南京大学本科生毕业论文（设计）中文摘要

毕业论文题目： 基于分布式多 Agent 系统的机器人足球游戏

计算机科学与技术系 院系 计算机科学与技术 专业 2008 级

本科生姓名： 余东亮

指导教师（姓名、职称）： 曹春 讲师

摘要：机器人足球比赛是一个热门研究领域，自上世纪 90 年代起已经得到了长足发展，并达到了相当高的水平。本文介绍了一个基于分布式多 Agent 系统环境设计的机器人足球游戏，一方面实践了通过多 Agent 系统来进行整个系统的架构，另一方面具体研究了其中的网络通信、决策系统等方面的解决方法。该系统在 Android 平台上进行了实现，实际运行时可以用两台 Android 手机连接之后模拟两个球队之间的比赛。

具体而言，本文工作包括：

1、我们研究了多 Agent 系统的架构方式，把两组反应式 Agent 与同一个服务端连接起来，和储存在服务端内的环境状态一起构成多 Agent 系统。同组 Agent 仅需要和服务端建立一条通信连接来感知和影响环境，减轻服务端压力的同时也提升了通信的效率。这样的架构还保证了所有的通信和决策以服务端状态为准，有利于保证 Agent 通信的有效性，也有提升了 Agent 决策的及时性和正确性。

2、我们研究了分布式多 Agent 系统中的网络连接问题，制定了用于两组 Agent 与服务端之间网络通信的策略,克服了网络延迟给系统带来的危害，增强了系统的实时性，保证了 Agent 决策的及时性。

3、我们研究了多种 Agent 设计方法，为 Agent 设计了一套基于角色的决策系统。该决策系统为每个 Agent 赋予角色，并让其依据自己的角色承担对应的任务。这样直观的设计有利于我们分角色测试 Agent 决策的正确性，提高了后期完善决策系统时的效率。

4、在具体实现时，我们基于 Android 平台设计了 C/S 模式的系统结构，将 Agent 按照队伍不同分布在不同的客户端手机上，每个客户端手机负责本方 Agent

的决策和通信功能，并连接到服务于所有客户的服务端手机以进行数据通信。体现了分组 Agent 对抗的公平性，并在实际运行中展现出了高效的通信能力。

关键词：多 Agent 系统；分布式系统；机器人足球；软件模拟；Android

南京大学本科生毕业论文（设计）英文摘要

THESIS: A Robotic Soccer Game Based on Distributed Multi-Agent System

DEPARTMENT: Computer Science and Technology

SPECIALIZATION: Computer Science and Technology

UNDERGRADUATE: Dongliang Yu

MENTOR: Chun Cao

ABSTRACT: Robotic soccer game is a hot research field and has made considerable progress since the 1990s. In this paper, we design a robotic soccer game based on the distributed multi-agent system environment. We apply the multi-agent system as the system architecture, and we study the solutions for network communication as well as decision-making system. This system is implemented on the android platform. Users can use two connected Android phones to simulate a soccer game.

Specifically, the work in this paper includes:

1、We study the architecture of Multi-Agent Systems. We establish a connection between two groups of Agents and the server. The game state inside the server and all the Agents constitute our Multi-Agent System. Only one communication connection between the Agents in a same group and the server is needed to percept and affect the environment, which reduced the pressure one the server and enhanced the communication efficiency. This architecture also ensures that all communications and decision-making to the server prevail, which makes the Agent communication more efficient and enhanced the timeliness and accuracy of the Agent decision.

2、We study the network connection problems in distributed multi-agent system and develop strategies for the network communication between Agents to overcome the network latency harm to the system, enhance the real-time system and ensure

the timeliness of the Agent decision.

3、 We study some design methodologies for Agent and develop a role-based decision-making system. The decision-making system endows every Agent with a role and the corresponding task. The intuitive design helps us to test the accuracy of the decision-making system by roles and enhanced the efficiency of consummating the system.

4、We develop a C/S system on the android platform to implement our system. Agents are distributed to different client phones by groups. Each client phone is responsible for the group's decision-making and communication. It also connects to server phone for all clients to make data communication. This architecture reflects the fairness of grouping Agent confrontation and demonstrates efficient communication capabilities in the actual operation.

KEY WORDS: Multi-Agent System; distributed system; robotic soccer; software simulation; Android

目 录

第1章 前言

1.1 研究背景

分布式人工智能(distributed artificial intelligence, DAI)是随着计算机网络、计算机通信和并发程序设计等技术的发展而产生的一个新的人工智能研究领域。它目前有两个主要的研究方向，一个是分布式问题求解（Distributed Problem Solving, DPS），另一个是多 Agent 系统（Multi-Agent Systems, MAS）。其中，多 Agent 系统是分布式智能研究的一个热点［1]。与此同时，基于分布式多 Agent 系统环境进行的机器人足球竞技正在不断发展，机器人足球世界杯(Robot World Cup, RoboCup)从1997年起至今已举行了16年，达到了相当高的水平。基于对多Agent系统和足球的双重兴趣，我们根据 RoboCup 的比赛模式，在 Android 平台上完整构建一套软件模拟的、可以进行实时比赛的分布式机器人足球多 Agent 系统[2]。

1.2 主要研究内容

- 为了构造软件模拟的机器人足球比赛，我们引入 MAS，每个球员作为一个Agent，一支球队所有 Agent 运行于一台 Android 手机上。进行比赛时将两台手机联网，此时所有球员及球场状态共同组成实时的分布式 MAS 环境。
- 对于每个 Agent，我们需要为其提供充分完备的策略来应对不断变化的环境，即引入 Agent 决策系统。这些策略需要结合足球运动的一般规律来制定，进行大量测试并不断修改才能达到较好的效果。
- 为了让两个球队能够联机对战，我们需要在网络上连接 Android 手机。由于 MAS 系统和网络的不稳定性，数据在网络上的传输变得不可靠，可能发生延迟、丢失等情况，从而导致系统运行效率的降低，我们需要制定策略来解决这些问题。
- 我们最终在 Android 平台上实现整个系统，这要求我们进行 Android 编程实现，并能通过在模拟器和真机上的测试。

1.3 主要研究成果

们构建了一个完整的分布式多 Agent 系统用于模拟机器人足球比赛，引入了完整的决策系统，使得 Agent 可以根据环境来选择恰当的行为，最终达到战胜对手的目标。

在网络传输方面，我们采用了缓存技术来消解网络的不稳定性，并且制定了相应丢弃策略来避免处理过多延迟信息导致的延迟叠加。

我们最终完成了一个基于分布式多 Agent 系统的机器人足球游戏，该系统可以在 Android 平台上进行联机对战。

1.4 论文的组织

本论文的内容安排如下：

第一章介绍研究的背景、需要研究的内容以及研究成果。

第二章介绍分布式多 Agent 系统技术。

第三章介绍网络连接方面的具体实现。

第四章介绍多 Agent 系统的具体实现。

第五章介绍系统在 Android 平台上的实现和运行情况。

第六章总结了我们的工作并提出展望。

第2章 分布式多 Agent 系统

智能 Agent 是指能自主活动的软件或者硬件实体,它们通过传感器感知环境并通过执行器输出行为，最终达到某种目标。多 Agent 系统(multi-agent system , MAS)是在一定环境中由多个互相作用的智能 Agent 组成的系统。MAS 可以用于处理那些单个 Agent 很难或无法解决的问题［3]。

2.1 Agent

2.2 Agent 定义

一种普遍的观点认为：Agent 是一种能在一定环境中自主运行和自主交互，以满足其设计目标的计算实体。此外，一个比较权威的定义是 Wooldridge 和 Jennings 在 1995 年给出的关于智能 Agent 的弱定义和强定义。

Agent 的弱定义认为：Agent 是具有自主性、社会性、反应性和能动性的计算机软件系统或硬件系统。

Agent 的强定义认为：Agent 是这样一个实体，它的状态可以看成是由新年、能力、选择、承诺等心智构件组成。即 Agent 除具有弱定义下的特性外，还应该具有人类的一些特性，图知识、信念、意图等，甚至包括情感［1]。

2.2.1 反应式 Agent

反应式 Agent 是一种不包括认知功能，仅对感知到的外界信息做出响应的 Agent。反应 Agent 的工作行为采取的是“感知-动作”模型。反应 Agent 虽具有对环境变化的快速响应能力，但其智能程度和灵活性较差［4]。我们的系统中将采用结构如图 2-1 这样的反应式 Agent。

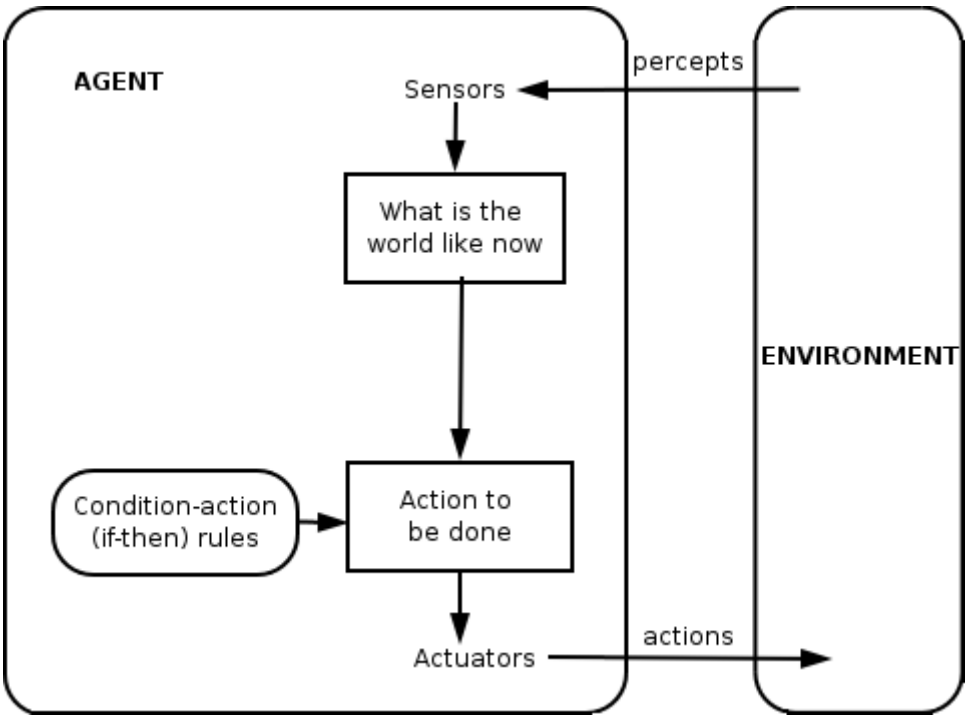

图 2-1

2.3 MAS

2.3.1 MAS 的概念

由于单个 Agent 能力的有限性，在实际应用中常用的是 MAS。MAS 是由多个自主 Agent 所组成的一种分布式系统。其主要任务是要创建一群自主的 Agent，并协调它们的智能行为。这样将各种具有不同能力的 Agent 结合起来，通过它们之间的相互作用，既分工又协作，共同解决问题。MAS 的结构可以用图 2-2 表示。

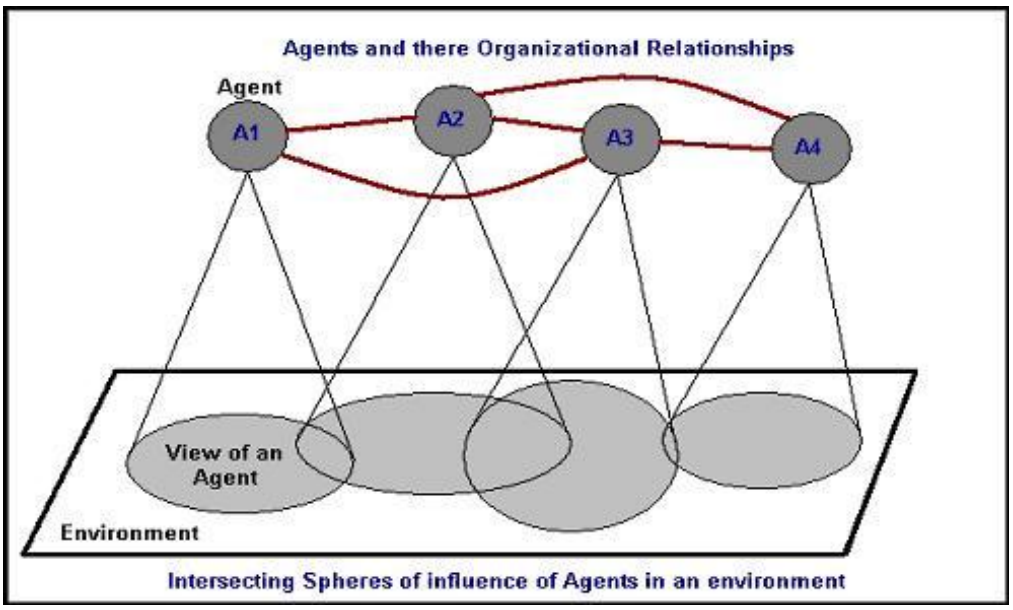

图 2-2

通过 Agent 之间的通信，可以开发新的规划或求解方法，用以处理不完全、不确定的知识； Agent 之间通过协作，改善了每个 Agent 的基本能力，而且可从 Agent 交互进一步理解社会行为；可以用模块化风格来组织系统也是 MAS 的一个特点。如果说模拟人是单 Agent 的目标，那么模拟人类社会则是多 Agent 系统的最终目标。

MAS 和单 Agent 系统之间的区别是，MAS 包含了许多各自形成独立的目标和行为的 Agent。从单个 Agent 的角度看来，主要的差别在于其他的 Agent 会在多 Agent 系统中不可预知地影响环境。更进一步说，MAS 中的 Agent 可能会通过信息交流进行直接的相互作用。在多 Agent 系统中当一组 Agent 有某个共同的长期目标时，我们可以将其看做一个队伍。为了达成这个目标，Agent 们必须协调彼此的行为（比如通过通信的方式）。他们必须在作为队伍一员的情况下独立地且效率地行动。如果环境中有其他一些 Agent，他们所要达成的目标与本队目标不一致时，可以认为这些 Agent 就是我们的对手。

2.3.2 MAS 的优势

MAS 在复杂系统设计中的具有相当大的吸引力。在某些领域甚至将使用 MAS 作为准则。举例来说，在某些具有不同种类实体的情况下，各个实体具有

不同（甚至冲突）的目标以及不同的信息，这时候使用多 Agent 系统在建模时就显得非常必要。不过即使在那些并不强制要求使用 MAS 的领域，运用 MAS 可能会带来以下优势：

1、使用 MAS 可以很方便地实现并行处理，故而增加系统的运行速度。这种情况在任务可以划分为若干个独立子任务并被不同 Agent 处理时显得尤其奏效。

2、MAS 在通常情况下鲁棒性非常好。当系统是由某个中枢单元控制时，仅仅一个很小的问题都有可能让整个系统崩溃。MAS 就完完全全不同了，当某个或某几个 Agent 失效时，整个系统仍然是可运行的。

3、MAS 的基本理念使得有关 MAS 的编程趋向于更简单。程序员可以确定子任务并将其分配给不同的 Agent，比较其集中控制的情况中整个任务仅由指定应用处理，MAS 的方式更加简单

4、MAS 的模块性使得我们可以在需要的时候随时增加新的 Agent 到系统中。这就是所谓的可拓展性。要知道，在一个整体性的系统中增加额外特性并不是一件容易的事情。

5、MAS 相对于单 Agent 系统而言的一个优势在于，Agent 们在 MAS 中可以不断观察环境，并且在多个地点同时开始行动。这就表明了 MAS 可以利用地域分布带来的好处。

6、相对于单 Agent 系统，MAS 通常有着更高的性价比。使用一个具有完成某个任务完备能力的单个机器人通常比使用多个具有部分能力的机器人（它们通常更便宜）要贵的多 [2]。

2.3.3 MAS 与机器人足球

从分布式人工智能的角度来看，机器人足球游戏是个特殊但同时也具有很大吸引力的多 Agent 环境。在机器人足球游戏中有两个互相对抗的队伍，每个队伍都有多个 Agent，它们互相合作来达到共同的目标：赢得比赛。为了完成这个目标，球队必须要进球，因此每个 Agent 必须根据局部和全局的情况进行考虑，以此来行动的更快，更灵活和更有协调性。这意味着尽管每个 Agent 的感知和行动都是局部性的，它们仍然需要考虑某个共享于全队的协调策略。因为两个队伍的目标不同，可以把对手看成是某种为了阻止我们达成目标而产生的动态的、有阻

碍性质的环境。这使得区域内同时存在着合作与对抗。另一个机器人足球游戏让人感兴趣的地方是比赛环境是高度动态化的，这意味着必须要有实时的决策，因为胜利依靠的是根据动态变化的环境来尽可能迅速的进行行动。需要说明的是，它们很有可能需要处理由感知器带来的环境噪声，而且由于对于环境了解的局限性，有相当一部分环境对于它们而言是不可见的。

我们所要做的工作就是制作一个软件仿真的机器人足球游戏，即一个分布式 MAS 系统。在服务端我们将足球比赛的实时情况建模，客户端的 Agent 必须根据它们从服务端所感知的信息进行及时的行动。

2.4 本章小结

本章介绍了 Agent 和 MAS 的相关概念，并讨论了 MAS 与机器人足球游戏的关联。

第3章 多 Agent 足球竞赛系统的设计

3.1 总体结构设计

我们的多 Agent 系统由两个客户端和一个服务端组成，服务端接收客户端输出的动作指令，执行它们以更新游戏状态，并将最新游戏状态反馈给客户端；客户端接收游戏状态信息，并交给 Agent 们处理，Agent 与球员一一对应，经过推理得出某位球员下一步要做的动作，转换成一定格式的动作指令，并发送给服务端。其结构如图 3-1。

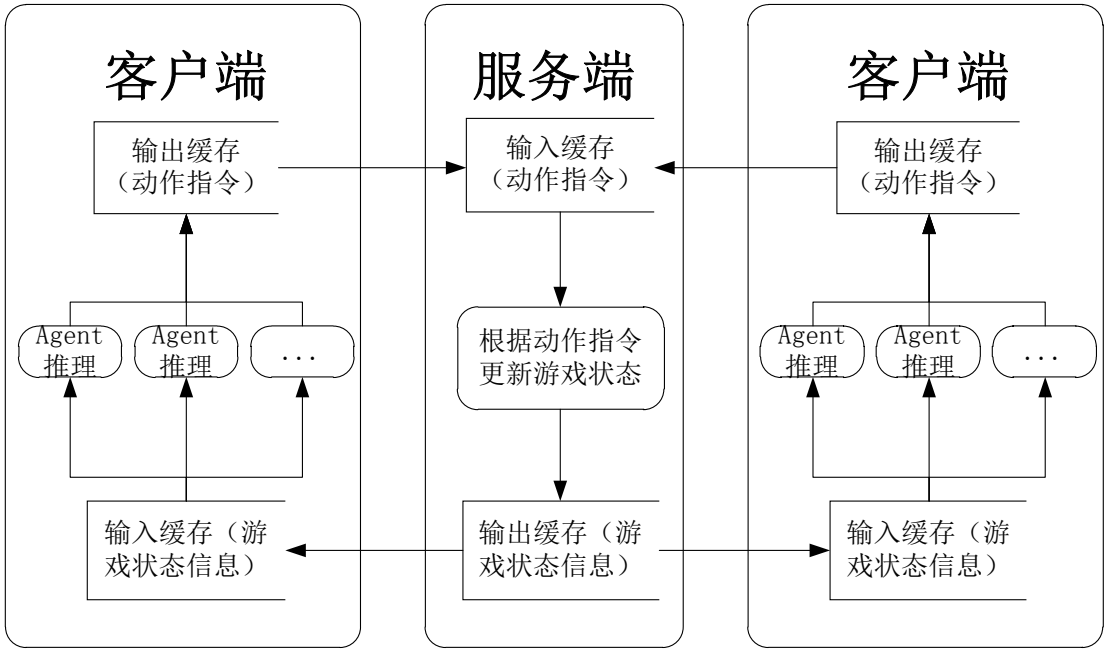

图 3-1

3.2 服务端设计

服务端接收到客户端传输过来的 Agent 动作指令后，一一尝试执行，执行完毕后进行一次状态推进。

当执行 kick 指令时，例如“0, 8 , kick , 1.3 , 40 ”，首先检查 0 队 8 号球员和球的距离是否在最大踢球距离内，如果超出了这个距离，则自动忽略这条指

令，如果没有超出，则根据 1.3 角度和 40 力量，对球进行相应速度叠加。

当执行 dash 指令时也类似，例如“1, 6 , dash , 4.2, 35”，1 队 6 号球员将给自己一个 4.2 角度、35 力量的奔跑驱动，这条指令被执行后球员的速度将有所改变。

所有指令都被执行完成后，球和球员的速度有所改变，然而对于速度的改变还没有结束。球场对球是有摩擦力的，因此每个周期球都应该被减少一定速度，直到速度为 0。球员的速度也会不断降低，只有进行 dash 动作时才会增加速度，因此认为球场对球员也有摩擦力，每个周期球员都会被减少一定速度，直到为 0。经历过这样的减速之后，得到最新的速度。随后我们根据球和球员的位置，加上他们的速度，得到最新的球和球员位置。

$$(\mathrm{px}', \mathrm{py}') = (px, py) + (vx', vy')$$

如果进球了，比分会有所改变，而比赛时间是在一直变化的，加上最新的速度和位置信息，包装成如图 4-2 所描述的游戏状态信息，放入服务端输出缓存等待被发送。

3.3 客户端设计

客户端的总体设计如图 3-2。我们在一个客户端上使用了多个 Agent 来为每个球员一对一地进行推理计算，这些 Agent 从一个共同的消息源，即图中“客户端可使用的球场状态信息”中获得自己需要的球场状态信息，然后各自进行推理，推理结束之后将输出信息放入“球员动作指令缓存”。我们注意到由于自身情况的不同，他们的推理时间可能并不完全相同，也就意味着有人早完成有人晚完成，只有在所有“球员动作指令缓存”都被更新过后，客户端才会把这些缓存内的信息提取后合并，然后送入客户端输出缓存等待发送给服务端。

我们注意到“客户端可使用的球场状态信息”的另一个用处是成为视频输出的来源，我们根据这些信息在手机屏幕上输出游戏的画面用于观察。

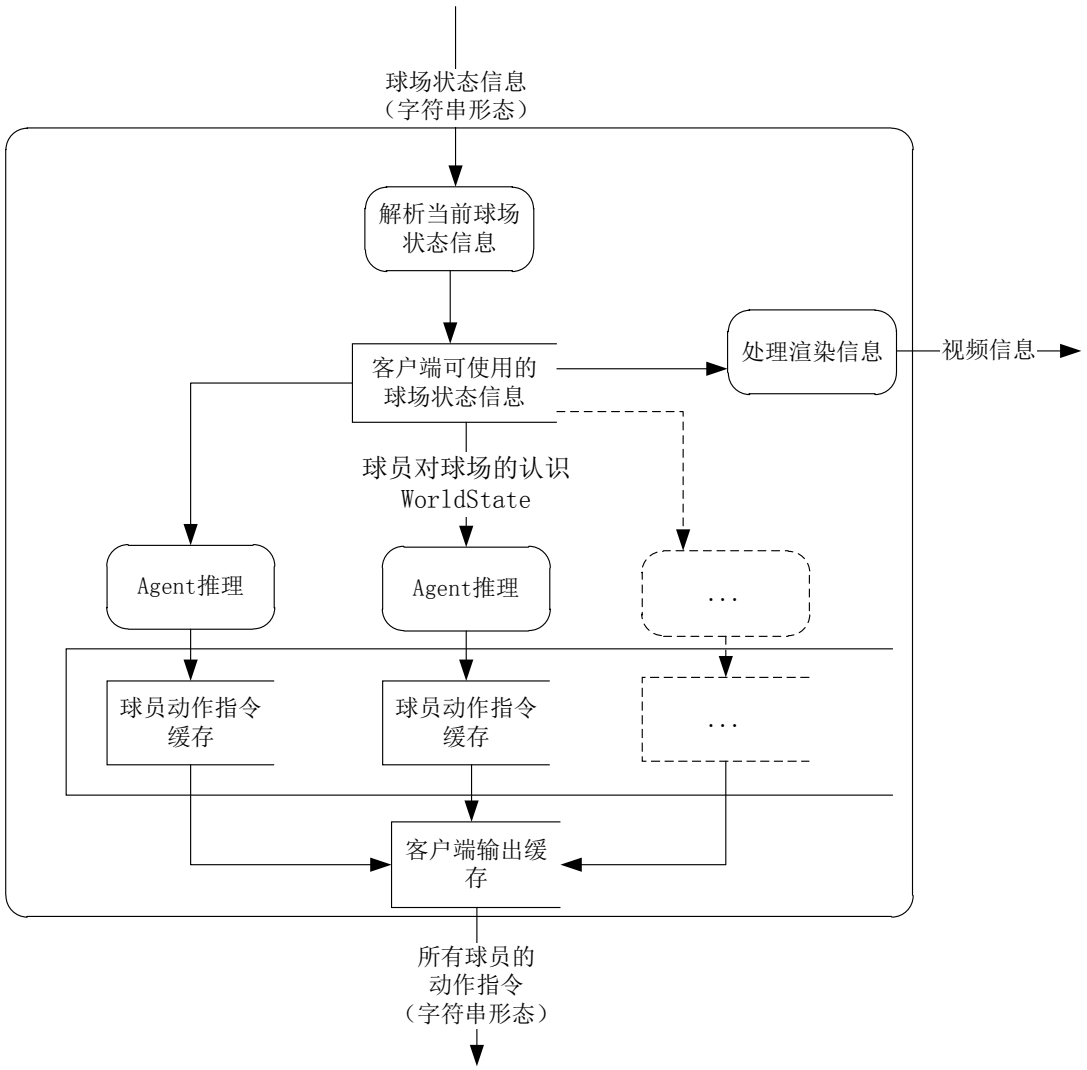

图 3-2

3.4 Agent 原子动作

球员的原子动作有跑动和踢球，其他所有复杂动作都由这两个动作组合而成。当 Agent 决定进行某个底层动作时，调用 Listing 3-1 和 Listing 3-2 展示的两个方法。

Listing 3-1

```java
/**
 * 返回踢球命令
 * @param angle
 *            踢球角度，PI制
 * @param power
 *          踢球力度
 * @return
 *          字符串形式的踢球指令
 */
private String kick(double angle, double power) {
    StringBuffer buffer = new StringBuffer();
    buffer.append("kick" + ",");
    buffer.append(angle + ",");
    buffer.append(power + ",");
    return buffer.toString();
}
```

Listing 3-2

```java
/**
 * 返回奔跑命令
 * @param angle
 *            奔跑施力角度，PI制
 * @param power
 *            奔跑用力程度
 * @return
 *          字符串形式的奔跑指令
 */

private String dash(double angle, double power) {
    StringBuffer buffer = new StringBuffer();
    buffer.append("dash" + ",");
    buffer.append(angle + ",");
    buffer.append(power + ",");
    return buffer.toString();
}
```

调用后返回的字符串指令类似于“kick，1.3，40”、“dash，4.2，35”。

3.5 Agent 底层动作

Agent 底层动作是由原子动作组合而成的更高级的动作，一般都是有球情况下产生的动作。

3.5.1 传球

传球的目标是把球传向既定的位置，因为球场摩擦力的存在，传球距离不同使用的力量也不同，而且需要考虑到如果在近距离传球时使用太大力量，可能会

让接球者反应不过来而接不到球。我们在传球前先计算当前球和目标位置之间的距离，根据这个距离选择恰当的力量，再计算出传球角度，将力量和角度作为参数传入踢球模块。

3.5.2 射门

射门其实就是对着球门把球踢一下，然而由于射门需要的是让对方球员无法截下来，因此需要用尽可能大的力量。在给定目标位置后，把相应的角度和最大踢球力量作为参数传入踢球模块。

3.5.3 带球

球员带球是经常出现的情况，当球员控制皮球，并不急于传球或射门时，他们会选择把球带向某个位置，即带球具有方向性。带球包括和踢球和追球两个动作，不断交替之中形成带球过程。具体过程为：先计算人和球之间距离，若距离大于最大可踢球距离（踢不到球）则向球的方向跑动，否则向要带球的方向踢一次球，这个踢球的力量可以调整，若想快速带球则力量大一些（踢得离身体远一点），若想慢慢带球则力量小一些。如此循环可形成带球过程。

3.6 Agent 中层动作

前面两节我们介绍了一些球员的动作，提到许多参数，例如带球，传球，射门的角度和力度。我们在这一节介绍一些更高级的动作，它们具备了计算这些参数的能力，调用这些动作可以使得球员的能力有明显提升。

3.6.1 选择性传球

传球是足球比赛中非常重要的组成部分，传球的准确性决定了一个球队对比赛的掌控能力。传球需要注意的方面很多，我们在系统中考虑了这三方面：1、球在被传出后不被对方拦截；2、球尽可能向对方底线传递以增加威胁度；3、接球队员拿球后要有尽可能多的活动空间。其中 1、3 是从传球安全性上考虑的，2 是从传球威胁性上考虑的。我们的传球必须要保证安全性，不能轻易把控球权拱

手相让，因为把球控制在自己脚下才能立于不败之地；但同时也要能够通过传球渗透对方防线，以获取进球，因为进攻才是最好的防守。

我们在进行选择性传球时，先给定几个候选目标，这些目标可以是某个球员，或者场上某个具体坐标。我们分别为这几个候选目标计算其传球可行性，用一个数值 k 标识，让 k 值越大表示越值得进行传球。很明显 k 为前述需要考虑的三方面所影响，我们给这三方面分别一个考察值 k1、k2、k3,。

我们考察 k1，当球与传球目标连线线段周围存在的对手个数越少时，球越不可能在传递过程中被截断，因此我们设定一个“安全范围”s，如图 3-3，计算球场上距离连线线段小于 s 的对手数量，即图中进入阴影区域的对手数量，数量越少则 k1 越大，反之亦然。

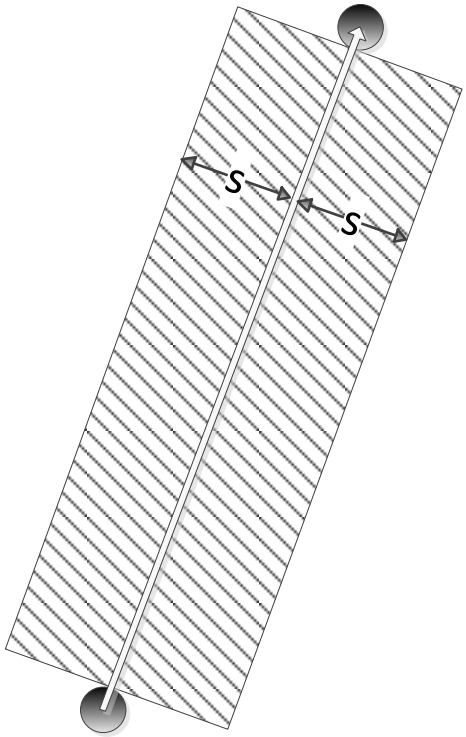

图 3-3

我们考察 k2，当传球目标离对方底线越近时 k2 越大，反之亦然。

我们考察 k3，接球球员拿球后周围的对方球员不能太多，否则很容易刚拿球就被反断，因此传球目标周围对手数量越少 k3 越大，反之亦然。

于是我们有 k = w1 * k1 + w2 * k2 + w3 * k3。

w 表示某个因素的权重，可以用于调节球队传球的侧重点，例如希望传球更有威胁，即每次传球都能更靠近对手底线，则将 w2 调高；如果希望传球更安全不被对手拦截，则把 w1 调高。

现在我们对所有候选目标计算 k 值，选取 k 值最大的目标进行传递，认为我们综合考虑了前述三个方面。

3.6.2 选择性射门

射门是得分的唯一手段，是直接改变比分的方式，在射门时需要考虑 1、当前射门方式能否让球的运行路线不偏离球门；2、射门路线上对方球员数量尽可能少。综合考虑这两方面我们才能选择出最佳射门路线。

我们给出球与球门两边立柱的连线形成的扇形，如图 3-4，在这个扇形上每隔 α° 选取一个射门候选角度，分别计算在这个线路上一定范围 s 内对手数量，数量越少则射门不被阻挡的可能性越大，得分可能性也就越大。注意到由于射门用的力量大，s 可以选取地更小，因为如果不是离得特别近，球速太快也是不会被拦截的。

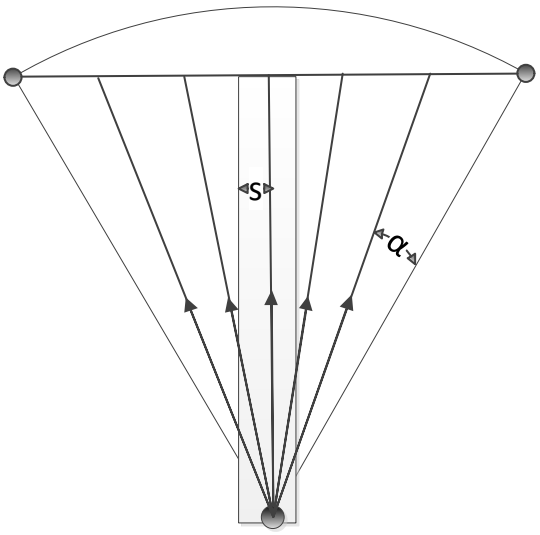

图 3-4

3.6.3 带球过人

除了传球外，突破对方防线的另一有效方法是带球突破，这要求带球球员有

很好的预判，根据当前情形选择下一步出球的路线，巧妙绕过对方防守。为了达到这样的效果，我们建立一个带球模型，如图 3-5，设定带球目标位置是“引力点”，即图中五角星，对方球员是“斥力点”，即图中白点，由这两股合力推动球员行进［5］。

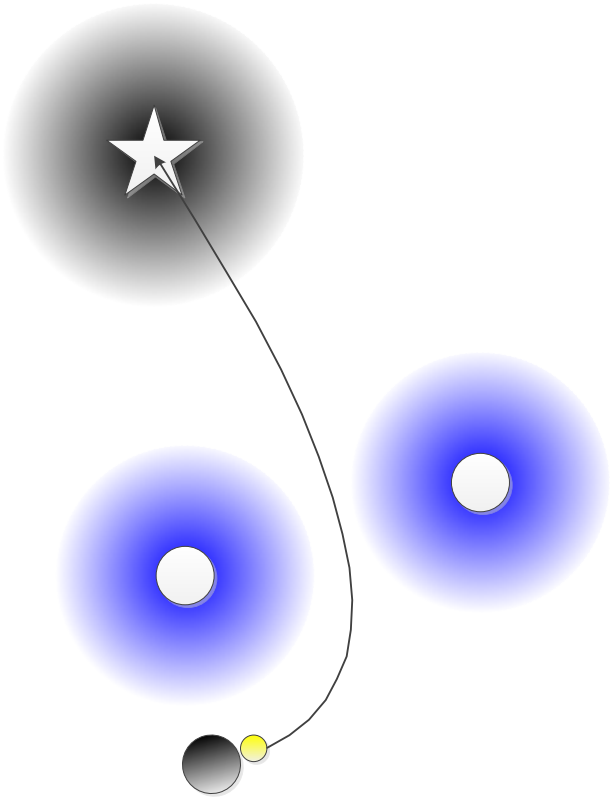

图 3-5

在实际操作中，考虑图 3-6 的情况，记位置 X 离目标的距离为 DT(X)。我们记球现在的位置为 O，选定若干可能运球方向，标出球运行某一距离 s 后的位置 P， 两个位置与目标距离之差 DT(P) - DT(O)，令 f(DT(P) - DT(O)) 为关于此距离差的激励函数。我们对 f() 的设定为：f 为单调递减的函数，当自变量大于零，即 P 离目标比起 O 来更远了，f 的值小于零，即给予负的激励；当自变量小于零，即 P 离目标比起 O 来更进了，这是需要鼓励的，f 的值大于零，给予正的激励。

记位置 X 离最近的对手距离为 DO(X)。还是选用 O 和 P，令 g(DO(P)) 为关于此距离的激励函数。我们对 g() 的设定为：g 为单调递增的函数，其趋势类

似于 y = - x^(-1)。即当 DO(P)很小、P 离对手位置很近时，g 的值很小，甚至为负，这意味着不鼓励我们把球带到离对手很近的位置；反之，DO(P)较大时，g 会变大，但增加趋势会变小，这意味着也不是离对手越远越好，能保证不被对手把球截掉就行了。

我们把 f 激励和 g 激励加起来，选取总激励最大的那个方向作为带球方向。

现在结合下面的例子进行解读：球原本在 O 位置，现在给定 8 个可能的运球方向，除了 0、1、2 之外其余的太不靠谱，我们现在只考虑这三个，可以看出能够得到 f 激励最多的是 P1，它离目标最近了，但是由于这个位置离对手太近，获得的 g 激励为绝对值很大的一个负值，因此会被舍弃。再看 P0 和 P2，由于 P2 离目标更近，且基本“逃出”了对手的斥力影响范围，经过计算发现 P2 的激励最大，因此我们选择将 P2 作为下一次带球的目标。

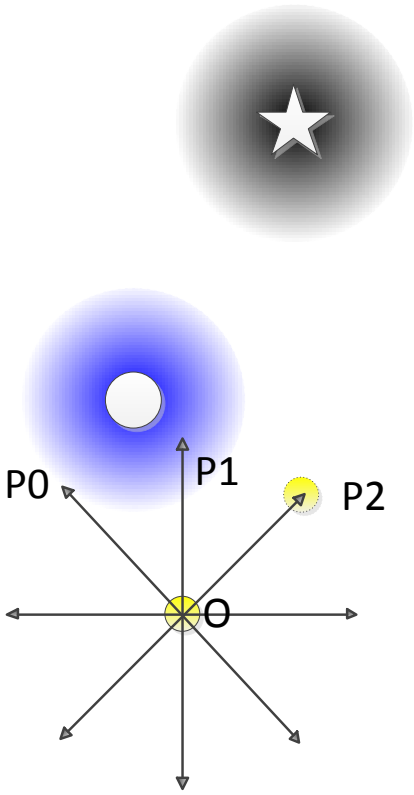

图 3-6

3.7 Agent 高层策略

3.7.1 阵型

在足球比赛中，阵型是非常重要的一个方面，当球员并不控制球，即他是无球球员时，他应该根据当前比赛状态和自己在球队中的角色选择恰当的站位。球员们如果能保持住相对稳健的阵型，在进攻时及时压上，在防守时人员到位，则可以在局部地区出现人数优势，以此来达到赢球的目的。为了在不同情况下让无球球员明白自己应该站在球场何处，我们把比赛的状态根据球在场内的不同位置进行区分（比如球在对方禁区内时比赛状态设置为“全力进攻”，而球在本方禁区内时比赛状态设置为“全力防守”），同时我们根据球员编号确定他在球队中的角色（比如边后卫或者前锋），最后我们结合比赛状态和球员角色确定某个无球球员的站位坐标。为了增加游戏的随机性，球员的站位并不是定死的，例如处于A状态下，B角色的无球球员的站位坐标是(x,y)，并不是说他一定要处于(x,y)，我们给定了一个“自由范围”Δ，不超出这个范围就是合法的，所以A状态下，B角色的球员可以在(x±Δ,y±Δ)内的任何位置。

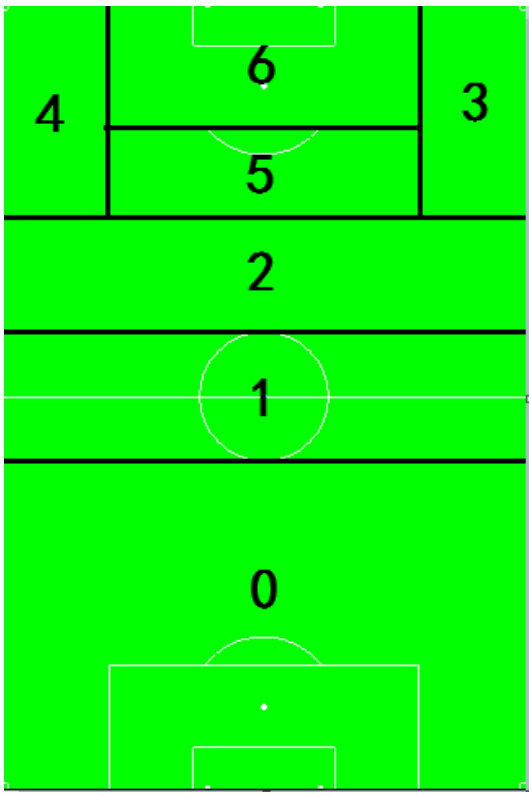

图 3-7

我们根据球在球场的位置，用 0-6 的数字表示比赛状态，一共七种，如图 3-7。假定我们是从 0 向 6 进攻的队伍，这里依次解读状态：

- 0 状态表示球到了本方半场且已经过了中圈弧，再向本方球门发展就很危险了，这时候应该让所有球员回撤一些以协助防守。
- 1 状态表示球在中场附近，此时双方应该处于相持阶段，如果球被对方拿到，则应迅速进入防守状态，如果球被本方拿到，则可以迅速进入进攻状态，因此应该兼顾防守和进攻，适中地站位，而不用像 0 状态那样回撤过多。
- 2 状态表示球进入对方半场，我们应该采取进攻战术，争夺皮球并且继续向前推进。此时后卫线适当提前，两个边锋和两个前锋都开始向前穿插，整体站位较 0 和 1 状态更加靠近对方球门。
- 3 和 4 状态表示球进入对方半场两个边路的位置，如果我们的球员持球并能形成突破，可以直接传中给前锋形成得分机会。此时两个边锋继续前插，以寻找传中机会，两个前锋则向禁区内游动，等待射门机会。
- 5 状态表示球就在对方禁区前，我们可以选择在中路突破或者分到边路寻求传中。此时边锋和边后卫前插，中场压上，前锋也要适当靠前以压迫对方防线。
- 6 状态表示球已经在对方禁区里了，我们要做的是争抢到皮球然后寻求射门，此时边锋和前锋都向禁区内靠拢，中场压到禁区前准备接应。

在我们的系统中，给出了传统 4-4-2 阵型，由两个前锋，四个中场（其中两个边锋）四个后卫和一个门将，这样的阵型攻守平衡，在进攻时，两个前锋作为桥头堡，四个中场中居中的两个控制住球场中部的位置，两个边锋在边路穿插，两个边后卫也适当前压以接应前场，中卫和门将则拖在最后面。图 3-8 展示了几种状态下的站位阵型图。

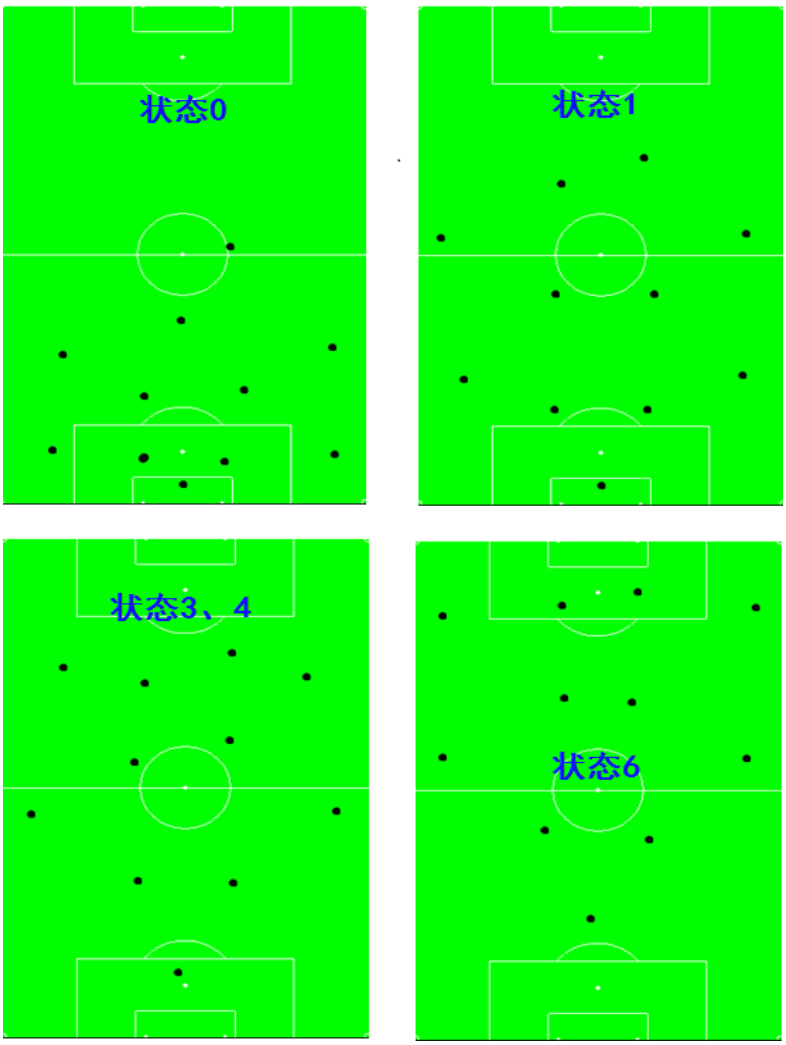

图 3-8

3.7.2 动作选择

每个 Agent 必须在自己的行为周期内选择一个动作来试图执行，这就是所谓的行为选择问题(Action Selection Problem, ASP)。在 ASP 的背景下，一种行为选择机制(Action Selction Mechanism, ASM)用于在给定外部或内部输入后产生选择好的行为作为输出。ASP 主要关注 Agent 在给定情形后选择了哪个行为，而 ASM 注重于细化这样的选择是如何进行的 [2]。

在软件模拟的足球游戏环境下，游戏状态是不断变化且不可预测的，因此我们不可能对 Agent 给定一系列行为让它去按顺序进行，以此来形成一个进攻或防守策略。在实践中，我们的 Agent 根据当前周期服务端发送来的游戏状态建立自己的世界模型，并根据这个模型推理出下一步的行为，也就是说前面周期的信息

并没有被使用，我们只对当前环境进行一次反应。前面已经描述了根据球员编号确定他们的角色，我们的动作选择策略将根据球员角色来进行。

- 门将：门将会尽量把靠近自己的球抢下来并传给其余队友，其余时间根据上节所述的阵型站好自己的位置。门将不应该试图带球，这会容易让对手截断产生很糟糕的后果。Algorithm 3-1 展示了具体算法。

Algorithm 3-1

守门员

```
If ball is kickble then
        pass with selection
else if distance to ball is shortest among all players then
        move to ball
else
        move to strategic position
end if
```

- 中后卫：中后卫会优先选择把球传给处在中场的球员，如果球离自己很近，或者已经到了一个需要采取行动的“预警距离”之内，则试图去抢球。中后卫同样不应该试图带球。Algorithm 3-2 展示了具体算法。

Algorithm 3-2

中后卫

```
If ball is kickble then
        pass to midfielders with selection
else if distance to ball is shortest among own players
        || distance to ball < EARLY_WARNING_DISTANCE then
        move to ball
else
        move to strategic position
end if
```

- 边后卫：边后卫经常在边路活动，如果自己周围很空没有对方球员，则试图往对方半场带球。其他情况下优先将球传给中场的队友。如果自己没有控球，但球到了需要采取一定行动的距离内，则试图去抢球。Algorithm 3-3 展示了具体算法。

Algorithm 3-3

边后卫

```
If ball is kickble then
        if no opponent in DRIBBLE_SAFE_DISTANCE then
                dribble until passed hald-way line
        else
                pass to midfielders with selection
else if distance to ball is shortest among own players
        || distance to ball < EARLY_WARNING_DISTANCE then
        move to ball
else
        move to strategic position
end if
```

- 中场：中场球员有时候会发现一些射门机会，此时应抓住机会尝试射门；其他情况下，他们应尽量把球传给前面位置更好的队友；或者带球向前突进也是不错的选择。如果这些选择都不太合适，适当回传也是可以的。当球不在自己控制之下时，和后防线上的队友一样，应该在皮球进入一定范围时试图去抢球。其余时间站住阵型位置。Algorithm 3-4 展示了具体算法。

Algorithm 3-4

中场

```
If ball is kickble then
        if goal probability is high then
                shoot
        else if there is a free teammate in front of me then
                pass to front with selection
        else if no opponent in DRIBBLE_SAFE_DISTANCE then
                dribble forward
        else
                pass backward with selection
else if distance to ball is shortest among own players
        || distance to ball < EARLY_WARNING_DISTANCE then
        move to ball
else
        move to strategic position
end if
```

- 边锋：边锋是现在进攻中的主要发起点，他们可以在有把握的时候选择射门，但更多时候，他们会选择带球突入边路，在有威胁的地带起脚传中，如果中路的前锋能抢到这些传中球就很有可能直接射门得分。如果对手防守太严密没有突破空间，传给位置更好的队友也是个选择。和其他人一样，他们也需要承担一定的防守任务。Algorithm 3-5 展示了具体算法。

Algorithm 3-5

边锋

```
If ball is kickble then
        if goal probability is high then
                shoot
        else if in a location where cross is threatening then
                pass to strikers(cross)
        else if no opponent in DRIBBLE_SAFE_DISTANCE then
                dribble to goal line
        else if there is a free teammate in front of me then
                pass to front with selection
        else
                pass backward with selection
else if distance to ball is shortest among own players
        || distance to ball < EARLY_WARNING_DISTANCE then
        move to ball
else
        move to strategic position
end if
```

- 前锋：前锋最主要的任务是射门得分，他们当然要在有机会的时候就选择射门，但是在射门路线被挡的情况下，传给另一个前锋或者已经插上的两个边锋也是很好的选择，这样可以让他们重新组织进攻。有时候自己直接带球突破也行，或许就能获得直接射门的机会。实在不行也可以往回传，让中场球员重新组织进攻。其他时候前锋们也需要承担前场的防守任务。Algorithm 3-6 展示了具体算法。

Algorithm 3-6

前锋

```
If ball is kickble then
        if goal probability is high then
                shoot
        else if one of the wings and strike is free then
                pass with selection
        else if no opponent in DRIBBLE_SAFE_DISTANCE then
                dribble to goal line
        else
                pass backward with selection
else if distance to ball is shortest among own players
        || distance to ball < EARLY_WARNING_DISTANCE then
        move to ball
else
        move to strategic position
end if
```

3.8 本章小结

我们在本章介绍了 MAS 的详细设计，包括服务端设计，客户端设计，以及从底层到高层的 Agent 实现细节。

第4章 Agent 通信系统设计

在这一章我们将要详细描述分布式多 Agent 系统中的通信系统。由于服务端和客户端是分离的独立个体，它们之间必须通过网络通信进行实时的数据传输。在网络上始终传输字符串形式的信息，我们规定了统一的数据传输格式，并且关注高效、迅速、没有遗漏地发送和接收数据。由于这是一个实时系统，因此发送和接收数据都应以一个固定的频率 f 不间断地进行，考虑到网络和系统的不稳定性，在信息发送、传输、接收过程中都有可能发生不可预知的延迟，我们的通信功能必须要能够应付这些风险，以保证系统正常运行。

4.1 Agent 通信系统总体结构

系统由一个服务端和两个客户端组成，服务端接收两个客户端发送来的信息，并能向两个客户端发送信息，客户端只需与服务端传输数据。为了增加容错能力，在服务端和客户端建立输入与输出缓存是必要的，它们可以缓解数据流速率不正常导致的阻塞问题。约定整个系统的传输频率为 f，图 4-1 是 Agent 的总体结构图。

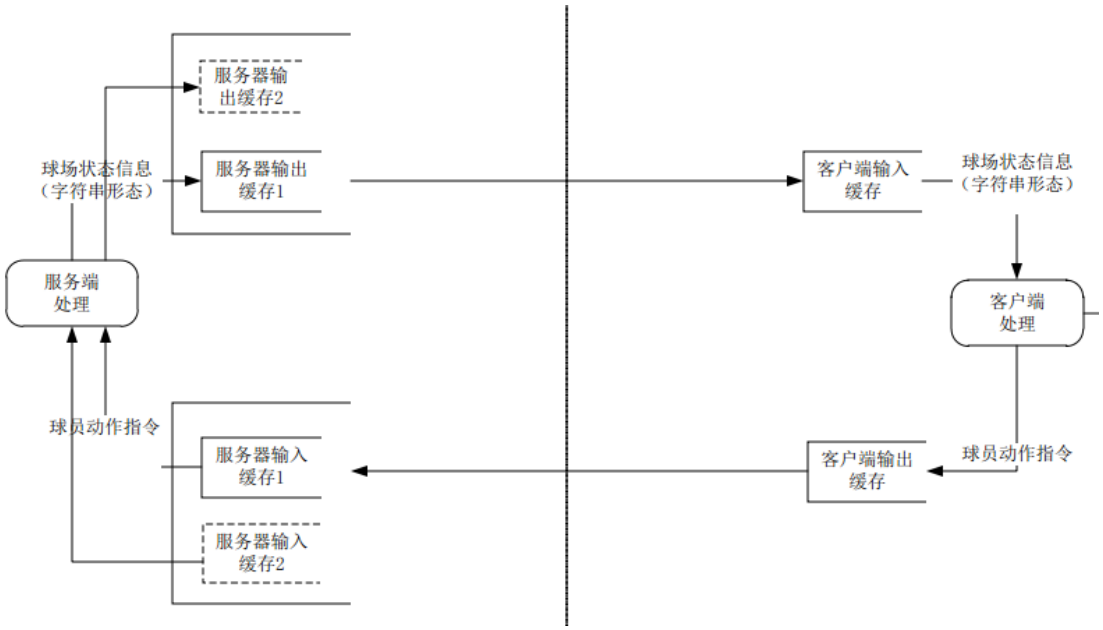

图 4-1

4.2 数据传输格式

4.2.1 客户端->服务端（动作指令）

服务端接收客户端发送来的动作指令有两种，dash 和 kick，分别对应奔跑和踢球，格式如下：

dash,angle,power

dash 用于表明这是一条奔跑指令，angle 表示奔跑方向的角度，power 表示奔跑用的力气大小，power 越大则人获得的动力越大。

kick,angle,power

kick 用于表明这是一条踢球指令，angle 表示踢球的方向，power 表示踢球所用的力气，power 越大则给球获得的动力越大。

Agent 将动作指令存入输出缓存前还要将自己的身份标识添加上去，这类似于网络协议里在数据包上添加头部，Agent 的身份标识包括自己所属球队（用 0 和 1 区分两个球队），自己的编号（0 到 10，因为一个球队有 11 名球员），标识之间用逗号隔开，标识与指令之间也用逗号隔开，某条 Agent 最终送至客户端输出缓存的动作指令字符串可能会像这样：

“0, 8 , kick , 1.3 , 40 ”

对这条指令我们可以这样解读：球队 0 中的 8 号球员，想要以 1.3 的角度（PI 制），40 的力量来踢球。

当然这些指令只是 Agent 的意愿，能否落实还要看当前的游戏状态，例如球离球员还有 10 米，他却想踢球，这显然是无法被执行的。

4.2.2 服务端->客户端（游戏状态信息）

游戏状态描述了当前足球游戏里所有的信息，包括球员们的位置、速度，球的位置、速度，以及场上的比分和比赛时间。这些信息由服务端发送给客户端。

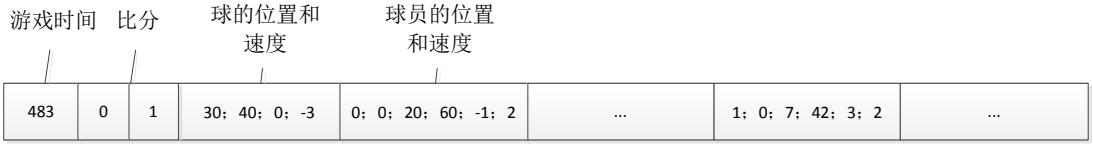

图 4-2

我们对图 4-2 表示的游戏状态进行解读：

“483”表示游戏已经运行至第 483 个周期；

“0 和 1”表示了现在比分是 0:1,我们约定队伍编号为 0 的分数显示在前，编号为 1 的分数显示在后；

“30;40;0;3”表示球的 x 坐标为 30，y 坐标为 40，x 方向速度为 0，y 方向速度为-3；

“0;0;20;60;-1;2”表示编号为 0 的球队中的编号为 0 的球员，他的 x 坐标为 20，y 坐标为 60，x 方向速度为-1，y 方向速度为 2；

其后 21 个球员的信息都类似，而我们在这样的信息包装成字符串时，在每个矩形之间用“|”隔开，某条送至服务端输出缓存的游戏状态信息字符串可能会像这样：

“483|0|1|30;40;0;-3|0;0;20;60;-1;2|.....|……|……”

这样的用于描述游戏状态的字符串是在服务端更新完一次状态后包装出来并准备发送给客户端的。

4.3 服务端部分

4.3.1 服务端输入

服务端有两个输入口，对应两个输入缓存(ServerInputBuffer)，分别服务于两个客户端。这两个输入缓存组成输入缓存池(ServerInputBufferPool)，同时存在一位负责输入的“工作人员”(ServerNetworkIn)，它在不断监视着 socket 输入流，一旦有数据传入，就将其放入缓存池中相应的缓存中。

缓存池结构如图 4-3。

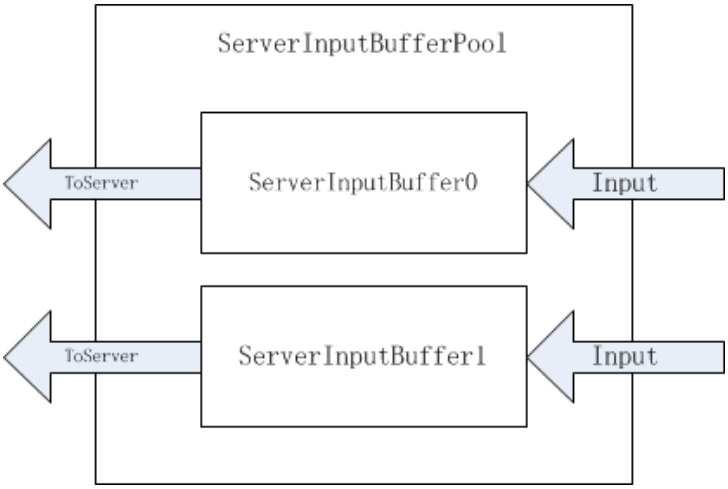

图 4-3

其中每个 ServerInputBuffer 的 UML 类图如图 4-4。

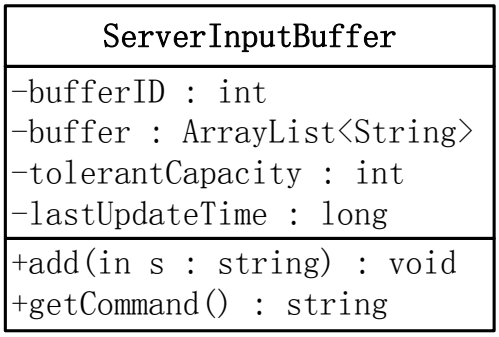

图 4-4

bufferID 表示这条 buffer 的序号（0 或 1），buffer 存储着 String 类型的缓存信息队列，tolerantCapacity 表示缓存最大可以容忍的队列长度（即 buffer 的 maxSize），lastUpdateTime 表示最后一次更新的系统时间，单位是 ms。

add(String s)方法用于向 buffer 中增加一条信息，getCommand()方法用于取出 buffer 队列中最前面的元素。

4.3.2 服务端输出

服务端有两个输出口，对应两个输出缓存(ServerOutputBuffer)用于存储将要输出至客户端的字符串信息，分别服务于两个客户端。这两个输出缓存组成输出

缓存池(ServerOutputBufferPool)，同时存在一位负责输出的“工作人员”(ServerNetworkOut)，它在不断监视着输出缓存，只要输出缓存中有数据，就将其通过 socket 发送至相应客户端，并在缓存中清空此数据。

缓存池结构如图 4-5。

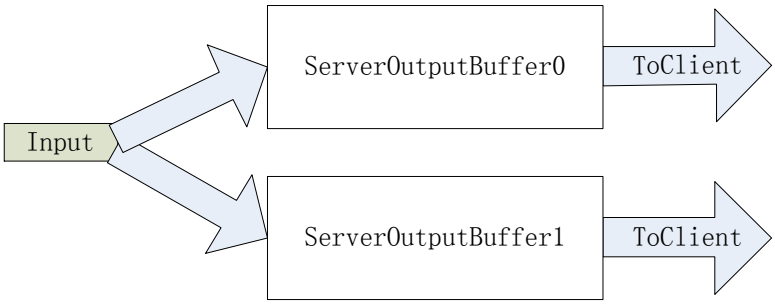

图 4-5

其中每个 ServerOutputBuffer 的 UML 类图与图 4-4 类似，只是缺少了成员 lastUpdateTime。bufferID 表示这条 buffer 的序号（0 或 1），buffer 存储着 String 类型的缓存信息队列，tolerantCapacity 表示缓存最大可以容忍的队列长度（即 buffer 的 maxSize）。

add(String s)方法用于向 buffer 中增加一条信息，getCommand()方法用于取出 buffer 队列中最前面的元素。

4.4 客户端部分

4.4.1 客户端输入

客户端输入缓存(ClientInputBuffer)存储服务端通过 socket 传过来字符串，存在一位一位负责输入的“工作人员”(ClientNetworkIn)在不断监视着 socket 输入流，一旦有数据传入，就将其存储于输入缓存。

ClientInputBuffer 的 UML 图如图 4-6：

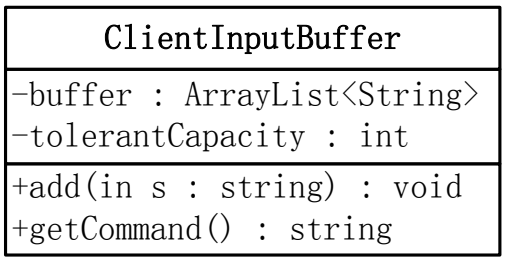

图 4-6

buffer 存储着 String 类型的缓存信息队列，tolerantCapacity 表示缓存最大可以容忍的队列长度（即 buffer 的 maxSize）。

add(String s)方法用于向 buffer 中增加一条信息，getCommand()方法用于取出 buffer 队列中最前面的元素。

4.4.2 客户端输出

客户端输出缓存(ClientOutputBuffer)，用于存储将要发送给服务端的字符串数据，同时存在一位一位负责输出的“工作人员”(ClientNetworkOut)，它在不断监视着输出缓存，只要输出缓存中有数据，就将其通过 socket 发送至服务端，并在缓存中清空此数据。

其中每个 ClientOutputBuffer 的 UML 类图与图 4-6 类似。buffer 存储着 String 类型的缓存信息队列，tolerantCapacity 表示缓存最大可以容忍的队列长度（即 buffer 的 maxSize）。

add(String s)方法用于向 buffer 中增加一条信息，getCommand()方法用于取出 buffer 队列中最前面的元素。

4.5 阻塞的发生

从上一节的描述可以看出，在一个数据传输循环中可能会有四处发生阻塞，如图 4-7。

其中 1、3 处的阻塞情况是：“工作人员”发现当前输出缓存里没有信息可以用于发送到 socket 中传输出去，因此阻塞，等待缓存里有信息后再行动。这种阻

塞发生的直接原因是前面的计算处理还未完成。

2、4 处的阻塞情况是：“工作人员”发现 socket 没有数据流传入，因此阻塞，等待 socket 有新的数据流传入后，再将其存入相应输入缓存。这种阻塞发生的直接原因是网络上的数据还没有传输过来。

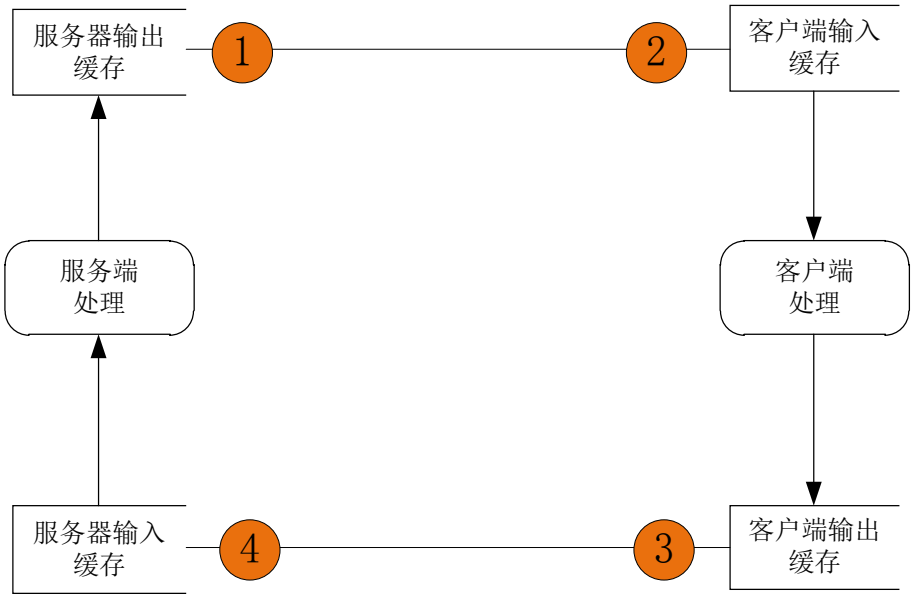

图 4-7

4.6 延迟现象

单独地看服务端和客户端一对一传输，整个系统有两个“输入-输出”对，分别是“服务端输出 - 客户端输入”和“客户端输出 - 服务端输入”。

由于我们的服务端以固定频率 f 运行，意味服务端的发送频率也是 f，不考虑网络延迟，则客户端的接收频率也应该是 f。在理想的状态下，客户端一经接收信息，就要迅速计算出要输出的信息，并发送至服务端，以赶在服务端下一次刷新游戏状态前能接收到这些客户端信息并予以处理。

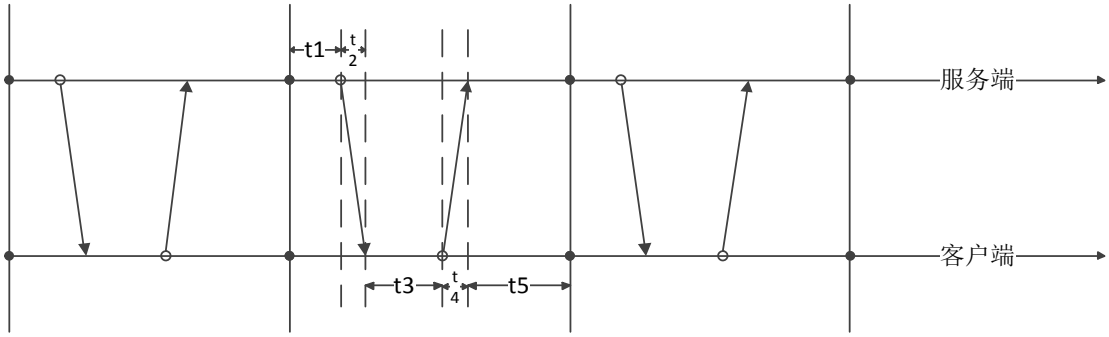

图 4-8

我们观察图 4-8，它展示了理想情况下的多个周期内，数据传输于服务端和客户端之间的情况。t1 的开端是服务端一次周期的开始时刻，图中每两条直线之间的部分代表服务端的一个周期。由服务端指向客户端的箭头表示消息由服务端传递至客户端，由客户端指向服务端的箭头表示消息由客户端传递至服务端。

t1:服务端进行一次处理操作所用的时间。由于不需要进行 Agent 推理，而只是将每次客户端发送过来的动作命令全部执行一遍以更新游戏状态（即进行一些数值的加减操作），我们认为 t1 是固定不变的。事实上 t1 相当的短。

t2:消息由服务端传递至客户端所用的时间。

t3:客户端进行一次处理操作所用的时间。

t4:消息由客户端传递至服务端所用的时间。

t5:服务端接收了客户端传递来的消息，但由于下一周期还未开始，服务端处于等待状态。

所以我们有 1/f = t1 + t2 + t3 + t4 + t5。

可以看出服务器在一个周期内只有 t1 时间段是处于运算状态，其他时间处于闲置状态，仅仅监听网络。客户端在一个周期内只有 t3 时间段是处于运算状态，其他时间处于闲置状态，仅仅监听网络。

我们来看某次服务端周期的开始：

t1 是固定的，我们不用操心。

t2 是可变的，随着网络情况的变化，t2 有可能变得非常短，这当然是我们乐意看见的，然而如果 t2 变得非常长，比如它跨越了本周期进入了后面周期，以至于本周期内客户端的 Agent 没有收到消息来进行推理。就像图 4-9 显示的那

样。

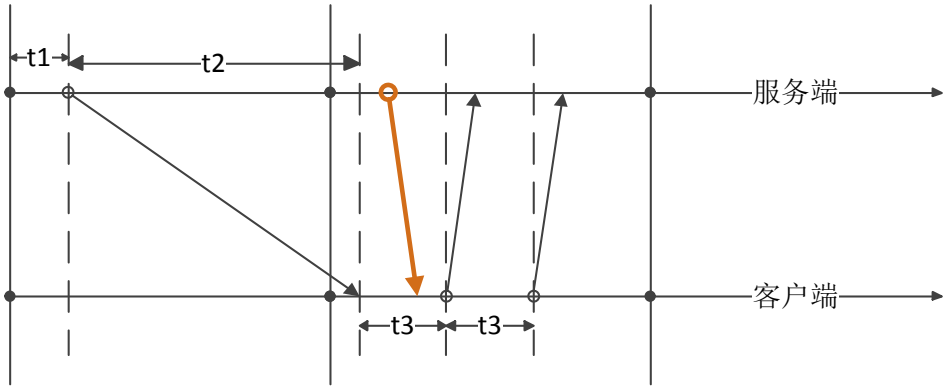

图 4-9

我们看到这条延迟的信息最终还是传过来了，在 t2 的尾端，已经进入了下个周期，客户端的 Agent 们在等待许久之后终于得知了“新的”环境，他们准备进行推理，却不知道这个游戏状态已经过去了一个周期，他们还在对上个周期的事情进行考虑，得出的结论显然是有差错的。这就像拿着过时的藏宝图去寻宝一样，殊不知宝藏早已被转移。

而且这还不是最糟的，我们看到随着 t2 的结束，客户端开始处理那个过时的消息，就在这个时候，更新的服务端消息传输过来了，但客户端的 Agent 们还在忙于前面的处理。等他们用 t3 时间处理完过时的消息，并将反馈发送至服务端，发现客户端输入缓存里又有了新的消息，于是开始用 t3 时间处理这个新的消息并发送至服务端。我们看到在一个周期内客户端发送了两条消息至服务端，于是下个周期服务端将要处理超过一条的消息。由于服务器一个周期只会给每个客户端一次处理机会，累计的消息将留到下一次处理，因此图中情况的频繁出现将会不断累计延迟。

4.7 具体策略

在我们的系统中，服务端是以固定频率 f 运行的，即无论外界情况如何，每秒钟服务端内的游戏状态都会进行 f 次更新，如果客户端发送的动作指令及时到达，服务端就在每次更新前依据这些指令对某些细节进行改变，如果指令没有及

时到达，则认为该周期内球员没有做动作，但对游戏状态的更新仍然进行。

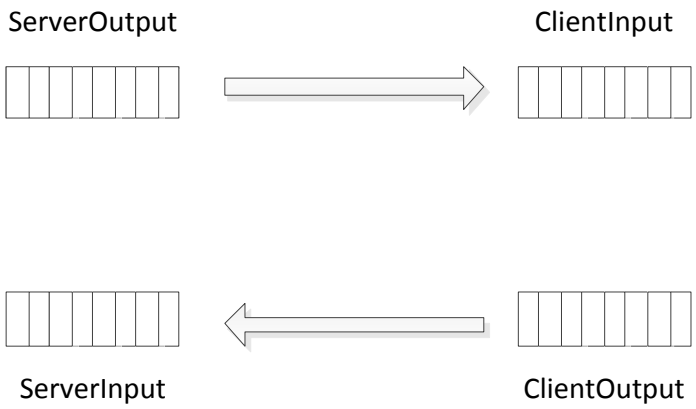

图 4-10

下面我们依次来考虑图 4-10 中的四种情况：

- 服务端输出：由于服务端运行周期固定，因此产生输出信息的周期也是固定的，再加上输出缓存一旦有内容就会被发送出去，我们在实际测试中发现服务端输出缓存的队列长度一直在 0 和 1 之间交替切换，几乎没有超过 1 的情况出现。这也很符合我们的预期：服务端的输出信息应该及时地发送给客户端，输出缓存里不应该出现排队等候的情况。综合考虑我们将服务端输出缓存的队列最大长度设为 1，如果超出则让新的数据覆盖旧的数据。这样即使偶尔有旧的数据没有来得及发送出去，我们也会用最新的数据覆盖它，这样可以避免延迟的累积，有助于提升系统的实时性。
- 客户端输入：当网络状况比较糟糕时，从服务端发送过来的信息可能会产生不可估算的延迟，因为延迟的不稳定性，有时候我们甚至会在收到较新的消息后收到较旧的消息。尽管这些极端情况在我们的测试中很少发生，但为了让系统更加可靠，我们在客户端输入缓存收到数据时对它们的时间戳进行检查，在系统中时间的计量是用周期数标识的，例如“当前时间是第 3240 周期”，表明从游戏开始已经过去了 3240 个周期，如果发现之前已经接收过比这个周期数更大的数据则简单地丢弃这条数据。这是为了保证客户端能尽可能只对最新的数据进行处理，而不浪费时间在旧数据上。
- 客户端输出：前面说到过 Agent 先把自己产生的动作指令存入动作指令缓存，

在所有 Agent 都做完这件事后才把所有动作指令缓存内的数据合并起来放入客户端输出缓存。那么某几个缓慢的 Agent 会拖慢整个客户端的输出速率，我们对 Agent 们每次提交动作指令的顺序进行统计，找出经常性最慢的那几个进行报告以找出问题，并且考虑在今后一段时间里只给一定概率 P 等待它，如果这样子做它还是经常拖慢整个队伍的节奏，继续降低这个概率。和服务端输出一样，我们可以简单地将客户端输出缓存队列最大长度设为 1，这符合实际测试的情况。

- 服务端输入：当网络状况比较糟糕时，从客户端发送过来的信息可能会延迟很久，这就造成了一个很大的问题：客户端发送来的信息是依据之前的游戏状态做出的动作指令，而现在的游戏状态可能已经完全不同了，这时候再套用那些陈旧的动作指令可能会产生匪夷所思的效果，例如球明明已经在身前了，球员仍然跑向球之前的位置去抢球。为了解决这样的问题，我们在客户端输出动作指令时加上其对应的周期数，在服务端输入时检查这个周期数和服务端的周期数，如果相差太多（例如 5 个周期）则直接丢弃。之所以容忍存在周期不匹配的情况，是因为当延迟严重时，旧的动作指令也可以认为“并不那么旧”。当然理想的情况是周期完全匹配，这也是我们在实际测试中经常可以达到的效果。

4.8 本章小结

本章介绍了 Agent 通信系统的具体构造，讨论了在数据通信过程中可能产生的问题，并给出了具体的应对策略。

第5章 基于 Android 平台的系统实现

5.1 总体结构

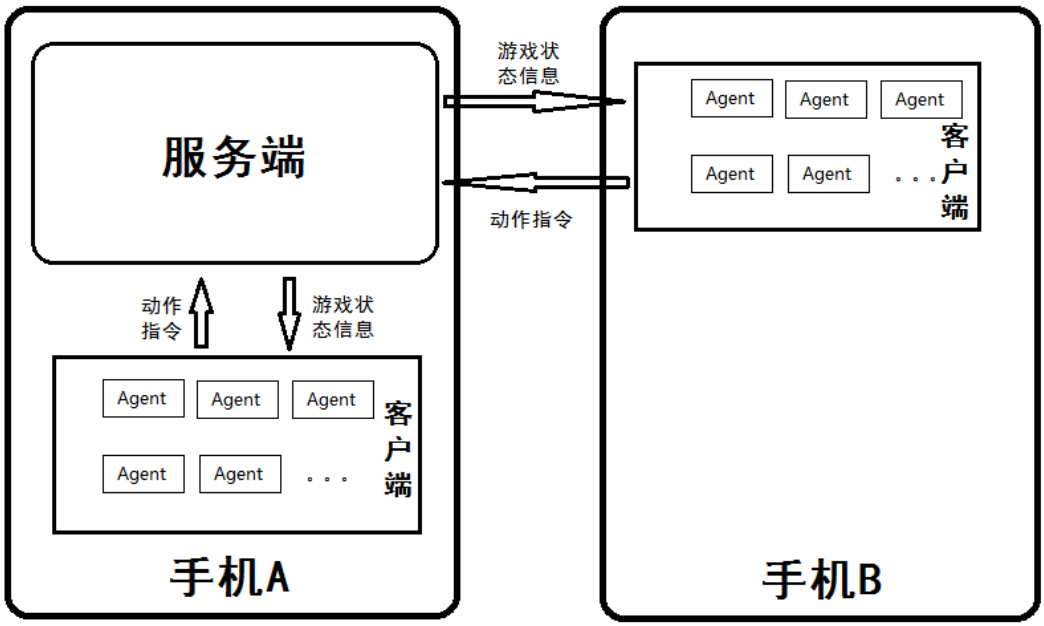

图 5-1

我们在实现过程中将服务端和客户端合并在一起，使得一台手机既可以做服务端又可以做客户端，随后进行两台手机的连接。这有点类似于我们玩 CS 游戏时自己创建游戏后和别人一起玩。图 5-1 是总体结构图。

5.2 Agent 实现

5.2.1 客户端 Agent 实现

Agent 的 UML 设计如图 5-2，side 表示所属的球队，NO 表示在队中唯一的编号，playerRole 表示自己在队中担任的角色，worldState 则是自己对整个游戏状态的认识。包含的方法这里只列出了重要的几个，updateState()用于更新 worldState，reasoing()用于根据现有 worldState 进行推理产生动作指令，output()用于将产生的动作指令输出到指令缓存。调用 act()可以让 Agent 进行一次完整的行动，包括更新状态信息，根据状态信息推理产生指令和输出指令。

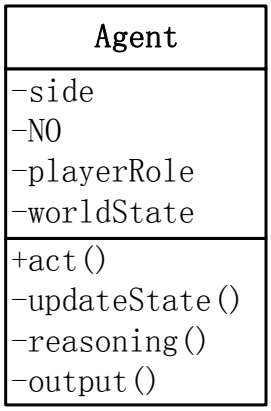

图 5-2

WorldState 包含了 Agent 推理所需的所有游戏信息，包括游戏时间(time)，本方得分(ownScore)，对方得分(oppoScore)，球场物理信息(pitch)，本方球员信息数组(player0)，对方球员信息数组(player1)，球的状态(ball)，每个元素都有自己的 get 方法。如图 5-3。

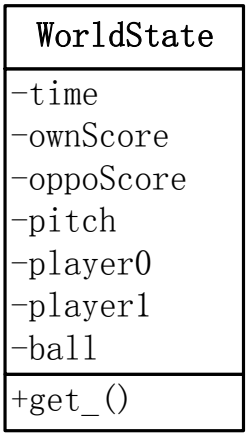

图 5-3

Algorithm 5-1 显示在 reasoning()中调用了 3.7.2 节介绍的几种算法，根据 Agent 的角色得到推理出的动作指令。

Algorithm 5-1

```
switch(球员角色)

case 守门员：
        相应算法(Algorithm3-1)
case 中后卫：
        相应算法(Algorithm3-2)
case 边后卫：
        相应算法(Algorithm3-3)
case 中场：
        相应算法(Algorithm3-4)
case 边锋：
        相应算法(Algorithm3-5)
case 前锋：
        相应算法(Algorithm3-6)

返回推理得出的指令
```

5.2.2 服务端 Agent 指令执行

动作指令传输到服务端后被执行，分别调用 executeDash()和 executeKick()方法用于执行 dash 和 kick 指令。Listing 5-1 显示了 executeDash()的代码，将 power 转换为球员可以获得的加速度后，利用 angle 得到 x 和 y 方向的加速度，然后为球员进行加速，加速过程是简单的加法运算，即在当前速度基础上加上 ax 和 ay。

Listing 5-1

```
/**
 *
 * @param side
 *            球员所属球队
 * @param NO
 *            球员编号
 * @param angle
 *            跑动方向，PI制
 * @param power
 *            跑动力量
 */
private void executeDash(Side side, int NO, double angle, double power){
        //根据DASH_POWER_TO_ACC将跑动力量power转换成球员获得的加速度acc
        double acc = power / DASH_POWER_TO_ACC;
        //根据acc和angle获得x和y方向的加速度
        double ax = ac * Math.cos(angle);
        double ay = ac * Math.sin(angle);
        //将球员根据(ax, ay)进行加速
        this.makePlayerAcc(side, NO, ax, ay);
}
```

Listing 5-2 显示了 executeKick()的代码，首先检查踢球者和球的距离是否在

可踢范围之内，否则不执行这个 kick 指令。然后将 power 转换为球可以获得的加速度后，利用 angle 得到 x 和 y 方向的加速度，然后为球进行加速，加速过程是简单的加法运算，即在当前速度基础上加上 ax 和 ay。

Listing 5-2

```
/**
 *
 * @param side
 *            球员所属球队
 * @param NO
 *            球员编号
 * @param angle
 *            踢球方向，PI制
 * @param power
 *            踢球力量
 */
private void executeKick(Side side, int NO, double angle, double power){
        //获得踢球球员和球的坐标信息并计算两者距离
        double dsts2ball = Util.calDistance(this.pitch.getPlayer(side, NO),
                                            this.pitch.getBall());
        //球在球员的可踢范围kickableMargin内才踢球，否则不执行踢球指令，并输出指示信息
        if(dsts2ball <= kickableMargin){
            //根据KICK_POWER_TO_ACC将踢球力量power转换成球获得的加速度acc
            double acc = power * KICK_POWER_TO_ACC;
            //根据acc和angle获得x和y方向的加速度
            double ax = acc * Math.cos(angle);
            double ay = acc * Math.sin(angle);
            //将球根据(ax, ay)进行加速
            this.makeBallAcc(ax, ay);
        } else{
            System.out.println("【" + side +"," + NO +" miss a kick】");
        }
}
```

在更新了球员和球的速度后，服务端需要根据当前球员和球的位置、速度信息，进行一次推算，即将游戏状态推入下一周期。这里调用了 calMovingObjectNextCycle()方法，球员和球都是移动物体，作为参数输入该方法。具体代码见 Listing 5-3。我们看到计算下周期坐标的方法就是在当前坐标值的基础上加上速度的值。另外由于球场阻力的存在需要在计算完位置信息后对移动物体进行减速，否则我们就违反了牛顿第二定律。

Listing 5-3

```
private void calMovingObjectNextCycle(MovingObject object) {
        // 下一周期的X坐标
        double nextX;
        // 下一周期的Y坐标
        double nextY;
        nextX = object.getPosition().getX() + object.getSpeed().getSpeedX();
        nextY = object.getPosition().getY() + object.getSpeed().getSpeedY();
        object.setPosition(nextX, nextY);
        // 根据摩擦力的进行减速
        speedDecay(object);
}
```

Listing 5-4 显示了计算所有移动物体下一周期状态的代码，需要注意的是计算球和球员的方法就是调用的 Listing 5-3。我们在计算后还需要修正一些碰撞事件，因为球和球员都是有一定大小的，如果两个物体有重叠部分，我们认为他们之间发生了碰撞，处理的方法是各自按照原速度的 0.1 倍回退，直到不再发生碰撞为止。

Listing 5-4

```
public void calAllObjectsNextCycle() {
    // 计算下一周期球的状态
    calBallNextCycle();
    // 计算下一周期球两队球员的状态
    for (int i = 0; i < numOfPlayer; i++) {
        calPlayerNextCycle(Side.LEFT, i);
        calPlayerNextCycle(Side.RIGHT, i);
    }
    // 修正球员之间的碰撞事件
    amendPlayerCrash();
    // 修正球和球员的碰撞事件
    amendBallHit();
}
```

因此 Agent 动作指令到达服务端后先是被执行，这样就更新了球和球员的速度信息。随后根据球和球员的位置、速度信息将游戏状态推入下一周期，在这一计算过程中位置和速度都有所改变，位置改变是因为速度的作用，速度改变是因为球场阻力的作用。

5.3 Agent 通信实现

5.3.1 服务端输入

服务端监视服务端 socket 输入流，一旦有数据传入，就将其放入缓存池中相应的缓存中。该监视者实现了 Runnable 接口，将在服务端开启后运行于独立的线程中，其 run()方法伪代码展示于 Listing 4-1。

Listing 5-5

```
public void run() {
        while (服务端在运行中) {
            String s = socket上读入的utf字符
                (此方法在没有输入流时会自动阻塞);
            将s存入缓存池中相应缓存中;
        }
}
```

在将字符串形式的 Agent 动作指令存入输入缓存前，如果该指令过于“陈旧”，则直接丢弃。

5.3.2 服务端输出

服务端监视服务端输出缓存，只要输出缓存中有数据，就将其通过 socket 发

送至相应客户端，并在缓存中清空此数据。监视者实现了 Runnable 接口，将在服务端开启后运行于独立的线程中，其 run()方法伪代码显示于 Listing 5-6。

Listing 5-6

```
public void run() {
        while (服务端在运行中) {
            String s = 缓存池中取出的字符串
                (此方法在缓存为空时会自动阻塞);
            将s通过socket发送到相应客户端;
        }
}
```

5.3.3 客户端输入

客户端监视客户端 socket 输入流，一旦有数据传入，就将其存储于输入缓存。监视者实现了 Runnable 接口，将在服务端开启后运行于独立的线程中，其 run()方法伪代码显示于 Listing 5-7。

Listing 5-7

```
public void run() {
        while (客户端在运行中) {
            String s = socket上读入的utf字符
         (此方法在没有输入流时会自动阻塞);
            将s存入缓存中;
        }
}
```

在将字符串形式的游戏状态信息存入输入缓存前，如果该状态的周期数低于之前已经接收到的状态信息，则没有必要进行处理可以直接丢弃。

5.3.4 客户端输出

客户端监视客户端输出缓存，只要输出缓存中有数据，就将其通过 socket 发送至服务端，并在缓存中清空此数据。监视者实现了 Runnable 接口，将在服务端开启后运行于独立的线程中，其 run()方法伪代码显示于 Listing 5-8。

Listing 5-8

```
public void run() {
        while (客户端在运行中) {
            String s = 缓存中取出的字符串
                (此方法在缓存为空时会自动阻塞);
            将s通过socket发送至服务端;
        }
}
```

5.4 运行情况

5.4.1 开始游戏

我们在手机 A 上输入服务端的监听端口（例如 5678），然后点击“开启服务端”，服务端就已经开始运行（这其实是在后台运行了一个 SERVICE［6］），等待客户端全部连接上后进行模拟的足球比赛。如果不想再接受服务请求，点击“关闭服务端”。如图 5-2。

图 5-2

我们准备在手机 A 和 B 上运行客户端，输入手机 A 的 IP 地址和它的监听端口，点击“开启客户端”可以进行连接。如图 5-3。

图 5-3

两个客户端全部连接上服务端后，模拟的足球比赛就开始了。两台手机的屏幕上都会显示出比赛进行的实时情况。显示情况如图 5-4。

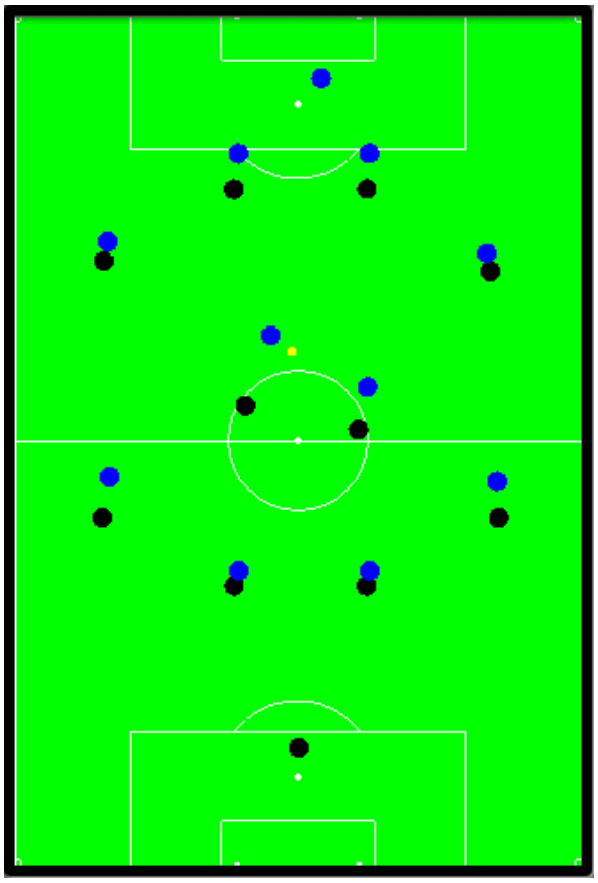

图 5-4

5.4.2 阵型展示

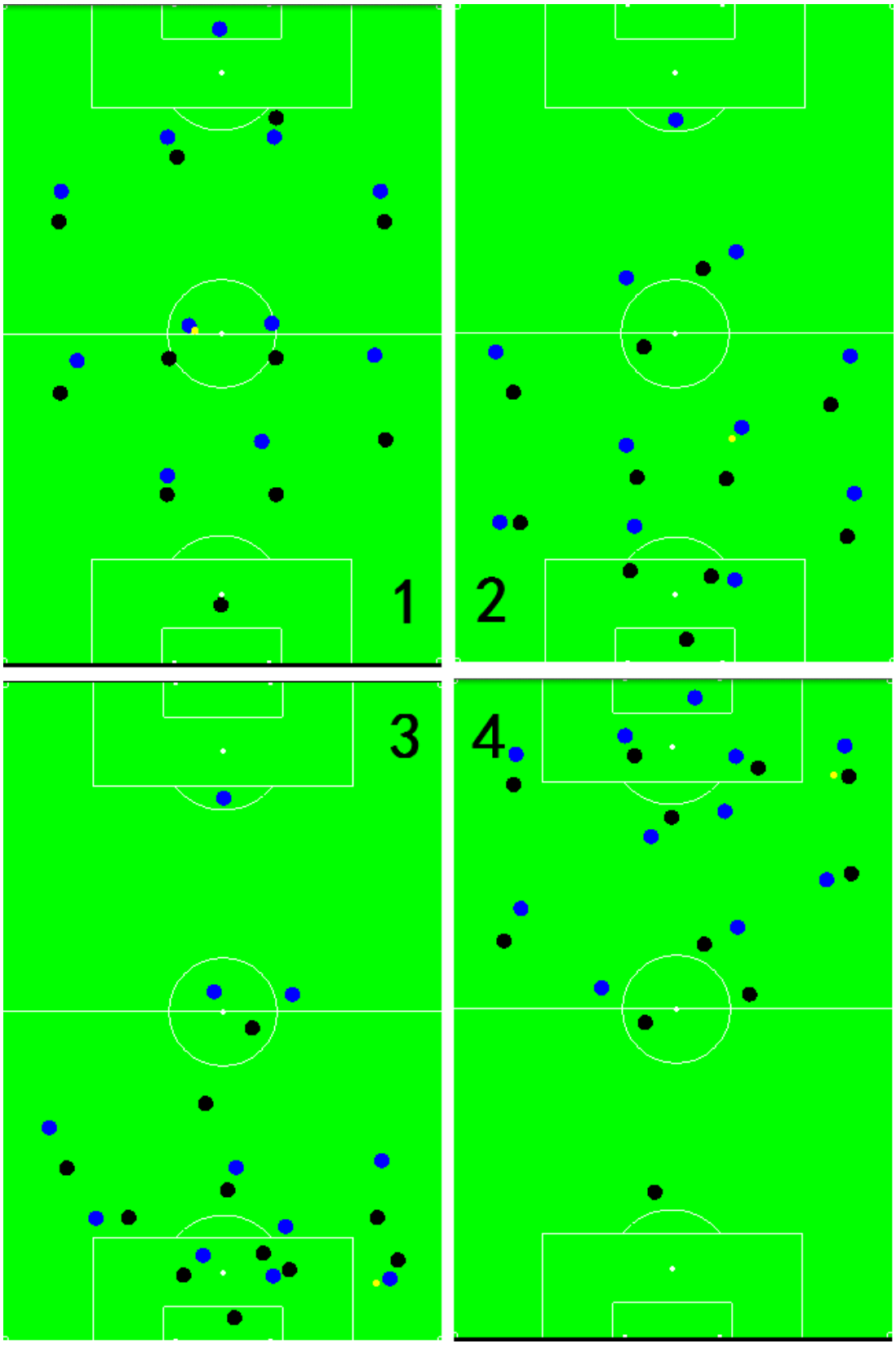

图 5-5

图 5-5（1）显示蓝方在中圈区域带球，双方处于均势，站位比较平衡，但由于蓝方马上就要发起进攻，双方阵型即将发生变化。图 5-5（2）显示蓝方中场球员已经带球进入黑方半场，蓝方整体阵型前压，门将甚至都已经站到了禁区之外，此时黑方整体阵型后撤，基本覆盖住了蓝方的进攻力量。图 5-5（3）显示蓝方经过一些传递将球发展到了对方右路防区的底部，此时蓝方前锋已经进入对方禁区等待射门机会，其他球员也前压的很靠上，仅留两个中后卫和门将留在本方半场防守，这样敢于在进攻中投入兵力会增加进攻的威胁性，而黑方除了整体阵型收缩防守外，也保留了一到两个球员在前面牵制对方，伺机反击。图 5-5（4）则显示了黑方进攻时的场景，和蓝方进攻时的情形相差无几。

5.4.3 传球展示

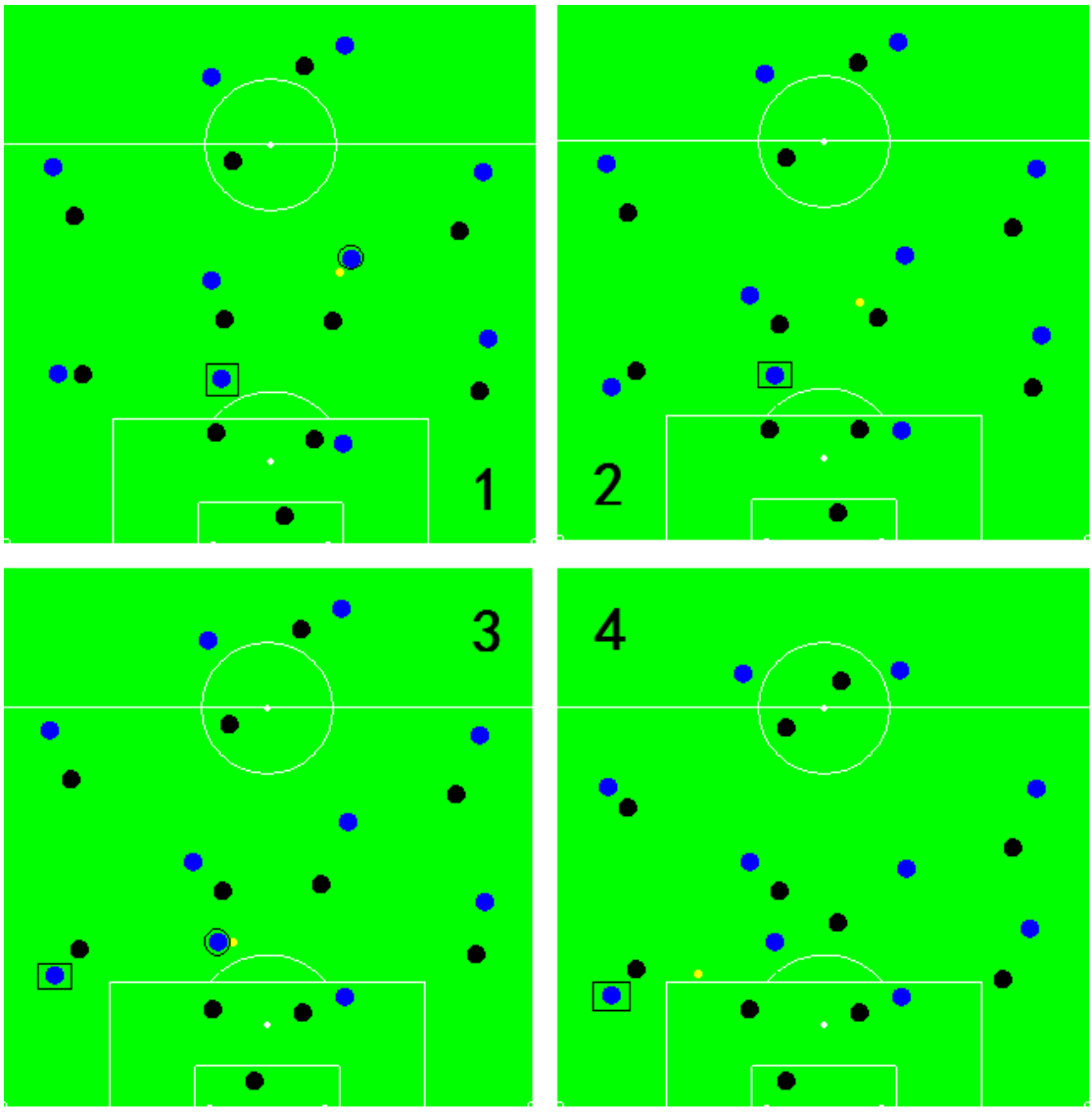

图 5-6

图 5-6（1）显示圆圈标识的蓝方中场球员准备传球，在经过 3.6.3 节中介绍的传球选择后，他选择了方块标识的球员作为传球目标，考虑到传球威胁性，尽

量往前传的话，禁区内的那个队友最靠前，但是我们看到有一个黑方球员就在传球路线不远处，很有可能截断这个方向的传球，传给右路的边锋没有被拦截的可能，但是他不够靠前。综合考虑我们选择传给禁区弧顶的那个前锋，这条传球路线不太可能被对手拦截，这个前锋比较靠前，而且他离防守球员有一定距离，拿到球后还能进行一些动作。图 5-6（2）显示球已经传向了方块标识的球员。图 5-6（3）显示接球球员拿到球之后，发现左路有一个本方球员摆脱了对手的防守找到了一个很大的空档，因此接球球员决定将球传给这个边路球员。图 5-6（4）显示经过一番调整，球已经传向了拉出空档的边路球员，如果他顺利拿到球，可以选择传中威胁对方球门。

5.4.4 射门展示

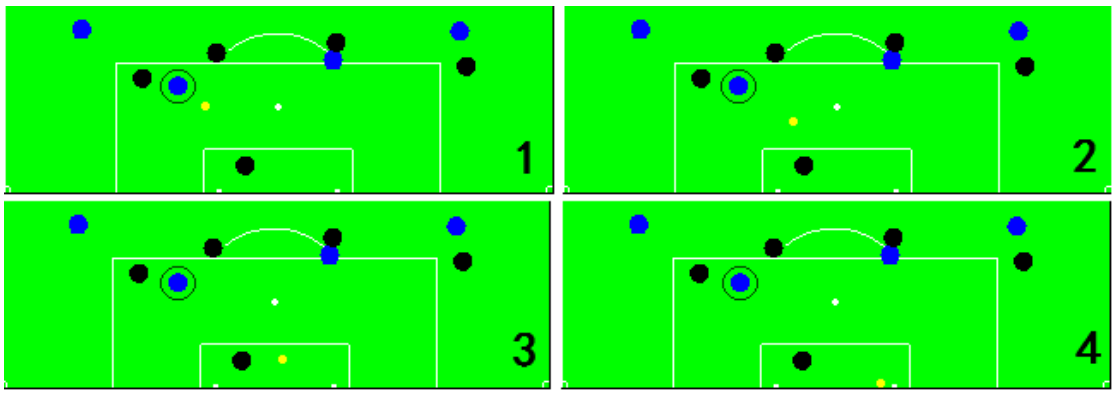

图 5-7

图 5-7 展示了球员射门的场景。我们在图 5-7（1）中看到圆圈指示的蓝方球员已经面对门将处于单刀之势，这时他果断选择起脚射门，并在射门前做了 3.5.2 节中介绍的射门角度选择。我们可以从图 5-7（2）中看到他选择射向远门柱方向，因为此时门将几乎封锁了近门柱的射门角度，却把远门柱方向漏出很多空间。从图 5-7（3）（4）中看到由于球的运行路线离门将较远，躲过了门将的扑救，并成功打入右侧球门范围。蓝方进球。

5.4.5 过人展示

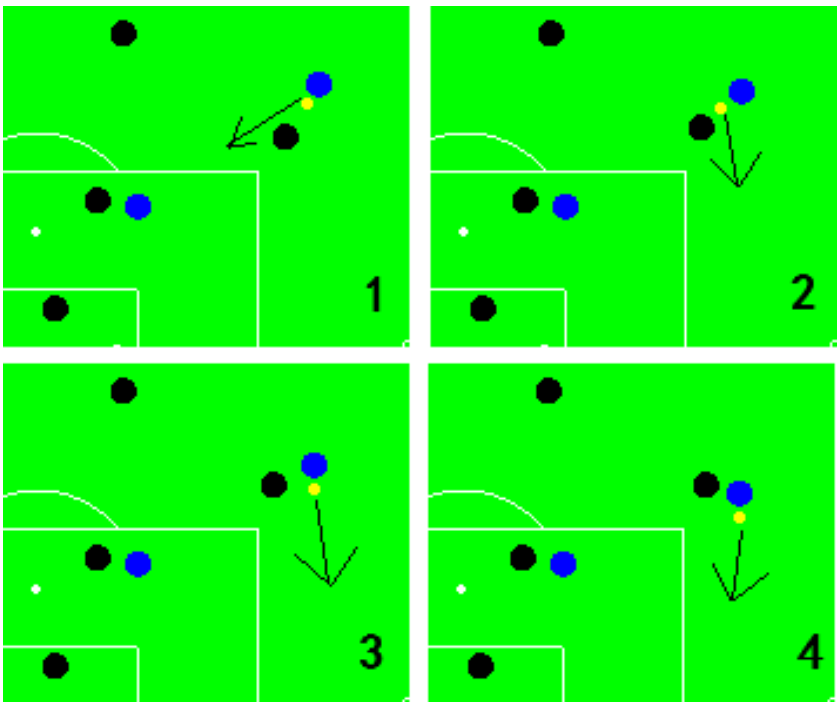

图 5-8

图 5-8（1）显示一位蓝方球员想要在边路寻求突破过人，他面前有一个对方防守球员，一开始他选择箭头指向的方向去，也许他在经过 3.6.3 节介绍的突破过人算法计算后觉得那样比较好，然而防守球员也跟着他移动了，图 5-8（2）显示此时两人离得很近，蓝方球员发现对方经过移动后，在右侧给自己留下了一个空档，于是决定利用这个空档将球踢向箭头方向，这样就形成了图 5-8（3）的情形，防守球员被蓝方球员的这一个变向所欺骗，没有及时反应过来，等到他想回追的时候已经到了图 5-8（4）的状态，蓝方球员突破了防守并继续向前带球。需要注意的是，由于双方球员的速度都是一样的，这样的直接突破成功率并不高，很多时候在图 5-8（2）的情形时如果球离防守球员太近则很容易被破坏掉。这时候选择通过传球来突破防守是比较好的选择，这也是符合足球运动一般规律的。

5.5 本章小结

我们在本章介绍了我们的系统在 Android 平台上的实现情况，并展示了使用方法和运行情况。

第6章 结束语

6.1 论文的主要工作

- 我们将分布式多 Agent 系统运用于机器人足球游戏的实现，为每个球员配备一个 Agent，将代表两个球队的两组 Agent 和球场环境状态联系起来构成多 Agent 系统。这样提升了 Agent 通信的有效性，也保证了 Agent 决策的及时性和正确性。
- 我们研究了分布式多 Agent 系统中的网络连接问题，为 Agent 与服务端之间的通信制定了策略以应对网络延迟给系统带来的危害，增强了系统的实时性。
- 我们研究了 Agent 设计方法，基于反应式 Agent 架构设计了一套关联角色的决策系统。该系统分配给每个 Agent 不同角色并让其承担相应任务，帮助我们分角色完善了决策系统的正确性。
- 最后我们在 Android 平台上实现了基于分布式多 Agent 系统的机器人足球游戏，将 Agent 按照队伍不同分布到不同的客户端手机上，每个客户端负责本方 Agent 的决策和通信功能，并连接到服务于所有客户端的服务端手机以进行数据通信。我们的系统在实际运行中体现出了高效的通信能力和良好的决策能力。

6.2 论文进一步的工作

- 反应式 Agent 能力有限，增加学习能力后的机器人足球运动员可以更好地应对变化的环境。
- 我们在实现时将一组 Agent 合并置于一个客户端内，在扩展的情况下可以让每个 Agent 成为一个客户端，并将所有客户端连接起来。这对 Agent 通信方面的要求将会更高。

参考文献

[1] 王万森，《人工智能原理及其应用》，电子工业出版社，2010.

[2] Remco de Boer & Jelle Kok. *The Incremental Development of a Synthetic Multi-Agent System: The UvA Trilearn 2001 Robotic Soccer Simulation Team*. University of Amsterdam, Feb 2002.

[3] http://en.wikipedia.org/wiki/Multi-agent_system.

[4] http://en.wikipedia.org/wiki/Intelligent_agent.

[5] Oussama Khatib. *Real-Time obstacle Avoidance for Manipulators and Mobile Robots*. Proc of The 1994 IEEE.

[6] http://developer.android.com/reference/android/app/Service.html.

致谢

经过几个月的工作最终完成了论文，首先要感谢的是我的导师曹春老师，他在这期间提供了很多指导性意见，理顺了整个工作的方向，并且在繁忙的工作中抽出时间审阅了论文的许多细节部分，在这里要特别感谢他辛勤的付出。

做这个系统的想法源自对足球经理游戏的热爱，在此向开发者 Sports Interactive 公司致敬，感谢你们让我对模拟类足球游戏产生了浓厚的兴趣。我和田伟宇的几次讨论也促成了这项工作的大致思路，感谢你给了我很多灵感。同时也要感谢这段时间一直陪伴我的女友。

在开发过程中，杨名阳和我交流了 Android 平台的学习经验，薛昊和许德华在开发细节方面给了我很多启发。整个四月和五月，钱行、刘华艇和钱煜陪我见证了令人难以置信的欧冠联赛和英超联赛，这让我对自己的工作产生了更大的激情。在此一并对他们表示感谢。