一种基于智能体的游戏消息公平处理方法*)

程卫星　郝爱民
(北京航空航天大学计算机学院　北京 100083)

摘　要　首先分析了现有网络游戏相关的消息处理方法,然后给出了分布式游戏服务器中一种基于智能体的消息处理结构,在此结构中,智能体处理游戏消息的算法可以实现一种公平的消息处理效果。通过选择离用户较近的服务器创建与用户直接通信的智能体,使得用户与该智能体间的网络时延抖动较小,不需要同步用户和服务器之间的时间就能够从游戏中获得一个相对公平的游戏效果。最后在模拟环境中给出了该算法的实验结果。

关键词　MMOG,公平,交互,智能体

An Agent-based Method for Fair Game Message Process

CHENG Wei-Xing　HAO Ai-Min
(School of Computing, Beihang University, Beijing 100083)

Abstract　An agent-based framework for game message exchange in distributed game servers is presented. Base on this framework, we propose message delivery algorithms that remove the unfair advantage that players with smaller message delays from the game server receive over players with large message delays from the server. The framework does not require assumptions of synchronized clocks at the players which is very difficult in internet, and user can connect to server with lower delay jitter in order to improve fairness.

Keywords　MMOG, Fairness, Interactivity, Agent

在大规模多用户同时在线的游戏世界中,不同用户与游戏服务器之间往往存在着不同的网络时延。不同用户在同一时间发出的响应消息有可能在不同的时间到达服务器。当一个用户由于距离服务器较远等因素导致消息传送时的时延较大时,相对于其它时延较小的用户来说,即使他的反应速度较快,也可能不能得到合理的回报,从而导致游戏结果的不公平。要使用户得到良好的交互体验,需要使用合理的设计,使得用户的输入能够得到及时、合理的响应。

由于单个服务器能够允许的同时在线用户数量有限,因此引入了多个服务器同时组成一个游戏世界[9,10],也有采用无服务器的分布式游戏[1,2]。前人的工作大部分集中在如何减少用户能体会到的消息响应时间,如用来补偿网络时延和丢包的 Dead-reckoning 算法[7,8],该算法利用最近接收到的几个状态消息来估计当前值,如果预测出来的值与实际值超过一定阈值,则传输一个包含实际位置值的消息用来更新当前值。

文[2,4,5,13]中提出了 Local Lag 以及其它类似的算法,用来保证游戏中的公平以及一致性问题。这些方法的主要思路是在产生和接收游戏事件时都加入适当的延迟,对于每个用户来说,根据他们和服务器之间时延的不同,所选取的延迟也不同。这些方法的目标就是在每个游戏事件发生后一段时间内,使得更新后的游戏状态同时显示在所有的用户屏幕中。Sysc-MS[9]提出了一种公平的消息传递服务,但是此服务需要两个前提,首先是所有用户和服务器的时间必须得到同步,然后是从服务器到各用户间的时延要求能够被精确估计到。文[10]根据用户接收到状态更新消息后发出响应消息的反应时间对响应消息进行排序后由游戏服务器依次处理,不需要在同一时间发送状态更新消息给用户。但是,此框架只适合在客户机/服务器模型中使用,而且由于需要在客户端计算用户的响应时间,因此无法很好地避免用户使用作弊手段以减少响应时间的值,并且在非一致视图下很难保证公平问题。文[14]一方面通过丢弃过时的不影响游戏状态的数据来提高消息传送的速度,另一方面采用了类似于 Local Lag 的算法来保证游戏的公平性,在提高游戏响应速度的同时保证了游戏的公平性。但是此算法也要求所有服务器和用户之间的时间都得到准确同步,这在实际中非常困难。

本文使用智能体进行消息的分发管理,由智能体负责更新各用户替身(Avatar)所需要的游戏状态数据。在处理用户的响应消息时,考虑了不同用户和服务器之间存在不同网络时延的特点。在保证响应速度的情况下,一方面使游戏状态更新消息在不同的时间内到达游戏客户端,另外使用户发出的响应消息以到达服务器的时间为序进行处理,避免了传统网络游戏中由于不同用户到游戏服务器的网络时延不一致而导致的不公平现象,从而提高了游戏的公平性。

1　基于智能体的游戏消息处理

1.1　基本定义

定义 1　响应消息(AM)　游戏客户端根据用户的输入动作,发给游戏服务器的消息。

定义 2　状态更新消息(UM)　游戏服务器在接收到游戏消息之后,对消息进行处理,从而改变游戏的状态,然后把更新后的游戏状态通知给用户。

*)基金项目:国家高技术研究发展计划(2004AA115130)。程卫星　博士生,主要研究方向为分布式虚拟现实;郝爱民　博士,教授,主要研究方向为虚拟现实。

定义 3 消息响应时间(ΔT) 从用户发出响应消息，一直到接收到对应的状态更新消息的时间差。对于同一个响应消息，可能有多个用户接收到对应的状态更新消息，由于网络条件等差异性因素的存在，不同用户对应的消息响应时间可能不一致。

1.2 分布式游戏服务器中的消息分发

在分布式游戏服务器结构中，每个服务器只需要维护游戏世界中某一范围内的游戏状态。用户在登录游戏世界时，选择一个最近的服务器进行连接，并在所连接的服务器上生成一个智能体(User Agent，简称 UA)，该智能体负责接收用户的响应消息，并发送最新的游戏状态给用户。

一些基于分布式的游戏使用了广播的方式，如 MiMaze[2]，把每个用户的状态都发送给其他所有用户，从而浪费了不必要的网络带宽。本文使用了移动智能体对游戏的数据分发进行管理，可以使每个用户只接收到自己感兴趣的数据。

消息分发流程如图 1 所示。在消息分发过程中，主要涉及到以下几种对象：

(1)Distributed Game Server(DGS)：分布式游戏服务器，每个游戏服务器负责游戏世界中一定范围内的游戏状态，所有对游戏状态的直接更新都由 DGS 完成。

(2)User Agent(UA)：负责接收用户的响应消息，对响应消息按游戏逻辑进行处理后发消息给对应的 DGS，并发送最新状态更新消息给用户。

(3)Data Agent(DA)：由 UA 发送的驻留在各个游戏服务器的移动智能体，负责收集最新的游戏状态，然后发送给 UA。

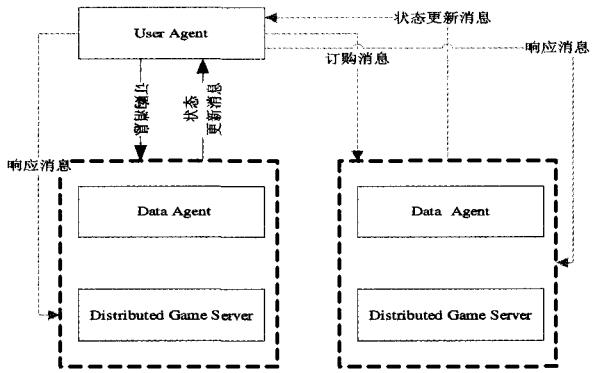

图 1 基于智能体的消息分发

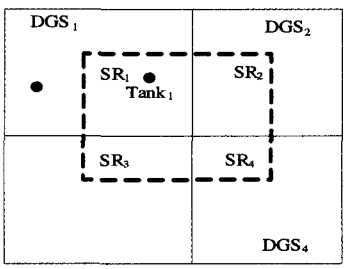

图 2 订购条件所处区域

1.3 移动智能体的管理

定义 4 订购请求消息 SubscribeRequestMsg 是一个二元组。SubscribeRequestMsg = {userAgent, region}，其中 userAgent 表示发起订购请求的智能体，region 表示订购对象所在区域。

系统通过 Data Agent 来发送最新的游戏状态消息。User Agent 首先利用订购表达式表示出对游戏中某些对象属性的兴趣，然后发送 Data Agent 到感兴趣对象所在的游戏服务器。当游戏服务器负责的对象状态发生变化时，Data Agent 对游戏状态数据进行过滤，并把过滤后的信息发送给 User Agent。同时，当 User Agent 改变订购关系时，每个 Data Agent 的内部过滤参数都会保持更新。

User Agent 利用订购表达式发送智能体的算法如下所示：

```
//发送订购消息
UserAgent. sendSubscribeRequestMsg(){
  SubscribeRequestMsg subMsg = new SubscribeRequestMsg(this,
region);
  //通过所在节点送
  node. sendMessage(subMsg);
}
//订购消息到达目标节点
SubscribeRequestMsg. arrivedAtDestination(Node localNode){
  if(localNode. containRegion(this. region){
  //此方法将回调方法 UserAgent. onSubscribed
  localNode. subscribed(this. userAgent);
}
else{
  List regions = this. region. split(localNode. region);
  foreach(Region region:regions){
     SubscribeRequestMsg subMsg = new SubscribeRequestMsg
(this. userAgent, subRegion);
    localNode. sendMessage(subMsg);
  }
 }
}
//发送智能体 DA 到目标节点
UserAgent. onSubscribed(Node node){
  //DA 中包含过滤条件和目标节点信息
  DataAgent dataAgent = new DataAgent(filters, node);
  daList. add(dataAgent);
  dataAgent. depart();
}
```

User Agent 首先根据订购表达式所对应的区域创建一个 SubscribeRequestMsg 消息，然后发送此消息到订购区域中心点所在的节点。当此消息到达目标节点以后，节点将回调消息的 arrivedAtDestination 方法。此时，如果节点所负责区域完全包含该消息对应的区域，则调用 subscribed 方法。此方法最终触发 User Agent 的 onSubscribed 方法，从而发送一个包含订购条件的智能体到目标节点；如果节点负责区域不能完全包含消息所对应的区域，则对消息所对应区域进行分割，然后根据分割的区域再次发送 SubscribeRequestMsg 消息。在进行区域分割时，沿着当前节点所负责区域的边界进行分割。

下面我们结合一个具体的例子来说明上述算法。假设游戏世界由四个 DGS 负责，游戏中有两辆坦克，如图 2 所示。User Agent 首先声明其订购条件为虚线区域内的坦克的位置，然后发送 SubscribeRequestMsg 消息。由于订购区域中心点位于 DGS_1，因此消息将到达 DGS_1 所在节点。这时，由于 DGS_1 负责区域不能完全包含订购区域，因此订购区域将沿着节点 1 的边界被分割成四个区域 SR_1、SR_2、SR_3 和 SR_4，然后发送对应各个分割后区域的 SubscribeRequestMsg 消息。当四个消息分别到达各个节点后，由于其订购区域都包含在节点所负责的区域之内，所以节点通过 subscribed 方法发送消息到 User Agent，从而触发其 onSubscribed 方法。因此，User Agent 将给四个节点分别发送一个包含订购条件的智能体。当 $Tank_1$ 和 $Tank_2$ 的位置发生变化时，由于 $Tank_2$ 所处位置刚好在 DA_1 的订购范围之内，符合订购条件，$Tank_2$ 则由于处在订购范围之外不符合订购条件。所以 DA_1 对 DGS_1 中的游戏状态进行过滤之后，只把符合条件的 $Tank_1$ 的最新位置发送给 User Agent。

1.4 消息传递算法

本文假设游戏状态由一系列对象以及对象的状态所组成。游戏状态的变化可能由于以下三个原因组成：删除对象、增加对象或者对象状态的变化。为了描述方便，定义以下几个符号表示：

(1)$UM_{i,q}$，分布式游戏服务器 DGS_q 负责的游戏状态第 i 次发生变化时，Data Agent 所发送的游戏状态消息，各个 Data Agent 会根据订购条件对消息内容进行相应的过滤；

(2)$AM_{j,k}$，用户 P_j 发出的第 k 个响应消息；

(3)RT_j^q，用户 P_j 接收到 $UM_{i,q}$ 时的物理时间；

(4)$AT_{j,k}$，响应消息 $AM_{j,k}$ 到达 User Agent 的时间，游戏服务器执行消息时将以此时间为顺序。由于在实际的网络条件中，后发的消息可能先于上一个消息到达，因此在 User Agent 处对未按顺序到达的消息的时间将进行重新赋值，详见本文后面部分；

(5)$ST_{j,k}^q$，$AM_{j,k}$ 经过 User Agent 处理后生成的更新消息到达分布式游戏服务器 DGS_q 的时间；

(6)$DT_{j,k}^q$，DGS_q 处理完 $AM_{j,k}$ 对应的更新消息后，Data Agent 发送状态更新消息的时间；

(7)$UT_{n,j,k}^q$，DGS_q 处理完 $AM_{j,k}$ 对应的更新消息后，UA_n 接收到对应状态更新消息的时间；

(8)$UST_{n,j,k}^q$，UA_n 给用户 P_n 发送状态更新消息时的时间；

(9)W_j，UA_j 在接收用户 P_j 的响应消息时所设置的等待超时时间，超过这个时间的响应消息将被丢弃。

在实际环境中，由于时延抖动、用户到服务器之间的时延差异等因素存在，很难达到理想的公平条件。为了衡量网络环境中游戏消息传递的效果，我们给出以下几个定义：

定义 5 网络公平(Network fairness) 为了实现网络游戏中的网络公平，要求保证所有的用户，不管他们的网络条件如何，在游戏中都有同样的获胜概率，这是一种理想状态下的公平。

由于实际环境的复杂性，要求消息在处理时的次序跟理想状态相比一样是无法实现的，也就是无法实现真正意义上的网络公平，因此给出下面这个定义。

定义 6 消息公平 在基于分布式游戏服务器的网络游戏中，如果消息处理满足以下三个条件，则认为网络游戏实现了消息公平。我们用→表示服务器 DGS_q 在执行响应消息时的前后关系。

(1)对于任意 k、l 以及 $n\neq j$，用户 P_j、P_n 在接收到 $UM_{i,q}$ 后，分别在 σ_j 和 σ_n 时间后作出响应(σ_j 为用户 P_j 的反应时间)，发出响应消息 $AM_{j,k}$ 和 $AM_{n,l}$ 后，满足 $|(AT_{j,k}-AT_{n,l})+(\sigma_n-\sigma_j)|<\theta$，$\theta$ 为一常数，不同游戏对 θ 的大小要求可能不一样。因为要求同时接收到响应消息不现实，所以只要满足在一定时间差内都接收到响应消息就认为是公平的；

(2)对于任意 k 和 $l>0$，存在 $AM_{j,k}\rightarrow AM_{j,k+1}$；

(3)对于任意 k、l 以及 $n\neq j$，如果 $AT_{j,k}<AT_{n,l}$，则 $AM_{j,k}\rightarrow AM_{n,l}$。

用户在发送响应消息时，消息包含序号信息 $ID_{j,k}$，在同一用户中，此 ID 是一个唯一递增的数值。响应消息到达 User Agent 之后，User Agent 根据 $ID_{j,k}$ 对消息进行排序。如果消息超时，则丢弃。发送消息给游戏服务器时，加入相对于每一个分布式游戏服务器的序号信息，算法如下：

```
UserAgent. processActionMessage(ActionMessage newMsg){
  if(! isTimeout(newMsg)){
    if (lastSendMsgId==(newMsg. id-1)){
      newMsg. arriveTime = now;
      processMessage(newMsg);
}
else{
  //调整消息到达时间 AT
  List nextMsgs = getMsgsIdLargeThan(newMsg. id);
  if(0==nextMsgs. size())
     newMsg. arriveTime = now;
  else
     newMsg. arriveTime = minArriveTimeInMsgs(nextMsgs);
  //加入消息队列中
  msgPool. add(newMsg);
 }
}
}
//消息队列处理线程
ActionMessageProcessThread. run(){
  while(true){
     ActionMessage msg = pool. first();
     //消息是否可以处理，如到达超时时间
     if (canProcess(msg)){
       processMessage (newMsg);
       msgPool. remove(newMsg);
     }
     //等待下一个消息到达队列或者超时时间到来
     wait();
  }
}
```

UA 接到消息后，首先判断消息是否超时。如果超时，则丢弃消息。如果之前的消息都已经到达，则直接根据游戏逻辑进行处理，否则以 $ID_{j,k}$ 为序把消息加入到消息队列中。消息的到达时间(AT)一般赋值为当前时间，对于消息队列中存在 ID 大于当前消息 ID 的情况，则应使到达时间值等于这些消息中最小的到达时间。处理响应消息时，根据游戏逻辑可以确定发给游戏服务器的消息，假设消息需要发送到游戏服务器 DGS_q，则发送消息时加入序号信息 $SID_{j,k}^q$，此序号对于每对 User Agent 和 DGS_q 来说是唯一递增的。

游戏服务器在接收到消息之后，需要计算消息执行时间(ET)，再对消息按 AT 值排序，然后在消息执行时间到来时按次序处理消息。每次有新消息到来时，消息队列中所有消息的 ET 值都需要重新计算。

1.5 响应消息处理算法

为方便起见，我们用 M_1，M_2，…，M_n 表示游戏服务器中的消息集合，$P(M_1,M_2,\cdots,M_n)$ 表示该消息集合对应的所有用户。T_q 表示 DGS_q 中所有 Data Agent 对应的用户集合，$\tau_{i,q}$ 表示 UA_i 所在服务器和 DGS_q 之间的超时时间，AT_k 表示消息 M_k 经过调整后的到达 UA_k 的时间。消息在服务器消息队列中的位置按 $\langle AT_{j,k},SID_{j,k}\rangle$ 进行排序。消息的执行时间 $ET(M_{j,k})$ 可以使得按非公平次序到达的消息能够以公平的次序得到执行。

首先假设用户发出的响应消息能够在公平的时间内到达 UA，也就是能够满足条件(1)。在这种情况下，为了描述方便，我们从最简单的条件出发来计算消息执行时间，然后逐步增加限制条件。

(4)假设用户发送的响应消息按顺序到达 User Agent，而 User Agent 发出的消息也按顺序到达游戏服务器。

引理 1 在满足条件(4)时，取 $ET(M_{j,k})$ 值如下，则可以满足条件(1)和(3)。

$$ET(M_{j,k})=AT_{j,k}+\max_{\{m\in T_q-P(M_1,M_2,\cdots,M_n)\}}(\tau_{m,q})$$

证明：因为响应消息按顺序到达，对于任意 $AM_{j,k}$，如果 $AM_{j,k+l}$ 在 $AM_{j,k}$ 执行前到达当前队列，由于 $AT_{j,k}<AT_{j,k+l}$，因此 $ET(M_{j,k})<ET(M_{j,k+l})$，所以此时能够满足条件(2)。下面证明条件(3)。

因为 $ST_{n,l}\leqslant AT_{n,l}+\tau_{n,q}$，所以取 $ET(M_{j,k})=AT_{j,k}+$

$\max_{\{m\in T_q-P(M_1,M_2,\cdots,M_n)\}}(\tau_{m,q})$时，能够满足条件 $\forall j,k,n,l|AT_{j,k}>AT_{n,l}\Rightarrow ET(M_{j,k})>ST_{n,l}$。此时一方面能够保证 $AM_{n,l}$能够在 $AM_{j,k}$被执行前到达游戏服务器，另外在 $AM_{n,l}$到达游戏服务器时，由于 $AT_{j,k}>AT_{n,l}$，因此满足 $ET(M_{j,k})>ET(M_{n,l})$，从而满足条件(3)。证明完毕。

此时，在计算执行时间时只需要考虑那些已经到达 User Agent 但还没到达游戏服务器的消息。利用此方法计算出的执行时间可以保证出现新的消息时，现有消息的位置相对不变。

(5)假设用户发送的消息按顺序到达 User Agent，而 User Agent 发出的消息不按顺序到达游戏服务器。

当 User Agent 发出的消息不按顺序到达时，我们使用序号 $SID^q_{j,k}$对来自 UA_j 的消息进行排序，从而判断较早前来自于 UA_j 的消息是否都已经到达。

用 $Q(M_1,M_2,\cdots,M_n)$表示消息 $M_1,M_2,\cdots,M_n$ 中的一个消息子集，此消息子集中的消息排在队列的最前面，并且都已经按顺序到达，$P(Q(M_1,M_2,\cdots,M_n))$表示消息集合 Q 对应的所有用户。

引理 2 在满足条件(5)时，取 $ET(M_{j,k})$值如下，则可以满足条件(2)和(3)。

$$ET(M_{j,k})=AT_{j,k}+\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(\tau_{m,q})$$

证明：同样地，当 $M_{j,k}$到达服务器之后，计算执行时间时需要满足以下条件，才能满足条件(3)。

$$\forall j,k,n,l|AT_{j,k}>AT_{n,l}\Rightarrow ET(M_{j,k})>ST_{n,l}$$

对于任意的 n 和 l，如果 $AT_{j,k}>AT_{n,l}$，那么消息到达游戏服务器的时间满足 $ST_{n,l}\leqslant AT_{n,l}+\tau_{n,l}$，由于 $P_n\in T_q-P(Q(M_1,M_2,\cdots,M_n))$，所以 $\tau_{n,q}\leqslant\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(\tau_{m,q})$。因此容易证明 $ET(M_{j,k})$能够满足条件(2)和(3)。

此时，计算执行时间时除了需要考虑那些还没有发出响应消息的用户外，还需要考虑到目前为止消息还没按顺序到达游戏服务器的用户。

(6)假设用户发送的消息不按顺序到达 User Agent，而 User Agent 发出的消息也不按顺序到达游戏服务器。这种情况是实际网络环境中存在的。

定理 1 在满足条件 C 时，取 $ET(M_{j,k})$值如下，则可以满足条件(2)和(3)。

$$ET(M_{j,k})=AT_{j,k}+\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(\tau_{m,q})+\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(W_j)$$

证明：由于用户发送的响应消息首先要在 User Agent 处排序之后才依次处理，假设 $AM_{n,l-1}$(含)之前的消息都已经到达，并且 $AM_{n,l+1}$也已经到达，而 $AM_{n,l}$还没到，这时 UA_n 将最多等待 W_n，也就是说 User Agent 最迟将在 $AT_{n,l+1}+W_n$ 时处理 $AM_{n,l+1}$，然后发对应的消息给游戏服务器。如果 $AM_{n,l}$在超时时间到来之前被接收，那么最迟也在 $AT_{n,l+1}+W_n$ 时被 User Agent 处理。

此时，只要在 $ET(M_{j,k})=AT_{j,k}+\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(\tau_{m,q})$的基础上，再等待 $\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(W_j)$，取执行时间值如下：

$$ET(M_{j,k})=AT_{j,k}+\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(\tau_{m,q})+\max_{\{m\in T_q-P(Q(M_1,M_2,\cdots,M_n))\}}(W_j)$$

此时，容易证明，对于所有的 n 和 l，如果 $AT_{j,k}>AT_{n,l}$，那么计算出来的执行时间就可以满足条件(2)和(3)。

1.6 状态更新消息处理算法

记用户 P_j 到 UA_j 的平均时延为 MT_j，UA_j 到游戏服务器 DGS_q 的平均时延为 $SMT_{j,q}$。

上一小节假设用户发出的响应消息能够在公平的时间内到达 UA，这在实际环境中可能无法得到满足。因此此时条件(1)就可能无法得到满足。状态更新消息处理算法的目的就是通过调整状态更新消息的发送时间，使其能够满足条件(1)。

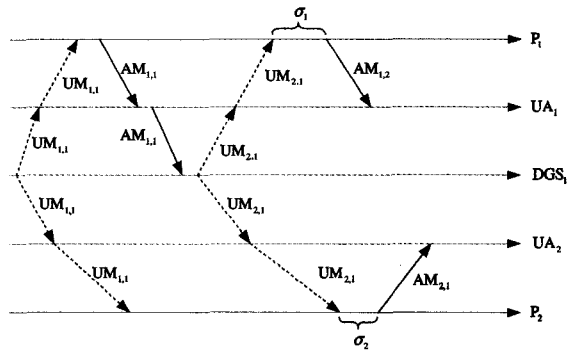

图 3 未考虑时延不一致的消息传递过程

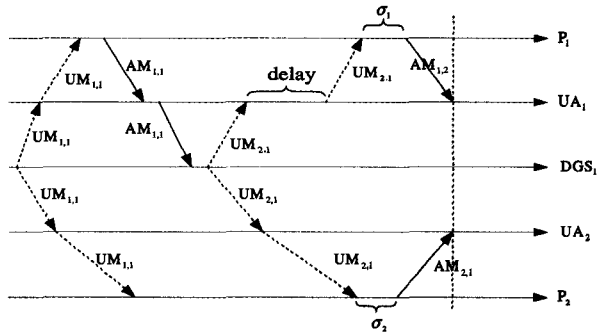

图 4 加入 cslag 后的消息传递过程

在图 3 中，用虚线表示状态更新消息，实线表示响应消息，DGS_1 中初始状态为 $S_1=\{tank_1,tank_2,tank_3\}$，$P_1$ 发出响应消息 $AM_{1,1}=\{Add\ tank_4\}$。当消息到达 DGS_1 后，游戏状态变为 $S_1=\{tank_1,tank_2,tank_3,tank_4\}$。$P_1$ 和 P_2 在接收到状态更新消息后，分别在经过反应时间 σ_1 和 σ_2($\sigma_1>\sigma_2$)后发出响应消息 $AM_{1,2}=\{Remove\ tank_4\}$、$AM_{2,1}=\{Remove\ tank_4\}$。由于时延不一致，$P_1$ 发出的响应消息早于 P_2 发出的响应消息到达各自的 UA，从而导致了不公平。

如果 UA_1 在接收到状态更新消息时，延迟一段时间后发送给 P_1，P_1 和 P_2 以同样的反应速度发出的响应消息就可能同时到达 UA，如图 4 所示。加入延迟是因为游戏服务器到 UA 之间以及用户到 UA 之间存在不同网络时延。对于响应消息 $AM_{j,k}$在 DGS_q 引起的状态更新消息，UA_m 在发送时的时间值为

$$UST^q_{m,j,k}=DT^q_{j,k}+\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})+2(\max_{i\in T_q}MT_i-MT_m)$$

其中，$\eta_{i,q}$为游戏服务器 DGS_q 到 UA_i 之间可能存在的最大时延与平均时延的差值，记 $cslag^q_{m,j,k}=\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})+2(\max_{i\in T_q}MT_i-MT_m)$。

定理 2 如果每个用户到 UA 之间的时延都没有抖动，那么取 $UST^q_{i,j,k}=DT^q_{j,k}+cslag^q_{m,j,k}$时，各个用户在经过一定的反应时间后发出的响应消息能够在公平的时间内到达 UA，此时取 $\theta=0$，条件(1)也将得到满足。

证明：由于

$$\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})+2(\max_{i\in T_q}MT_i-MT_j)\geqslant$$

$\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})$，

因此 $UST^q_{i,j,k} \geqslant DT^q_{j,k} + \max_{i\in T_q}(SMT_{i,q} + \eta_{i,q}) \geqslant UT^q_{i,j,k}$，所以状态更新消息发送时间的取值是可行的。

对于任意 $n\neq m$，用户 P_m、P_n 接收到 $UM_{i,q}$，分别在 σ_m 和 σ_n 后发送响应消息 $AM_{m,r}$ 和 $AM_{n,s}$，那么

$$AT_{m,r}=UST^q_{m,j,k}+MT_m+\sigma_m+MT_m$$
$$=DT^q_{j,k}+\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})+2\max_{i\in T_q}MT_i+\sigma_m$$

同理，

$$AT_{n,s}=DT^q_{j,k}+\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})+2\max_{i\in T_q}MT_i+\sigma_n$$

因此，$|(AT_{m,r}-AT_{n,s})+(\sigma_n-\sigma_m)|=0$，证明完毕。

用户发出一个响应消息后，需要能够尽快看到动作执行后的效果。一般来说，在实时多人在线网络游戏中，消息响应时间为 100ms 被认为是可以接受的，对于非实时的 RPG 游戏来说，还可以承受更高的时延，达到 200ms。

对于用户来说，为保证消息响应时间，可接受的 cslag 值如下：

$$cslag^q_{m,j,k}\leqslant\Delta T-(UT^q_{m,j,k}-AT_{j,k})-MT_m-MT_j$$

其中，ΔT 为游戏中设定的用户能够接受的消息响应时间。因此，在保证响应时间的前提下，cslag 取值如下：

$$cslag^q_{m,j,k}=\min(\max_{i\in T_q}(SMT_{i,q}+\eta_{i,q})+2(\max_{i\in T_q}MT_i-MT_m),\Delta T-(UT^q_{m,j,k}-AT_{j,k})-MT_m-MT_j)$$

2 实验及结果

2.1 实验设计

实验中，我们使用了 9 台 HP ProLiant DL360 G4 服务器：CPU 为 Xeon 3.0G * 2，内存为 4G，3 块千兆网卡，操作系统为 Redhat Linux AS 4，并使用局域网中的 NTP 服务器进行时间同步，各机器在实验期间的时间误差小于 1ms。五台游戏服务器组成一个游戏世界，还有一台安装了 NISL[15] 来模拟游戏中的各种网络环境，除了另外三台运行游戏客户端软件，五台游戏服务器也运行着游戏客户端。

NIST 网络中各机器之间的时延设置如下：每台服务器都有单项时延分别为 5ms、10ms、18ms、25ms 的机器连接，单项时延抖动值分别为 1ms、2ms、3ms 和 4ms。源机器到目标机器之间的单向时延和目标机器到源机器的单向时延一样。

不失一般性，假设所有游戏事件的处理都在服务器 DGS_0 中进行。每台服务器连接着 4 个用户，用户到服务器的平均时延取值范围为 5ms，10ms，18ms 和 25ms。

我们还实现了 bucket synchronization(简称 bucket)方法进行对比。和普通 bucket 方法相比，除了接收响应消息时加入同步时延(synchronization delay)之外，在发送状态更新消息时也加入了一定的时延，目的是为了所有用户能够同时接收到更新消息。在实验中我们使用了以下两个指标：

平均响应时间：消息响应时间的平均值；

消息公平率：符合消息公平三个条件的消息在所有响应消息中的比例；

每个用户每隔 300ms 发出一个响应消息，用户在发出响应消息后，如果过了平均时延没有再发出响应消息，则程序自动发送一个心跳响应消息给服务器。Bucket 算法中，接收消息的同步时延为 2 倍的平均时延，发送更新消息时的时间为：更新消息在游戏服务器端的发送时间＋服务器间最大时延＋用户最大平均时延－当前 UA 对应的用户平均时延。

2.2 实验结果及分析

2.2.1 平均响应时间

图 5 中，横坐标表示测试中最大的用户平均时延，纵坐标表示用户的平均响应时间，cslag 在 ΔT 取值为 80ms、110ms 和 150ms 时分别测试。可以看出，最大用户平均时延较大时，此时减少 ΔT，用户的平均响应时间随之减少。但是我们在后面可以看到，这是以降低消息传递公平率为代价的。当最大用户平均时延较小时，减少 ΔT 对用户的平均响应时间影响不大，这是因为此时 ΔT 还没能影响 cslag 的取值。对于 bucket 算法来说，其响应时间相对 cslag 算法来说要大，这是因为 bucket 算法的延迟取决于时延最大的部分。

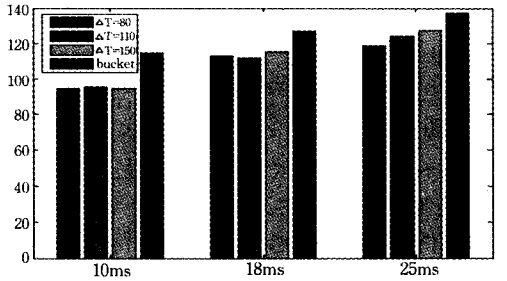

图 5 平均响应时间

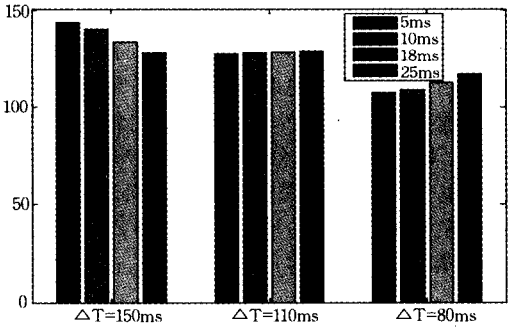

图 6 cslag 算法中不同用户的平均响应时间

图 6 中，分别位于 DGS_1、DGS_2、DGS_3、DGS_4 所在服务器上的四个客户，连接到 DGS_0 所在服务器时，具有从 5ms 到 25ms 等不同的平均时延。图中显示了这四个用户在不同 ΔT 下的平均响应时间。可以看出，当 ΔT 足够大时，为了实现消息的公平传送，发送状态更新消息时，时延小的用户比时延大的用户要晚些接收到更新消息，因此此时平均时延越小，响应时间越大。随着 ΔT 的减小，为了保证响应时间，UA 在接收到状态更新消息的时候，等待的时间越来越小，因此时延小的用户的平均响应时间逐渐接近直到小于时延大的用户。

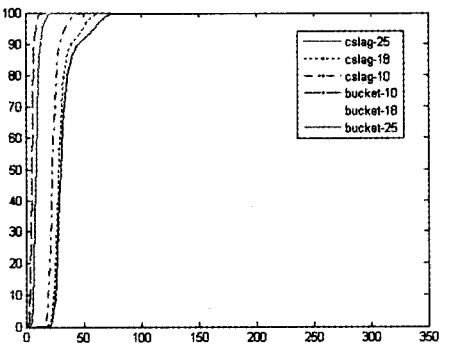

图 7 不同最大客户平均时延下的公平率(ΔT=150ms)

2.2.2 消息公平率

用户在收到状态更新消息后以一定的反应时间(从

150ms 到 154ms 之间随机选择)发出响应消息,因此我们在收集到所有针对该状态更新消息的响应消息后,就可以计算出每一个$|(AT_{j,k}-AT_{n,l})+(\sigma_n-\sigma_j)|$的值,并取其中的最大值来判断该消息是否在所有用户之间实现了消息公平。

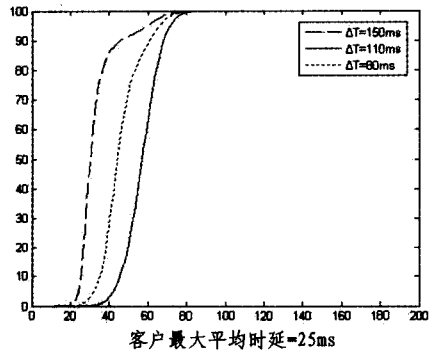

图 8 不同 ΔT 下的消息传送公平率

图 7 中,cslag-25ms 表示 cslag 算法在最大客户平均时延为 25ms 下的统计结果,其它类似。可以看出,用户离服务器越近,相同条件下的公平率越高,这是因为我们能够精确同步各个服务器的物理时间,使得算法能够使用 η 来补偿服务器的时延抖动,从而使得消息公平率只跟客户端和 UA 之间的时延抖动有关。因此,我们可以通过增加服务器的方法来减少用户与服务器之间的时延抖动,从而提高消息公平率。Bucket 算法由于在处理消息时以用户发送响应消息的时间为序进行处理,它的公平与否取决于状态更新消息是否同时到达客户端。因此该算法的公平率只受用户单向时延抖动的影响,而 cslag 算法要受到用户双向时延抖动的影响,所以 bucket 算法在相同条件下的消息公平率要高于 cslag 算法。但是 Bucket 算法要求同步客户端电脑时间,这在实际环境中很难做到。

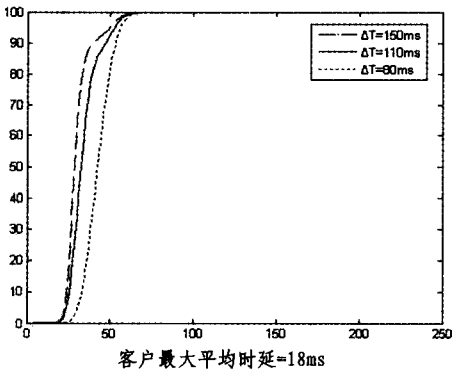

图 9 不同 ΔT 下的消息传送公平率

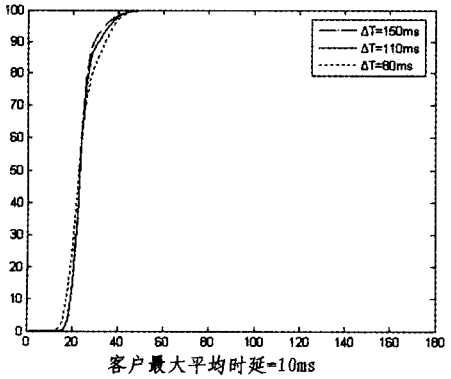

图 10 不同 ΔT 下的消息传送公平率

图 8～10 为不同 ΔT 下的消息传递公平率。可以看出,在最大客户平均时延较大时,ΔT 越大,消息公平率越高。而在最大客户平均时延较小时,消息公平率与 ΔT 的变化关系不大,这是因为时延较大时,ΔT 越大,cslag 的取值受限于 ΔT 的概率越小,因此消息公平率就越高。而当最大客户平均时延越小时,cslag 的取值受限于 ΔT 的概率也越小。

结论 本文提出了分布式游戏服务器中一种基于智能体的消息分发结构,并在此基础上提出了一种公平的消息传递算法 cslag,该算法一方面考虑到了用户的反应时间,使得反应快的用户具有较高的获胜概率,另一方面又避免了文[10]中对某些消息的不平等处理。

使用 cslag 算法时,服务器之间的时延抖动不影响消息传递的公平性,因此通过分布多台服务器,使得用户连接到离自己较近的一台服务器,减少到服务器的时延抖动,还可以提高游戏的公平性。

参 考 文 献

1 Diot C, Gautier L. A distributed architecture for multiplayer interactive applications on the internet. IEEE Networks, Jul./Aug. 1999,13:6～15

2 Gautier L, Diot C. Design and Evaluation of MiMaze, a Multiplayer Game on the Internet. In: Proc. IEEE Multimedia (ICMCS'98), Austin, TX, USA, 1998. 233～236

3 Vogel J, Mauve M. Consistency control for distributed interactive media. In: ACM MM'01Sep./Oct. 2001. 221～230

4 Mauve M, Vogel J, Hilt V, et al. Local-lag and timewarp: Providing consistency for replicated continuous applications. IEEE Transactions on Multimedia, Feb. 2004, 6:47～57

5 Pantel L, Wolf L C. On the Impact of Delay on Real-time Multiplayer Games. In: Proc. of ACM NOSSDAV'02, May 2002

6 Farber J. Network Game Traffic Modelling. In: Proc. of NetGames2002, Apr. 2002

7 Mauve M. Consistency in Replicated Continuous Interactive Media. In: Proc. of the ACMConference on Computer Supported Cooperative Work (CSCW'00), 2000. 181～190

8 Singhal S K, Cheriton D R. Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality. Teleoperators and Virtual Environments, 1995,4(2): 169～193

9 Lin Y, Guo K, Paul S. Sync-MS: Synchronized Messaging Service for Real-time Multi-player Distributed Games. In: Proc. of 10th IEEE International Conference on Network Protocols (ICNP), Nov. 2002

10 Guo K, Mukherjee S, Rangarajan S, et al. A fair message exchange framework for distributed multi-player games. In: Proc. ACM NetGames'03, May 2003. 21～33

11 Zander S, Leeder I, Armitage G. Achieving Fairness in Multiplayer Network Games through Automated Latency Balancing. In: Proc. of ACM SIGCHI ACE2005, Valencia, Spain, 2005. 117～124

12 Pantel L, Wolf L C. On the Impact of Delay on Real-Time Multiplayer Games. In: Proc. of NOSSDAV 02, Miami, FL, USA, 2002. 23～29

13 Kim S, Kuester F, Kim K H. A Global Timestamp-based Approach to Enhanced Data Consistency and Fairness in Collaborative Virtual Environments. Multimedia Systems, 2005, 10(3): 220～229

14 el Levante B, el Poniente P. In Search of Fairness Through Interactivity in Massively Multiplayer Online Games. In: IEEE CCNC 2006 Proceedings, 2006

15 Carson M, Santay D. NIST Net: a linux-based network emulation tool [J]. ACM SIGCOMM Computer CommunicationReview, 2003,33(3):111～126