。t_{std}值很大程度上决定了处理引擎每次所取任务数，也决定了处理引擎访问元组空间的频率，通过测试 2 可看出，当t_{std}＝10 s时，处理引擎较为频繁地访问元组空间，也就是较为频繁地进行网络通信，通信所占用时间的延长推迟了总任务的完成，当t_{std}＝3 min和t_{std}＝8 min的时候，处理引擎访问网络的频率明显降低，总任务的完成时间也有所减小，由此可见，如果能恰当地设置式(1)中的各个参数(尤其是t_{std})，能更好地发挥模型的性能。

通过测试 1 和测试 2 可知，本模型具有较好的实用性，可以较大程度地减少处理时间。

5 结束语

模型将元组空间技术引入即时通信的信息还原过程，实现了信息还原处理的并行化和负载均衡，并通过改进提取任务的算法，获得了信息还原的高效性和可靠性。理论分析和实验结果表明，该模型具有较好的性能。在该模型中，提取引擎和整合引擎的故障恢复和容错处理是下一步研究的方向。

参考文献

[1] 孙　波. 即时通信息监听技术的研究与实现[D]. 北京: 中国科学院软件研究所, 2004.

[2] 金　勇. 网络信息内容监控技术及应用研究[D]. 成都: 四川大学, 2005.

[3] IBM Almaden Research Center. Intelligent Connectionware TspacesTechnology[EB/OL]. (2001-05-02). http://www.almaden.ibm.com/cs/TSpaces/papers/TSpaces.pdf.

[4] IEIF. A Model for Presence and Instant Messaging[S]. RFC 2778, 2000- 02.

[5] IEIF. Instant Messaging / Presence Protocol Requirements[S]. RFC 2779, 2000-02.

（上接第 118 页）

上述实验结果也符合文献[5]给出的公式：对于 TCP 数据流，有

$$p=\frac{0.76}{w^2}$$

其中，w为平均发送窗口；p为平均丢包率。由于新策略增大了慢启动发送窗口，使得分组丢失率明显降低。

同时，当N个同等TCP连接共享一条瓶颈链路时(如图 5 中路由器R1, R2 之间)，设该路由缓存允许的最大排队数为K，瓶颈带宽延迟积为L。如果连接符合公平性原则，那么所有窗口之和应满足 $N\cdot w=K+L$[6]。代入上式可得

$$p=0.76\times\frac{N^2}{(K+L)^2}$$

由上式可知，增加缓冲容量和带宽延迟积，或减少链路连接数都有助于减少分组丢失。

5 结束语

本文分析了目前基于窗口的TCP拥塞控制机制及其慢启动策略对网络性能的影响，提出了一种新的慢启动策略COS-Slow-Start。理论分析证明该策略对网络的稳定性和发送效率都有明显的提高。实验表明该策略能有效地降低拥塞发生的可能性，减少分组的丢失，实现高效稳定的 TCP 慢启动过程。由于超时重传的减少，间接地提高了链路带宽的有效利用率，并一定程度上提高了路由器的队列管理性能。但该策略依赖于 *ssthresh* 的准确选取。针对目前策略的不足，下一步将基于延迟时间 RTT 的有效测量以及门限阈值 *ssthresh* 的动态调整，并结合主动队列管理的链路算法作为今后的研究方向，以期进一步提高网络的性能。

参考文献

[1] Jacobson V. Congestion Avoidance and Control[J]. ACM Computer Communication Review, 1988, 18(4): 314-329.

[2] Stevens W. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms[S]. RFC 2001, 1997.

[3] Floyd S. HighSpeed TCP for Large Congestion Windows[S]. RFC 3649, 2003.

[4] Allman M, Floyd S, Partridge C. Increasing TCP’s Initial Window[S]. RFC 2414, 1998.

[5] Morris R. Scalable TCP Congestion Control[C]//Proc. of IEEE INFOCOM’00. Israel, CA: IEEE Computer Society, 2000.

[6] Padhye J, Towsley D, Firoiu V, et al. Modeling TCP Throughput: A Simple Model and Its Empirical Vilidation[J]. ACM SIGCOMM Computer Communication Review, 1998, 28(4): 303-314.