一种基于移动 Agent 技术的分布式死锁检测算法*)

戴 茵 吴 堃 周竞扬 陆桑璐 陈道蓄
(南京大学计算机软件新技术国家重点实验室 南京大学计算机系 南京 210093)

摘 要 死锁的处理长期以来一直是分布式系统的研究重点,已有许多成熟算法。随着网络技术的发展,越来越多的客户和资源可在网络中自由移动,这种可移动性使得传统算法面临了新的挑战。在这种新的应用背景下,本文结合移动 Agent 技术,提出了一种分布式系统死锁检测和解除算法:Agent Guard。该算法使用一个移动 Agent,使其遵循一定的路线算法在各个站点间移动来收集资源请求和分配信息并进行分析,从而发现并解除死锁。模拟实验证明,Agent Guard 算法能取得较短的死锁持续时间,较小的伪死锁率,且网络的通信复杂度也有降低。

关键词 分布式系统,移动 Agent,分布式算法,死锁检测

A New Distributed Deadlock Detection Algorithm Based on Mobile Agent Technology

DAI Han WU Kun ZHOU Jing-Yang LU Sang-Lu CHEN Dao-Xu
(State Key Laboratory for Novel Software Technology, Department of Computer Science & Technology, Nanjing University, Nanjing 210093)

Abstract Deadlock detection and resolution in distributed systems have been studied extensively, and many effective algorithms are proposed. However, traditional algorithms are not suited in the environment in which the clients and resources may move freely. This paper presents a new mobile agent based algorithm, Agent Guard, for deadlock detection and resolution in distributed systems. A mobile agent visits the sites in the distributed system according to some special itinerary algorithm. While moving in the system, the mobile agent collects resource allocation information, analyzes it and resolves deadlocks when deadlock cycles are formed. As the simulation showed, the algorithm can achieve shorter deadlock duration and smaller phantom deadlock ratio; the network communication load is decreased, too.

Keywords Distributed system, Mobile agent, Distributed algorithm, Deadlock detection

1 引言

在分布式系统中,若形成了一个循环等待资源的进程集合,集合中的每个进程都占有部分资源且在等待该集合中的其他进程占有的资源,就产生了"死锁"。死锁降低了系统中资源的利用率,并延长了进程的响应时间。传统方法从死锁的防止、死锁的避免、死锁的检测和解除这三个方面来解决死锁问题。而死锁检测不会对系统的吞吐量产生较大影响,所以,大多数分布式系统倾向于使用死锁检测和解除策略。

从 20 世纪 80 年代开始出现了许多分布式系统的死锁检测和解除算法,但大多数传统的死锁检测算法都建立于一些特定的消息传递机制和静态网络拓扑的假设之上。而在移动网络环境下,网络拓扑结构的频繁变化对基于消息传递的算法提出了新的挑战:由于资源和用户都是可移动的,对资源或进程的准确定位有一定难度,导致消息传递的延迟增加,或导致消息的错误传递乃至丢失,从而使得死锁检测算法的效率大为降低。

近年来,移动 Agent 技术被应用到死锁检测算法中。与一般算法相比,采用移动 Agent 技术的算法至少有如下几个优点:有效节省了网络带宽,缩短了网络延时;具有封装私有特征的能力,可以对协议进行封装;可以匿名执行和异步执行,具有灵活的协调能力;具有很好的动态适应性,更适应动态网络拓扑;具有较强的鲁棒性和容错性[3]。

本文提出了一种基于移动 Agent 技术的分布式死锁检测和解除算法,称为 Agent Guard 算法。该算法利用一个移动 Agent 遵循一定的算法在各个站点间移动并收集资源请求和分配的信息并进行分析,从而发现并解除死锁。Agent Guard 是一种集中和分布相结合的死锁检测算法,移动 Agent 在进程或资源移动时能准确定位,且避免了消息传递在动态网络拓扑中的不可靠性,能够更好地适应移动计算环境。

本文第 2 节简要介绍相关工作。第 3 节给出了 Agent Guard 死锁检测和解除算法的详细描述。第 4 节对该算法进行了性能分析。最后对本文工作进行了总结和展望。

2 相关工作

分布式系统中某个时刻的资源请求和分配情况通常可以用 wait-for-graph(WFG)[5] 来表示。WFG 图是一种有向图,图中的结点表示进程,边表示进程间的等待关系。当进程 P_i 当前申请的资源被进程 P_j 占用时,就在图中表示为一条从 P_i 到 P_j 的有向边。可见系统发生死锁当且仅当 WFG 图中出现有向环或结。

目前已有的分布式系统中的死锁检测算法可以分为三类[10]:集中式算法[6],分布式算法[1,7,11]和层次式算法[4]。集中式算法的主要思想是:系统的 WFG 由一个指定的站点来

*)基金项目:国家高技术研究发展计划 863 项目(No. 2001AA113050),国家重点基础研究发规划 973 项目(No. 2002CB312002)。戴 茵 硕士研究生,研究方向为分布计算与并行计算;吴 堃 硕士研究生,研究方向为分布计算与并行计算;周竞扬 硕士研究生,研究方向分布计算与并行计算;陆桑璐 教授,研究方向为分布计算与并行计算;陈道蓄 教授,博士生导师,研究方向为分布计算与并行计算。

维护，因此检测死锁的任务就由这个中心站点来完成。代表算法有 Ho-Ramamoorthy 的两阶段、一阶段算法[6]。分布式算法的主要思想是：系统的 WFG 分布在各个站点上，并且任何一个站点都只有局部的 WFG 信息，站点之间需要进行一些信息的交换才能检测出全局的死锁环。代表算法有 Chandy-Misra-Haas 的 probe 算法[7]，Obermarck 算法[11]，Choudhary 的 SET 算法[1]等。在层次式算法中：系统中的站点被分成不同的层次，一个站点只能发现它的下层站点形成的死锁。代表算法有 Menasce-Muntz 算法[4]等。

较常用的分布式算法又可以分为四类[5]：路径推进(Path Pushing)，边界追逐(Edge Chasing)，扩散计算(Diffusing Computations)，全局状态检测(Global State Detection)。路径推进算法的主要思想是每个站点都将本地的 WFG 图发送给其他站点，从而系统中每个站点都可以建立全局 WFG 图的某种简化形式，从中发现死锁环的存在。边界追逐算法的主要思想是系统中的等待进程发送一些称为“探针(probe)”的探测消息[7~9,13]，这些探测消息沿着 WFG 图的有向边被转发，当一个“探针”回到发送者时，就表明检测到了一个死锁环。扩散计算和全局状态检测两类算法主要用于通信死锁，本文不再赘述。

传统的死锁检测算法，如集中式算法[6]、边界追逐算法[2,7]和路径推进算法[1,11]等，都对消息传递和网络拓扑做出了如下假设：站点之间通过消息传递来进行通信；传递的消息有两种类型：有关资源管理的消息(包括资源请求、授予、释放等)，以及为检测死锁而交换的消息；消息传递的延时是一个固定值，或者有一个上限；相同发送者和接受者之间的消息是按序到达的；消息在传递过程中不会丢失，不会重复，不会产生错误。系统中的网络拓扑是静态的，且进程和共享资源都不会在站点间移动。

因为在集中式算法[6]中，中心站点周期性地收集各个站点发送来的进程资源关系信息来构建全局 WFG 图，当某些站点的位置发生改变时，中心站点和普通站点之间的通信将受到影响。而在边界追逐算法[2,7]中，主要依靠探针消息在 WFG 图中的传播来检测死锁。当网络中的进程和资源位置发生移动时，一方面消息传递不再是可靠的，传送给特定接收者的探针消息会因为接收者的位置的突然改变而丢失，而如果采取可靠协议的确认机制又会增加网络的通信开销；另一方面，网络中资源数目和位置的改变无法及时通知各个使用者，从而算法的执行效率会有所降低。考虑一些路径推进算法[1,11]，当本站点的进程移动到其他站点时，就会导致在此之前传送给其他站点的本地 WFG 图信息不再准确，从而各个站点都无法构建一个一致的全局 WFG 图。

由此可见，上述的传统算法虽然在普通的分布式环境下有较好的性能，但在应用到移动计算环境中时，性能便有所降低。

本文的工作就是针对上述问题，借助移动 Agent 技术，结合集中式与分布式算法的优点而设计了 Agent Guard 算法。

3 Agent Guard 死锁检测算法

3.1 系统模型和数据结构

我们假设分布式系统中有 n 个站点，表示为$\{S_i \mid 0 \leqslant i < n\}$。站点间通过消息传递进行通信。所有站点在物理上都是连通的，即任意两个站点间的通信都是可达的，并且可以通过某个路由算法选择最佳通信路径。每个站点上会初始化一个或多个进程，每个进程唯一标志为 P_i，不同的进程具有不同的优先级，进程在运行期间可能在站点间移动。系统中的共享资源分布在各个站点上，这些资源可以被本地或其他站点上的进程互斥使用。系统中的每个站点有一个站点管理器 SM(site manager)，SM 知道当前系统中的资源分布情况，并有一个接口与在本站点初始化的进程进行通信。

进程在运行期间会根据需求提出资源请求。我们为进程定义了两种状态，活跃态和阻塞态。进程 P_i 初始为活跃态，当 P_i 当前申请的资源被进程 P_j 占用时，P_i 就进入阻塞态。P_i 从阻塞态转为活跃态当且仅当 P_i 得到它所申请的资源。

系统中的每个站点的 SM 将会维护一个本地的 WFG 图，即 LWFG，LWFG 的数据结构如表 1 所示。

表 1 LWFG 的数据结构

Request Process ID	Block Process ID	Remote Site ID	Flag

其中，Request Process ID 代表了申请某个资源的进程号；Block Process ID 代表了当前占有该资源的进程号，即阻塞进程号；Remote Site ID 代表了阻塞进程所在的站点号；Flag 用来标记该条记录是否已被 Agent guard 读取。

Agent Guard 将会维护一个全局的 WFG 图，即 GWFG，GWFG 的数据结构如表 2 所示。

表 2 GWFG 的数据结构

Request Process ID	Request Site ID	Block Process ID	Block Site ID

其中，Request Site ID 表示申请资源的进程所在的站点号，其余各项与 SM 维护的项的意义相同。

在 Agent Guard 中维护一个双端队列 siteQueue，siteQueue 兼具队列和栈的功能，其中的元素是将要访问的站点的编号。另外 Agent Guard 还维护一个整型数组 site_visited[n]，用于标记当前各个站点的访问情况。有关 siteQueue 和 site_visited[n]的使用将在 3.3 节路线算法中描述。

当 Agent Guard 检测到死锁并选定被杀死的进程 P_i 后，会发送一个 victim 消息给 P_i 的初始化站点的 SM，由 SM 来通知该进程退出(abort)。“退出”的意义为：进程收回已发出的资源请求，释放占有的资源，回到初始状态后重新启动。

3.2 Agent Guard 算法

Agent Guard 算法的基本思想是：每个站点上的 SM 负责管理当前本地的资源，所有进程必须通过 SM 请求与释放资源。由此 SM 可构建 WFG 图。当有本地进程等待资源时，将该项信息记录在 LWFG 中；当本地进程请求的资源得到满足或请求取消时，则在 LWFG 中删除该项。SM 定期检查 LWFG，取出请求进程和阻塞进程都在本地站点的项来构建 WFG 图，一旦发现环的形成立刻通知死锁环中优先级最小的一个进程退出，从而解除本地死锁环。

系统中有一个移动 Agent 负责检测全局死锁环，我们将其命名为 Agent Guard。Agent Guard 遵循某个给定的移动算法访问系统中的各个站点，收集、合并及分析各个站点上的局部 WFG 信息，一旦发现死锁存在，按一定的解死锁算法选择某一个进程退出。Agent Guard 在系统中移动时会记录下进程以及资源的分布情况，以便在进程位置发生移动时准确进行定位。

可以看出，Agent Guard 扮演了两个角色，在系统中检测和解除死锁的同时它还能及时将进程和资源的数目以及位置

变化信息通知其他站点。算法中的主要技术难点是 Agent Guard 如何与所访问站点进行交互以收集信息并进行分析，以及如何为 Agent Guard 选择合适的路线算法。显而易见，采用适当的移动路线算法时，Agent Guard 能够快速准确地发现并解除死锁。如果路线算法不适当，会导致 Agent Guard 在系统中移动的次数增多，解锁效率却没有提高。为此本算法中为 Agent Guard 设计了广度优先，深度优先以及随机访问三种移动路线算法。

下文先就死锁检测和死锁解除两个方面描述本算法细节，再分别介绍广度优先，深度优先和随机访问三种路线算法的思想。

3.2.1 *死锁检测算法*

a)SM 部分的算法

当本地站点上有进程 P_i 在等待站点 S_j 上的进程 P_j 时，在 LWFG 图中添加如表 3 所示一项，Flag=1 表示该项等待 Agent Guard 读取。

表 3 本地 WFG 图中的一项

P_i	S_j	P_j	1

如果进程 P_i 经过一段时间后得到它请求的资源或取消资源请求时：

I. 若对应项 Flag=1，表明 Agent Guard 还未读取该信息，所以可以直接删除该项；

II. 若 Flag=0，代表 Agent Guard 已读取该信息，为保持与 Agent Guard 维护的信息一致，不能简单删除该项，而是置 flag 为−1，当 Agent Guard 下次读取时，便知道该信息应该删除。

如果进程 P_i 经过一段时间后离开本站点，有关进程 P_i 的项仍然保存在 LWFG 中以便 Agent Guard 读取该信息。

当 SM 收到 Agent Guard 发来的消息，告知在本站点初始化的进程 P_i 被选为 victim 时，SM 采取下列动作：释放 P_i 占有的资源，通知 P_i 退出，在 LWFG 表中删除该项。

b) Agent Guard 部分的算法

我们规定当系统初始化时，Agent Guard 总在系统的 0 号站点 S_0 上进行初始化。Agent Guard 首先读取 S_0 的 LWFG 信息，然后根据 3.3 节所述的路线算法 itinerary()来依次访问各个站点。每次移动到一个新的站点 S_i，Agent Guard 执行下列动作：

I. 对于 LWFG 表中 Flag=1 的项，读取之；若阻塞进程在站点 $S_j(i\neq j)$ 上且当前一轮遍历中还未访问过 S_j，S_j 进队列 siteQueue，以利于路线算法选择下一个站点；对于 LWFG 表中 Flag=−1 的项，在 Agent Guard 维护的 GWFG 图和站点维护的 LWFG 图中都删除之。

II. 若当前站点上没有任何进程在等待其他站点上的进程占有的资源，这就说明此刻本地进程都不可能陷入全局死锁环，若此时 siteQueue 为空，Agent Guard 会暂时在该站点上等待一段时间后再次读取信息，等待时间可以根据当前死锁发生的频率和网络的延时来设定。为了防止在同一个站点上等待多次，我们规定 Agent Guard 在同一个站点上只会连续等待一次。

III. 读取完信息后，Agent Guard 检查 GWFG 图，若有死锁环存在，则选择一个 victim 进程，发送 victim 消息给相应的 SM。

IV. Agent Guard 记录下本站点上进程以及资源的分布情况，以便在进程或资源的位置发生移动时准确进行定位，通知其他站点更新相应的信息。

3.2.2 *死锁解除算法* 通常的解除死锁的策略是选择陷入死锁环中一个进程作为 victim，使其退出，从而破坏进程间的循环等待关系。在本算法中，系统中的进程有不同的优先级，为了降低系统中发生死锁而造成的损失，选择环中一个优先级最低的进程作为 victim，将其进程号发送给该进程的初始化站点的 SM，由 SM 来释放它占有的资源，通知它退出。由于是由单个 Agent Guard 维护全局的 WFG 信息，因此不存在分布式算法中常有的同一个死锁环被不同的进程多次解锁的问题。

3.3 路线算法 itinerary()

Agent Guard 在各个站点间收集信息并分析来解除死锁，它的解锁效率是与它访问各个站点的路线紧密相关的。本文为 Agent Guard 设计了广度优先、深度优先以及随机访问三种路线算法。

在系统某个时刻，各个站点间由于进程的等待关系会形成一定的依赖关系，如图 1 所示的系统某时刻的 WFG 图可以抽象为图 2 所示的依赖图。所谓“广度优先”和“深度优先”都意指 Agent Guard 遍历依赖图的方法。Agent Guard 维护了 3.1 节中所述的双端队列 siteQueue 和整型数组 site_visited[n]以便于执行这两种遍历算法。siteQueue 既可作为队列，又可作为栈使用。site_visited[n]用于标记当前的一轮遍历中各个站点的访问情况。若站点 S_i 被访问过，则 site_visited[i]=1，反之为 0。

当 Agent Guard 在各个站点间移动时，由于进程也可能在站点间移动，因此需要注意的一点是：Agent Guard 是根据之前访问过的站点上保留的静态 LWFG 信息来产生依赖图并分析判断下一个将要移往的站点的，这样做的目的是为了让 Agent Guard 尽量按照进程间的依赖关系来移动，从而在有死锁环产生时，能较快地发现死锁。

随机访问则是随机挑选一个还未访问过的站点进行访问。

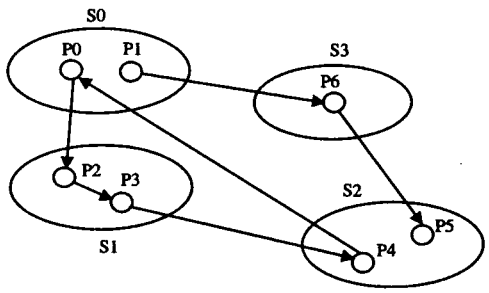

图 1 系统某时刻的 WFG 图

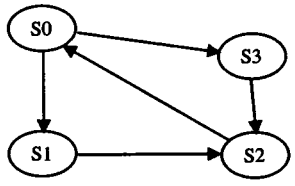

图 2 图 1 对应的依赖图

3.3.1 *广度优先算法* “广度优先”指 Agent Guard 在

遍历站点间的依赖图时采用的是广度优先遍历。以下就以图1、图2为例说明 Agent Guard 的广度优先移动路线算法：

图2是系统某时刻的依赖图，Agent Guard 在站点 S_0 初始化后，按照广度优先算法，依次访问站点 S_0、S_3、S_1、S_2，按照3.2节所述执行相应动作。对应于图1来看，Agent Guard 在访问完 S_2 时可以检测到死锁环 $P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0$。最后，Agent Guard 从 S_2 开始新的一轮遍历。

3.3.2 *深度优先算法* "深度优先"指 Agent Guard 在遍历站点间的依赖图时采用的是深度优先遍历。以下仍用图1、图2说明 Agent Guard 的深度优先移动路线算法。Agent Guard 在站点 S_0 初始化后，按照深度优先算法，依次访问站点 S_0、S_3、S_2、S_1。对应于图1来看，Agent Guard 在访问完 S_1 时可以检测到死锁环 $P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0$。最后，Agent Guard 从 S_1 开始新的一轮遍历。

3.3.3 *随机访问算法* Agent Guard 随机选择一个当前遍历中未访问过的站点作为下一个访问站点。

在前两种算法中，Agent Guard 基本上是按照站点间的有向边来移动的，从而在有死锁环产生时，能较快地发现死锁。总的来讲，前两种算法的性能要优于随机访问路线算法，并且当系统中形成的死锁环较短时，广度优先路线算法的性能略优于深度优先路线算法。

4 性能分析

4.1 环境设置

在 single resource 模型下，一个进程一次只能请求一个资源，在这个资源得到满足前，进程不能再提出申请。因此，一个进程在同一时刻最多只能陷入一个死锁环中。single resource 模型最简单且广泛使用，并可以根据环境很方便地扩展到其他几种模型。为简单考虑，在本文的算法模拟中，采用了 single resource 模型。

在本文的实验环境中，有足够多站点互联成一个网络，每个站点上都装有 IBM 公司开发的 Aglet 系统来提供移动 Agent 运行的平台。假设系统中有 m 个进程，每个站点上平均分布 C_p 个进程，站点是不动的，即网络拓扑是静态的，但是进程可以在站点间自由移动。系统中共有 N 个资源，随机分布在各个站点上。

由于站点是静止的，因此在 SM 之间交换的有关资源管理的消息传递是可靠的，即对于同一个目的地的消息是按序到达的，并且没有丢失和复制的情况发生。实验中假设消息从一个站点传到另一个站点平均需要 T_m 时间，站点内部的消息交换时间可以忽略。

因为进程的移动对死锁检测算法的影响主要体现在消息传递的可靠性方面，所以为简单考虑，本文的实验中用进程间消息的不可靠传递来模拟移动环境对死锁检测的影响。

传统的分布式算法 Chandy-Misra-Haas 算法[7]和 SET 算法[1]具有较好的性能，所以我们将 Agent Guard 算法的性能与两者在实验中作了比较。对这两种算法的消息传递模型假设如下：1)Chandy 算法中，死锁的检测主要是通过进程之间转发的 probe 消息来完成的，在移动环境下，进程的移动会造成 probe 消息产生一定的丢失率。2)SET 算法中，由于 global candidate 是在 SM 之间交换的，因此传递是可靠的。但是由于进程的移动，可能导致接收到的 global candidate 会因为本地某个进程的移动而无法与本地的 global candidate 连接，从而也对死锁检测产生影响。

一个进程 P_i 进入系统后，执行下列动作：等待 T_w 时间后，向资源 R_i 所在站点的 SM 发送请求，请求使用资源 R_i，若资源 R_i 空闲，进程 P_i 得到资源 R_i，使用一段时间 T_h 以后释放资源，等待 T_w 时间后再次提出新的资源请求；若资源 R_i 当前正在被进程 P_j 使用，则 SM 把进程 P_i 放入等待队列，标志 P_i 等待 P_j。

在这个过程中，进程释放资源后到提出新的请求之间的时间间隔 T_w 服从 $\alpha=1$、$\beta=2$ 的韦伯分布，进程请求的资源号 R_i 是 $1 \sim N$ 之间的随机整数，进程占有资源的时间 T_h 服从 $\alpha=1$、$\beta=10$ 的韦伯分布。另外，Agent Guard 在某个没有出度的站点上的等待时间为 T_{gwt}，Agent Guard 的路线算法采用了广度优先算法。

Chandy-Misra-Haas 算法和 SET 算法发起死锁检测的时间定为 T_{init}，由于进程间消息的不可靠传递，probe 消息会有0.1的丢失率，而 global candidate 消息由于进程的移动会有0.1的概率不能正确联接，其他参数设置与 Agent Guard 算法保持一致。

4.2 实验结果

在实验中，我们把 Agent Guard 算法的性能与 Chandy，SET 算法做了比较。系统中的每个进程有唯一的一个进程号，我们规定进程号较大的进程具有较小的优先级。表4列出了实验中系统参数的含义及设置。实验时系统中的进程数 m 从12递增到52。

表4 系统参数含义及设置

参数	含义	设置值
T_m	站点间的消息延时	2
C_p	每个站点平均分布的进程数	4
T_{gwt}	Agent Guard 在某个没有出度的站点上的等待时间	4
T_{init}	Probe、SET 算法发起死锁检测的时间	26
N	系统中的资源数	20
T_{total}	算法总的运行时间单位	10000

通常情况下，一个死锁检测算法的性能主要用三个指标来衡量[10,12]：

(1) 死锁持续时间。指从系统中产生死锁到该死锁环被解除的时间差。

(2) 伪死锁率。指算法报告一个事实上并不存在的死锁环的概率。

(3) 给系统增加的通信开销。由于死锁检测算法要求在进程之间以及站点之间交换一些数据，因此会给系统带来一些额外的通信开销。

因此，下文从这三个方面比较了三种算法的性能。

图3给出进程数增加时，三种死锁检测算法的伪死锁率。Agent Guard 算法的伪死锁率比较小，始终维持在0.2左右。Chandy 算法的伪死锁率维持在0.5以内。SET 算法的伪死锁率比较高，且呈现线性增长的趋势。通过比较可以发现 Agent Guard 算法的伪死锁率是很小的。

分布式系统采用死锁检测算法后会增加一定的通信开销。图4比较了分布式系统为检测死锁而分别采用三种算法后增加的通信复杂度。图4中三条曲线的含义分别为：随着进程数的增长，采用 Agent Guard 算法后相对于每个死锁环 Agent Guard 的平均移动次数，采用 Chandy 算法后系统中检测到每个死锁环平均需要交换的 probe 消息数目，以及采用

SET 算法后系统中检测到每个死锁环平均需要交换的 SET 消息数目。可以看出，Agent Guard 算法对系统的通信开销影响要小得多。

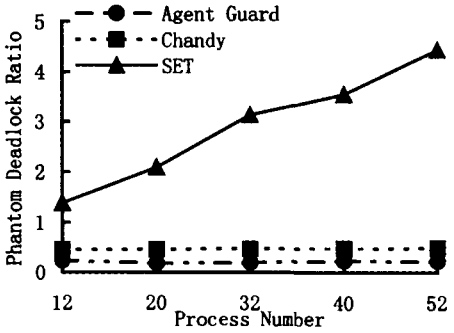

图 3　三种算法伪死锁率的比较

图 5 分析了当进程数增加时，系统中死锁的持续时间。可以看出对于 Agent Guard 算法，当系统中的进程数维持在 50 个以下时，死锁的持续时间与 Chandy，SET 算法相比是较短的。当进程数增多时，由于 Agent Guard 遍历各个站点所需的时间较长，从而导致死锁的持续时间也多较大增长。当 Agent Guard 主要应用于中小规模的分布式系统时，就可以保证较短的死锁持续时间。

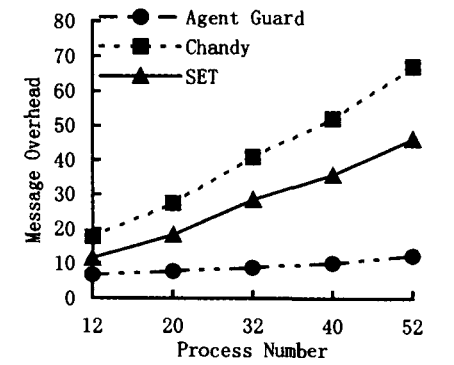

图 4　三种算法通信复杂度的比较

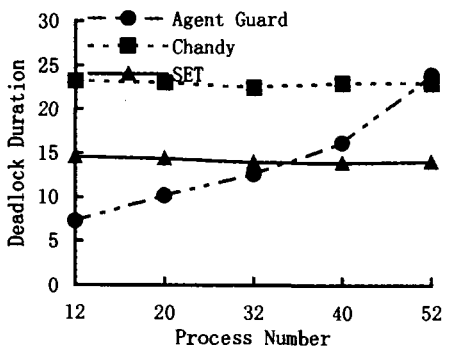

图 5　三种算法死锁持续时间的比较

结论　本文结合移动 Agent 技术，设计了分布式系统的死锁检测算法 Agent Guard。Agent Guard 算法综合了集中式解锁和分布式解锁的优点，在分布式系统中利用一个移动 Agent 遵循特定的路线在各个站点间移动并采集信息，从而准确地分析资源使用情况，及时检测到死锁并解除。

与传统的分布式死锁检测算法 Chandy 算法和 SET 算法相比，Agent Guard 算法的主要优点是：当进程移动时，能对进程准确定位，明显降低了死锁检测的伪死锁率，减少了系统为死锁检测和解除而增加的通信开销，并且在小规模的系统中，死锁的持续时间也有大幅度减少。

进一步的工作可以考虑对 Agent Guard 应用启发式策略，使它能够依据以往的解锁经验，适度增加在死锁发生频率较高的网络区域内的移动次数，使其在大规模的分布式系统中应用也能得到较短的死锁持续时间。另外还可以增加 Agent 的个数，通过多个 Agent 之间的相互协作来避免单个 Agent Guard 的单点失效问题，进一步提高算法性能。

参 考 文 献

1　Choudhary A N. Cost of Distributed Deadlock Detection: A Performance Study. In: Proc. of the 6th Intl. Conf. on Data Engineering, Los Angeles, California, USA, Feb. 1990. 174～181

2　Choudhary A N , Kohler W H, Stankovic J A, Towsley D. A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution. IEEE Transactions on Software Engineering, 1989, 15(1): 10～17

3　Lange D B, Oshima M. Seven Good Reasons for Mobile Agents. Communications of the ACM, 1999, 42(3): 88～90

4　Menasce D E , Muntz R R. Locking and Deadlock Detection in Distributed Databases. IEEE Transactions on Software Engineering, 1979, 5(3): 195～202

5　Knapp E. Deadlock Detection in Distributed Databases. ACM Computing Surveys, 1987, 19(4): 303～328

6　Ho, Ramamoorthy C V. Protocols for Deadlock Detection in Distributed Database Systems. IEEE Transactions on Software Engineering, 1982, 8(6): 554～557

7　Chandy K M, Misra J , Haas L M. Distributed Deadlock Detection. ACM Transactions on Computer Systems, 1983, 1(2): 144～156.

8　Sinha M K, Natarajan N. A Priority Based Distributed Deadlock Detection Algorithm. IEEE Transactions on Software Engineering, 1985, 11(1): 67～80

9　Roesler M, Burkhard W A. Resolution of Deadlocks in Object-Oriented Dstributed Systems. IEEE Transactions on Computers, 1989, 38(8): 1212～1224

10　Singhal M. Deadlock Detection in Distributed Systems. IEEE Computer, 1989, 22(11): 37～48

11　Obermarck R. Distributed Deadlock Detection Algorithm. ACM Transactions on Database Systems, 1982, 7(2): 187～208

12　Lee S, Kim J L . Performance Analysis of Distributed Deadlock Detection Algorithms. IEEE Transactions on Knowledge and Data Engineering, 2001, 13(4): 623～636

13　Park Y C, Scheuermann P, Tung H L. A Distributed Deadlock Detection and Resolution Algorithm Based on a Hybrid Wait-for Graph and Probe Generation Scheme. In: Proc. of the fourth Intl. Conf. on Information and Knowledge Management, 1995. 378～386