中国研究生创新实践系列大赛

“华为杯”第二十届中国研究生数学建模竞赛

学　　校	上海交通大学
参赛队号	23102480015
队员姓名	1. 税昀昊
	2. 陈晓萌
	3. 曾卓

中国研究生创新实践系列大赛
“华为杯”第二十届中国研究生数学建模竞赛

题　　目　**基于 Markov chain 的 WLAN 网络信道接入机制建模**

摘　　要：

本文主要从 CSMA/CA 机制的建立方面来实现分布式协调功能。在模型建立上面，主要是基于 Markov chain 建模。对于不同情况下的网络信道接入机制建模问题，我们主要从节点互听和传输成功与否两个方面进行深入研究，节点互听问题实质上是多个 BSS 距离导致的信道占用判定问题；传输能否成功的问题本质上是受 SIR 影响的互相干扰问题。在基于这两个方面产生的多种复杂传输情况下，我们发现经典的 Bianchi 模型[1] 不能正确的反映实际情况，因此我们深入研究了多种传输情况下的基于改进 Markov chain 的机制建模，最终实现多种情况下的 WLAN 网络接入信道机制的模型建立。

对于**问题**一，要求建模两 BSS 互听情况下并发传输不能成功时的吞吐。针对上述问题，我们采用经典的 Bianchi 模型为基础，对信道状态建模，从而求解出信道各状态的分布情况，最终评估系统的吞吐，其中归一化吞吐为：0.14738，系统吞吐量为：67.17442Mbps。

对于**问题**二，由于并发传输必定成功，这时信道在空闲和传输成功两种状态转移，所以我们将第二问建模为一维 Markov chain，，并最终求出信道各状态的概率，从而最终得到归一化吞吐为：0.24122，系统吞吐，我们考虑了并发时传输信息量为 2 倍，则系统吞吐为：70.558Mbps。

对于**问题**三，需要对两 BSS 不互听情况建模，并且需要考虑信道干扰。对于该问题，我们在基本模型上针对不同情况下的重要参数区别进行了分析，重构了模型状态转移概率表达式。此外，我们还提出了新的基于交叠情况下的信道吞吐算法，通过对不同交叠情况在 Markov chain 中的稳态表达进行求解即可实现最终系统归一化吞吐和系统吞吐量的计算。

对于**问题四**，主要目标是对 3BSS 模型进行建模。针对这一问题，我们采取分类讨论的方法，先确定传输成功和失败时分别会有哪些可能 BSS 组合，然后得到分别的概率，最后，利用前三个问题中对多 BSS 模型进行的数学建模进行计算求解，得到系统归一化吞吐。

关键字：网络信道　马尔科夫链　分布式协调　状态机

1. 问题重述

1.1 问题的背景

无线局域网在生活中被广泛使用，主要目的是提供低成本、高吞吐和便利的无线通信服务。基本服务集是 WLAN 的基本组成部分。处于某一特定覆盖区域内的站点（STA, station）与一个专职管理 BSS 的无线接入点（AP, access point）组成一个 BSS，称 STA 关联到 AP。常见的 AP 有无线路由器、WiFi 热点等，而手机、笔记本、物联设备等是 STA。AP 给 STA 发送数据叫作下行方向，反之称为上行方向，本文将 AP 和 STA 统称为节点，每个节点的发送和接收不能同时发生。各节点共享信道，通过载波侦听多址接入/退避的机制避免冲突，称为分布式协调功能。DCF 在日常无线局域网组网中具有广泛的应用。

DCF 机制提供了一种分布式、基于竞争的信道接入功能。可将每个节点接入信道进行数据传输的过程分为 3 个阶段，信道可用性评估（CCA，clear channel assessment）、随机回退、数据传输。

目前讨论最深入，最具代表性的 DCF 机制模型是 Bianchi 基于 Markov chain 建模。这一模型主要基于单 BSS 情况，具体来说，当 2 个及以上节点同时回退到 0 发送数据时，由于碰撞而丢包。那么信道可能处于三种状态：空闲、成功传输、碰撞，将每个状态看作一个虚拟时隙，那么信道在三种虚拟时隙中转化。将退避器所处的阶数和随机回退数用二维 Markov chain 表示，推导节点在每个虚拟时隙的发送概率 τ 和发生碰撞的条件概率 p，从而评估 BSS 的吞吐[1]。

在 Bianchi 模型中，假设了单 BSS、理想信道。但在实际应用的场景中，传输情况则较为复杂，不仅需要考虑多个 BSS 之间的互听和同频干扰，也需要考虑信道非理想状态下的干扰。因此，亟需通过优化改进已有的模型以适应不同的传输情况。

1.2 问题的提出

1.2.1 问题的提出内容一

考虑两个 BSS 互听的场景，仅下行，即两个 AP 分别向各自关联的 STA 发送数据。以 AP1->STA1 方向的数据传输为例，其会受到相邻 BSS2 的干扰，对于 STA1 来说，AP1->STA1 是信号，AP2->STA1 是干扰。对于 AP2->STA2 情况类似。假设 ACK 一定能发送成功。根据节点之间的 RSSI 估算两个 AP 并发时的 SIR，考虑不同的情景进行建模。

问题一：针对提供的传输参数。考虑数据传输一定成功，且两个 BSS 并发传输时，一定会导致两个 AP 的数据传输都失败。请对该 2BSS 系统进行建模，用数值分析方法求解，评估系统的吞吐。

问题二：针对提供的传输参数。考虑当两个 BSS 发生并发传输时，两个 AP 的数据传

输都会成功，其他条件同问题一。请对该 2BSS 系统进行建模，用数值分析方法求解，评估系统的吞吐。

1.2.2 问题的提出内容二

考虑两个 BSS 不互听的场景，仅下行。AP 认为信道空闲，因此总是在退避和发送数据。可以预见的是，有很大概率出现二者同时或先后开始发送数据的情况。此时信号包接收成功与否由 SIR 决定。基于这个场景，进行问题的建模。

问题三：针对提供的传输参数。假设因信道质量导致的丢包率 $P_e = 10\%$。当两个 AP 发包在时间上有交叠时，假设 SIR 比较小，会导致两个 AP 的发包均失败。请对该 2BSS 系统进行建模，尽量用数值分析方法求解，评估系统的吞吐。

1.2.3 问题的提出内容三

考虑 3BSS 场景。场景中，AP1 与 AP3 不互听，AP2 与两者都互听，可以预见的是，AP2 的发送机会被 AP1 和 AP3 挤占。AP1 与 AP3 由于不互听可能同时或先后发送数据。

问题四：在上述场景中，针对给出的传输参数，假设 AP1 和 AP3 发包时间交叠时，两者发送均成功。请对该系统进行建模，尽量用数值分析方法求解，评估系统的吞吐。

2. 模型假设与符号说明

2.1 模型假设

- **单信道网络**：即所有节点共享一个信道，在不额外说明的情况下，这个网络中的所有节点都是可以互相监听到对方的；
- **忽略传播时间**：由于电磁波传播的速度很快，并且在该问题下，无明显距离很远的说明，故在此忽略传播的时间；
- **统一数据帧长度**：即所有节点传输的数据帧长度相同，冲突与否不影响数据帧传输长度；
- **节点传输碰撞过程可解耦**：即无论节点重传了几次，其在传输时候的冲突概率都是一个定值，且都是相对独立的；
- **信道状态理想**：在题目没有额外说明时，信道假设为一个理想信道；
- **饱和吞吐量**：本文中所建立模型为饱和情况下的吞吐量，即任意一个节点都是假设有数据包的，不存在节点竞争到信道之后，没有数据包待发送的情况出现；
- **丢包设置**：在本文模型中，设定重传次数超上限时，无论传输成功与否，数据帧均会被丢弃。

2.2 符号说明

符号	意义
τ	发送概率（此时节点退避计数器归零）
p	条件传输失败概率（退避计数器升阶）
δ	协议中标准时隙长度
P_{tr}	传输概率
P_s	条件传输成功概率
T_s	成功传输花费时间
T_c	传输失败花费时间
S^*	归一化吞吐
r	最大重传次数
W_i	第 i 阶竞争窗口大小
$b_{i,k}$	当第 i 次重传，退避计数器到达 k 时节点状态

3. 问题分析

3.1 问题一分析

该问题要求分析两同频且互听的 AP 分别向其对应的 STA 发送数据情况下的吞吐，对于该系统，在随机回避阶段，如果退避计数器同时归零出现并发的时候传输一定会失败。在此基础上建立模型，首先可以确定的是由于是同频的多 BSS 系统，所以他们在一个信道上进行传输，当信道上正在传输某一个 AP 发出的数据时，另一个 AP 将会监听到其正在传输，所以其判断为繁忙，当信道空闲时，二者会进入随机回退阶段，从而可能造成并发的情况，在并发时，信道上容纳了两者发出的数据，在问题一的假设下，这时由于 SIR 较低，造成了数据传输都失败，即重新进入随机回退。

概括上述内容，类似于 Bianchi 模型，信道的状态分别由三部分组成：空闲、传输成功、传输失败（碰撞）。那么在此处，对于多 BSS 系统，同频的 n 个 AP 向 STA 发送下行数据时，将上述的每个状态看作一个虚拟时隙，那么信道在三种虚拟时隙中转化，随机回避阶段中的退避器状态转移是一个二维随机过程 $\{b(t), s(t)\}$，可以用 Markov chain 表示。

因此，我们的模型基于二维 Markov chain，在对于 AP 传输的不同状态转移进行建模后，得到的 Markov chain 具有稳态解，在求稳态解的过程中，我们得到了能够求解重要未知传输参数的二元非线性方程，从而允许我们进一步求解系统吞吐。

3.2 问题二分析

问题二为两个 BSS 并发传输时，由于 SIR 高，其传输一定成功，其他条件与问题一相同，重新建立模型。

在本问题中，并发传输通过较高的 SIR 保证了接收机的一定成功。在这种情况下，信道中将不会存在“碰撞”这一状态。由此在问题一所建立的二维 Markov chain 中，不会存在碰撞，即不会存在二进制指数退避的发生而升阶的转移过程，即碰撞概率 p 为 0，从而竞争窗口 (CW, contention window) 的大小维持初值不变。所以信道状态有两种：空闲、传输成功。我们在问题一中所建立的 Markov chain 将退化为一维 Markov chain。因此，可以针对变化后的一维 Markov chain，利用其同样具有稳态解的特性，再次如问题一分析中思路求解。

3.3 问题三分析

相比之前的问题，问题三提供了一个传输情况更加复杂的环境模型，在问题三中，同样为同频干扰，但是分析 AP 之间的 RSSI（-90dBm<-84dBm），两 AP 不互听，那么二者发送数据时会始终认为信道处于空闲状态，所以即使有一方的 STA 在接受数据的过程中，另一个 AP 仍然有可能继续发送数据，这与互听的情况不同，互听时会监测到信道繁忙，并停止发送。所以，此时存在信号交叠的情况，此时由于题目条件 SIR 较小，所以发送会均

失败。同时还需要引入由于信道质量导致丢包的传输失败。

所以，在此情况下，信道的状态可能有以下三种：空闲、传输成功、传输失败 (交叠或丢包)。同样的建立二维 Markov chain，其中退避计数器升阶的概率在此处为信号交叠和丢包导致失败的概率。类似的，求出其稳态解，从而可以得出吞吐量。

3.4 问题四分析

问题四将传输推广至 3BSS 场景，其中 AP2 可以同时和 AP1、AP3 互听，AP1 和 AP3 不互听，但是 AP1 和 AP3 交叠时，由于 SIR 较大，两者发送均成功。问题四和上述问题最大的不同点在于 AP 存在不对称性，由于互听关系，AP1 和 AP3 是等价的，他们都与 AP2 互听但彼此不互听，但是 AP2 和 AP1、AP3 不等价，AP2 与 AP1、AP3 互听。分析该情况，当系统建立完成时，其吞吐则是确定的，信道状态同样由三个状态构成：空闲、传输成功、传输失败（AP2 与 AP1、AP3 任意一个发生或两个碰撞），但是由于 AP 之间的不对称性，我们无法只通过一个 Markov chain 模型描述该系统状态的变化。所以我们在问题四中建立了两个二维 Markov chain 用于分别描述 AP1、AP3 和 AP2。对于 AP1、AP3 的 Markov chain，其退避指数器升阶的情况为 AP1（或 AP3）与 AP2 碰撞，对于 AP2 的 Markov chain，器退避指数器升阶的情况为 AP2 与 AP1、AP3 的其中一个或两个碰撞。那么依据此，可以求解出这两个 Markov chain 的相关参数，从而求解出系统吞吐。

4. 基本模型

4.1 随机回退

在传输过程中，“碰撞” 这一过程既是信道可能存在的三个状态之一，同时也是多 BSS 模型下必然会出现的可能状况。而实现碰撞过程对节点传输状态的影响，以及三个虚拟时隙的转化，则依赖于随机回退的实现。

随机回退的过程可以被描述为：信道空闲时，可能有多个节点准备好了数据，为避免碰撞，节点从 [0, CW-1] 中等概率地选取一个随机数作为回退数，等待该回退数个 δ（本文中为 9μs），随机回退时段时长为回退数乘以 δ。如果信道在随机回退时段保持空闲，则节点开始一次数据传输，传输失败则 CW 翻倍。如果 CW 达到了 CW_{max}，则保持此值，直到被重置为止。每次数据传输成功时 CW 重置，开始下一次回退。若传输连续失败，达到 r 次重传则数据帧被丢弃，CW 重置传输下一个数据帧。

在随机回退时段节点持续监听信道，如果期间信道变繁忙，则节点将回退暂停，直到信道在一个 DIFS 时长重新变为空闲，再继续前面没有回退完的时间。

4.2 传输模型

对于单 BSS，n 个 STA 给 AP 发送上行数据，由于信道只有一个所以在统筹信号传输时采用随机回退机制，可以基于 Markov chain 建模。**在本文中，由于我们已在模型假设中提到，本文是基于问题一到四进行的讨论，由于他们都是同频下的信号传输，所以信道始终只有一个，因此我们仍然可以对该单一的信道利用 Markov chain 对多 BSS 中的节点进行建模**。对于每一个问题中的情况，我们基本的建模思路是认为信道有三种状态：空闲、成功传输以及传输冲突。将每一个状态看作一个虚拟时隙，那么信道将会在三种虚拟时隙中转化。对于每一个节点，求解出它在当前时刻，也就是每个虚拟时隙的发送概率 τ 和传输发生冲突导致失败的条件概率 p，即可评估 BSS 系统的吞吐。

令 $b(t)$ 和 $s(t)$ 代表 t 时刻一个节点退避随机过程的退避计数和退避阶数，这里的 t 是一个离散的虚拟时隙的开始时刻。用 i 表示一个数据的发送次数，也叫作阶数，r 为最大重传次数，m 是最大退避阶数，则 CW 可用公式(4-1)表示：

$$\begin{cases} W_i = 2^i W_0, & 0 \le i \ge m \\ W_i = 2^m W_0, & m \le i \ge r \end{cases} \tag{4-1}$$

基于 Markov chain 推导 τ 和 p 的过程如下：

二维 $\{b(t), s(t)\}$ 随机过程可用二维 Markov chain 表示，如图 4.1 所示

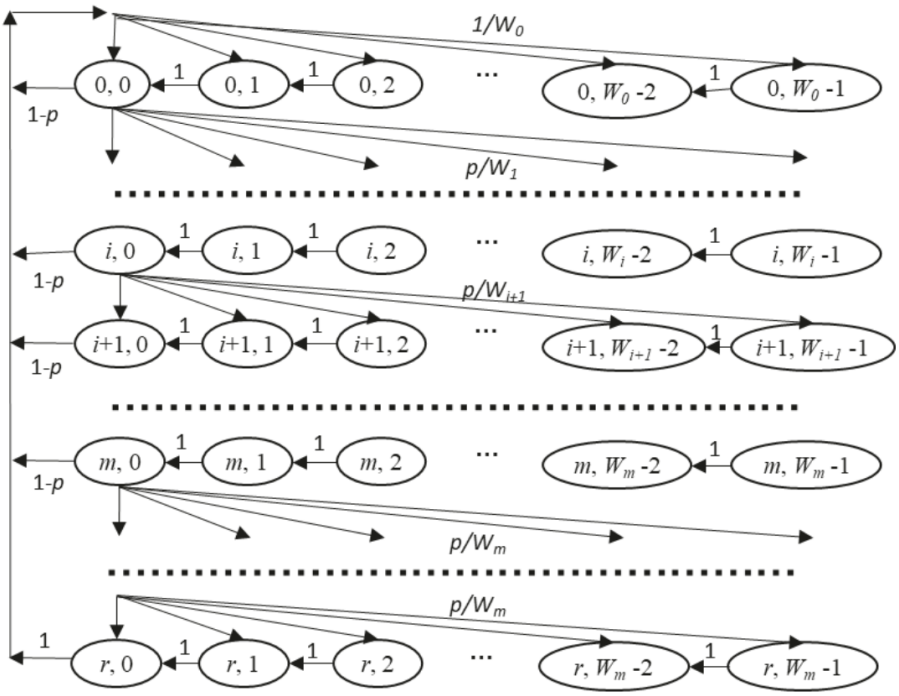

图 4.1　DCF 机制的 Markov 链模型

图中 p 为某个时隙传输失败的概率，Markov chain 一步状态转移概率为(4-2)所示：

$$\begin{cases} P\{i,k|i,k+1\}=1, & k\in[0,W_i-2], i\in[0,r] \\ P\{0,k|i,0\}=(1-p)/W_0 & k\in[0,W_0-1], i\in[0,r] \\ P\{0,k|i-1,0\}=p/W_i & k\in[0,W_i-1], i\in[1,r] \\ P\{0,k|r,0\}=1/W_0 & k\in[0,W_0-1] \end{cases} \tag{4-2}$$

式(4-2)中每个式子分别代表一定的物理含义。第一个等式代表，未达到重传上限时，退避计数器在每个空闲时隙的开始时刻减 1 的概率是 1。第二个等式代表，未达到重传上限时，当一个数据成功传输后，新到达的数据在 [0, W_0-1] 中等概率选一个随机数进行回退。第三个等式代表，未达到重传上限时，当一个数据第 i-1 次传输过程发生碰撞，节点进入第 i 阶回退过程，并在 [0, W_i-1] 中等概率选一个随机数进行回退。最后一个等式代表，当节点到达最大的传输次数以后，无论成功还是失败，CW 都会重置。

该 Markov chain 的任意状态之间可达，是不可约的。任意状态到另一状态的步长不存在周期, 具有非周期性。从任何状态出发，都能到达另一状态，具有常返性。因此该二进制退避过程的非周期不可约 Markov chain 具有稳态解，且所有稳态的概率之和为 1。

在传输模型建立至这一步时，我们发现，我们所建立的传输模型与经典的 Bianchi 传输模型有着极大的相似之处。这一事实更加佐证了我们模型的正确性。本文中我们的工作因此将基于这一基础的 Markov chain 上进行相关的工作。

对建立的 Markov chain 模型求稳态解，令 $b_{i,k} = \lim_{t\to\infty} P\{s(t) = i, b(t) = k\}, i \in (0, m), k \in (0, W_i - 1)$ 表示 Markov 链的稳态解，从图 4.1 中可以看出，对于一次发送失败的情况，状态 $b_{i-1,0}$ 到状态 $b_{i,0}$ 的步长包括：$b_{i-1,0} \to b_{i,0}, b_{i-1,0} \to b_{i,1} \to b_{i,0}, \cdots\cdots b_{i-1,0} \to b_{i,W_i-1} \to \cdots\cdots \to b_{i,1} \to b_{i,0}$，共 W_i 种，求和可得，$b_{i,0} = b_{i-1,0} * (p * 1/W_i + p * 1/W_i * 1 + \cdots + p * 1/W_i * 1^{W_i-1}) = p * b_{i-1,0}$, 即，

$$b_{i,0} = p * b_{i-1,0} \quad 0 < i \le r \quad \to \quad b_{i,0} = p^i * b_{0,0} \quad 0 < i \le r \tag{4-3}$$

同理，对于任一状态 $b_{i,k}$，若 $0 < i < r$，则是从一次发送失败的状态，通过竞争窗口加倍之后转移过来的。若 $i = 0$，则是从任一阶段发送成功，或达到重传次数限制后转移过来的。因此有，

$$b_{i,k} = \begin{cases} b_{i-1,0} * p * \frac{W_i - k}{W_i} & 0 < i < r \\ (1-p) * \frac{W_i - k}{W_i} * \sum_{j=0}^{r} b_{j,0} & i = 0 \end{cases} \tag{4-4}$$

将式 4-3 代入式 4-4，可得，

$$b_{i,k} = \frac{W_i - k}{W_i} * b_{i,0} \quad 0 \le i \le r, 0 \le k \le W_i - 1 \tag{4-5}$$

根据 Markov chain 的性质，所有稳态的概率之和为 1，因此有，

$$1 = \sum_{k=0}^{W_i-1} \sum_{i=0}^{r} b_{i,k} = \sum_{i=0}^{r} b_{i,0} \sum_{k=0}^{W_i-1} \frac{W_i - k}{W_i} = \sum_{i=0}^{r} b_{i,0} \frac{W_i + 1}{2} \tag{4-6}$$

根据式 4-1 和式 4-6，可以求得：

$$\mathrm{b}_{00} = \begin{cases} \frac{2(1-p)(1-2p)}{(1-2p)(1-p^{r+1}) + W_0(1-p)(1-(2p)^{r+1})} & r \le m \\ \frac{2(1-p)(1-2p)}{W_0(1-(2p)^{m+1})(1-p) + (1-2p)(1-p^{r+1}) + W_0 2^m p^{m+1}(1-p^{r-m})(1-2p)} & m < r \end{cases} \tag{4-7}$$

节点随机回退到 0 时发送数据，因此节点在一个时隙发送数据帧的概率为：

$$\tau = \sum_{0}^{r} b_{i,0} = b_{0,0} * \frac{1 - p^{r+1}}{1 - p} \tag{4-8}$$

传输数据发生冲突时，至少有另外一个节点也传播数据，共有 n 个节点，因此条件碰撞概率可表示为

$$p = 1 - (1 - \tau)^{n-1} \tag{4-9}$$

由此我们得到了式 4-8 和 4-9 两个含有未知待求参数 p 和 τ 的二元非线性方程组，联立即可求解。

4.3 基于节点 Markov chain 建模的信道吞吐计算

在上述的建模过程中，我们是基于多 BSS 系统（仅下行）中的单个节点进行分析的，下面将介绍如何利用单节点的建模计算出信道吞吐的过程。

我们将按照互听与不互听、并发传输是否成功进行分类讨论：

1. **互听、并发传输失败的时候**。对于多 BSS 系统，信道的状态可能会是①空闲，这时节点处于退避计数的状态下，正在退避计数，占用一个标准时隙长度 SLOTTIME；②传输成功（繁忙），这时多 BSS 中存在一个节点正在发送数据，其余的节点处于 hold 或者 DIFS 的监测时间中，所占用的时间即为数据成功传输时间和后续 ACK 相关时间；③传输失败（并发），这时由于传输失败所以 ACK 信号无法传回，发送端会等待 ACKTimeout，反应在信道上就是数据传输时间和后续 ACKTimeout 时间。值得注意的是当一个节点在 hold 时，另一个节点必定是在传输数据所以对信道来说不需要考虑 hold 相关问题。
2. **互听，并发传输成功的时候**。分析类似，但是在此时，并发传输成功，所以信道的状态有：①空闲，占用时间为一个 SLOTTIME; ②传输成功，占用数据成功传输时间和后续 ACK 相关时间。对于 hold 的情况处理同上。
3. **不互听，传输失败的时候**。此时信道状态为三种：①空闲，占用时间为一个标准时隙长度 SLOTTIME；②传输成功，占用数据成功传输时间和后续 ACK 相关时间；③传输失败，占用数据传输时间和后续 ACKTimeout 时间。由于不互听，所以节点永远监测到信道空闲，不存在 hold 的情况。
4. **不互听，传输成功的时候**。此时信道状态为三种：①空闲，占用时间为一个标准时隙长度 SLOTTIME；②传输成功，占用数据成功传输时间和后续 ACK 相关时间；

综上所述，信道上一个虚拟时隙的时间长度，可以以如下表示：

$$1\text{ 虚拟时隙时长} = \begin{cases} \sigma,\ \text{w.p. } 1-P_{tr} \\ T_s,\ \text{w.p. } P_{tr}P_s \\ T_c,\ \text{w.p. } P_{tr}\left(1-P_s\right) \end{cases}$$

其中 σ 代表协议中的标准时隙长度 SLOTTIME（在本模型中为 9μs），T_s 为成功传输所花费的总时间，T_c 为碰撞一次所花费的总时间。P_{tr} 为至少有一个节点准备发送的概率，P_s 为成功传输的概率。

对于系统的吞吐量，首先，吞吐是单位时间内发送数据有效载荷的比特数，单位为 bps。系统吞吐量 S 可以由信道的利用率与物理层速率（单位 bps）的乘积表示，

$$S = \frac{E[\text{一个时隙内传输的有效载荷发送时长}]}{E[\text{一个时隙长度}]} * \text{物理层速率} \tag{4-10}$$

信道处于三种虚拟时隙的概率可以由已经求解得到的 τ 和 p 表示，空闲时隙的长度 T_e

是 slotTime。成功传输和碰撞的传输时长 T_s 和 T_c 分别表示为

$$\begin{aligned} T_s &= H + E[P] + SIFS + ACK + DIFS \\ T_c &= H + E^*[P] + DIFS + ACKTimeout \\ E[P] &= E^*[P] \\ E[P] &= \frac{PAYLOAD}{SPEED} \\ MAC &= \frac{MAC_{length}}{SPEED} \end{aligned} \tag{4-11}$$

对于 4-10 中的不同参数所代表的含义的解释如下：H 为数据帧头，包括 MAC 层头和物理层 (PHY) 头，E[P] 为数据帧的有效载荷传输时长，$E^*[P]$ 为发生冲突时较长数据帧的有效载荷传输时长，本文中假设所有节点的数据帧长度一样，则 $E[P]$ 与 $E^*[P]$ 相等。PHY 头时长固定，MAC 头和有效载荷的发送时长由其字节长度除以物理层速率 $SPEED$ 得到。具体的，简单的帧序列如图 4.2 所示。

基于了成功传输和碰撞的传输时长 T_s 和 T_c 的表达式，为了求解归一化吞吐 S^*，我们首先需要计算一个时隙长度均值 E[一个时隙长度]，而在上述讨论中我们已经知道在一个虚拟时隙中，只可能有三种状态，为了求出均值，所以我们需要分别求出各种状态出现的概率，从而需要引入传输概率 P_{tr} 和成功传输概率 P_s。

对于上述两者的计算，则可以通过我们对节点建立的 Markov chain 求解，至此我们就建立出来单节点的 Markov chain 和信道状态的关系。

所以，对于传输概率 P_{tr}，其物理意义是至少有一个节点，在这个时隙内准备传输。对于基本的传输模型，它的表达式为

$$P_{tr} = 1 - (1 - \tau)^n \tag{4-12}$$

该式子中的 n 是指总的竞争节点数目，由于我们的模型假设中是单一信道网络，并且始终计算饱和吞吐量，所以可以认为所有的节点都在竞争信道。

对于传输成功概率 P_s，可以将其视作一个条件概率，物理意义是在有节点发送的前提下，能成功传输的概率。在不添加任何其他条件的基本模型中，它的表达式可以写作：

$$P_s = \frac{n\tau(1-\tau)^{n-1}}{P_{tr}} \tag{4-13}$$

由新引入的两个量，我们可以将上文中提到的归一化吞吐具体地表达出来，也即

$$S^* = \frac{P_s P_{tr} E[P]}{(1 - P_{tr})\,\sigma + P_s P_{tr} T_s + P_{tr}\,(1 - P_s)\,T_c} \tag{4-14}$$

由此我们就得到了完整的基本传输模型，并可以对其中的重要参数求解，最终得到系统的吞吐量。**注意：对于实际系统中的吞吐量，不是简单的归一化吞吐与物理层速率的简单乘积，在后续的问题中应具体分析**。

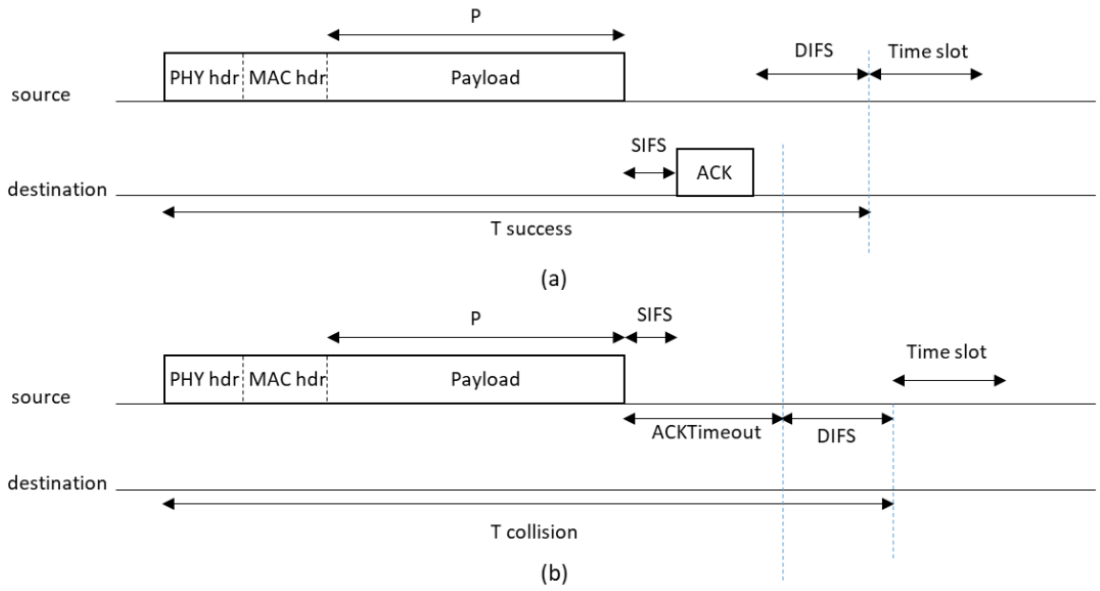

图 4.2　帧序列：(a) 成功发送　(b) 冲突

5. 仿真算法

5.1 仿真思路分析

对于本文基本机制 CSMA/CA 机制的仿真，我们选择采用基于离散事件的仿真。在这种仿真方法中，系统时间被划分为小的离散时间步进（本文以 us 为单位）。在一个指定最大仿真时间的时间轴上，每进行一个离散时间，我们会对当前网络中事件的发生进行监控。而对于本题中的事件，即网络信道状态的转变和每个节点状态的转变，我们使用有限状态机的 (Finite State Machine) 思想进行仿真，在每一个时间点对信道状态、节点状态进行检测。

5.2 仿真模型建立

5.2.1 信道模型建立

对于信道 (channel)，我们使用 3 种状态来表示信道状态：

1. **IDLE_STATE.** 指信道中没有信息进行传递，即此时信道中的节点都处于信道可用评估或随机回退阶段。
2. **TX_STATE.** 指信道中有一个节点进行信息传递。
3. **COLLISION_STATE.** 指信道中有两个及以上节点进行信息传递。此时信道处于碰撞状态。

5.2.2 节点模型建立

而对于节点 (node)，我们采用如下 6 种状态来表示每个节点可能的状态：

1. **IDLE_STATE.** 节点处于信道可用评估阶段，需等待一个信道始终空闲的 DIFS 时长。
2. **BACKOFF_STATE.** 节点处于回退阶段，需等待数个信道始终空闲的时隙长度 SLOT-TIME。
3. **SEND_DATA_STATE.** 节点度过了回退阶段，会发送一个数据帧，等待数据帧发送时长。这个时长中会对信道进行监控，根据是否有多个节点发送数据帧转化信道状态。同时根据信道状况与实际 SIR 大小的情况判断传输结果。如果判断节点发送成功，会先进入 UNKNOWN_STATE，否则进入表示发送失败的 WAIT_ACKTIMEOUT_STATE.
4. **UNKNOWN_STATE.** 节点即使在信道成功传输数据，也会根据丢包率在此状态进行判断。如果不丢包，才会真正进入表示成功传输的 WAIT_ACK_STATE，否则同样进入 WAIT_ACKTIMEOUT_STATE.
5. **WAIT_ACK_STATE.** 表示节点成功传输数据帧，会等待 SIFS+ACK 的时长。
6. **WAIT_ACKTIMEOUT_STATE.** 表示节点失败传输数据帧，会等待 ACKTIMEOUT 的时长。

此外，我们发现上述状态中除了 UNKNOWN_STATE 状态作为过渡状态，其余状态均有等待时长。只有当每个状态的等待时长结束，才会进入下一个状态，并且由于信道状态可能转为 TX_STATE 或 COLLISION_STATE 的忙碌状态，让节点进入等待，因此我们使用一个计数器 (timer) 对等待时长进行倒计时，并且在可能的等待状态暂停倒计时。

计时器也会进行状态转换，我们采用如下 3 种状态来表示计数器点可能的状态。

1. **COUNTING_STATE.** 计数器处于倒计时状态，根据目前的等待时长进行倒计时。
2. **WAIT_STATE.** 当信道处于忙碌状态时，如果此时节点未处于发送状态，需要暂停等待信道空闲。
3. **READY_STATE.** 倒计时结束，只有此时节点才会转换为下一状态。

对于每个节点状态机的转化，我么可以使用 Fig. 5.3表示:

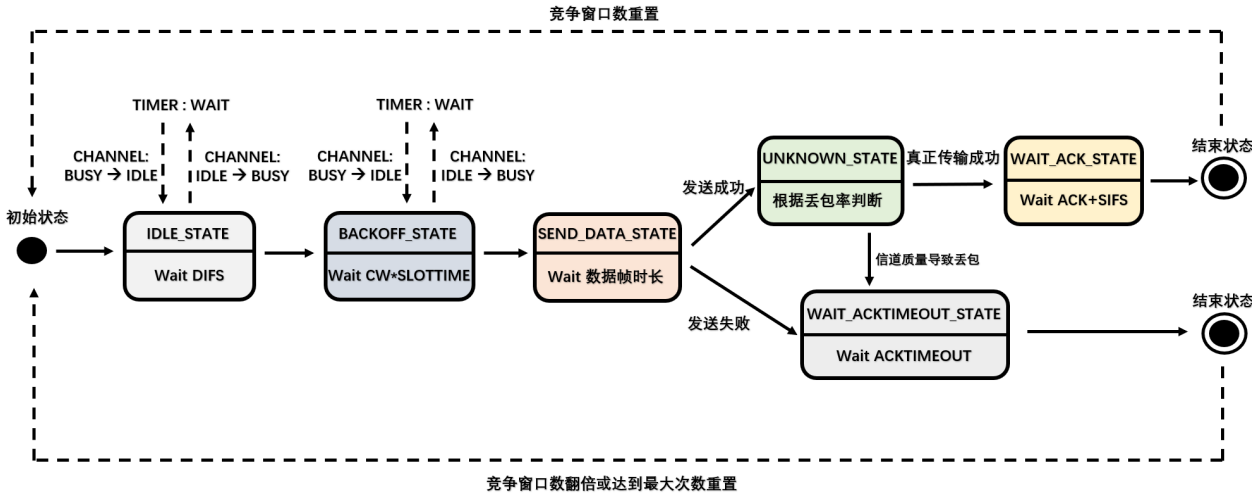

图 5.3　节点的状态机转化示意图

5.3 仿真主算法简述

在对信道模型以及节点模型进行建立之后，我们可以将节点封装为一个类，只需要通过传递每个时刻的信道状态，每个节点便会在内部进行状态的转化。我们可以运用离散事件的思想，通过设置一个最大仿真时间，在每一个时刻对每个节点的状态进行更新，再通过节点此时状态对信道状态进行更新。在每个循环最后，我们会统计已经成功传输的节点，将有效载荷发送时长进行统计。

运行结束，我们可以通过如下公式得到吞吐 S:

$$\boldsymbol{S = \frac{TX\_total\_time}{sim\_time} * phy\_rate} \tag{5-1}$$

在上式中，$\boldsymbol{TX\_total\_time}$ 表示总共的有效载荷发送时长，$\boldsymbol{sim\_time}$ 表示总共仿真时间，而 $\boldsymbol{phy\_rate}$ 表示物理层速率。

上述仿真算法具有如下简便性与通用性：我们通过将每一个节点封装为类，每一个时刻只需要传入当前信道状态进行更新，极大地简化了主算法，**并且对于多节点下的互听/不互听，以及并发时能否成功传输两种状况，能用极简单的代码进行处理：**

互听/不互听：当节点与其余节点存在不互听状态时，我们可以认为这个节点接受的信道状态是不完全的，只接受到了可以彼此互听节点的状态，可以认为这个节点只接受到了一个 $sub_channel_state$。如题目 2 中，只存在两个节点，彼此不互听的情况下，可以认为两个节点一致认为信道”空闲“，因此在循环中只需要一直传入信道状态为空闲即可; 在题目四中，AP1 与 AP3 不互听，AP2 与两者都互听，可以认为 AP2 听到的的是整个信道的状态，而 AP1、AP3 由于只能听到 AP2，因此接收到的信道状态是由 AP2 节点状态决定的，等同于个这个 $sub_channel_state$ 是由 AP2 决定的：当 AP2 空闲，$sub_channel_state$ 是空闲状态；当 AP2 在传输状态，$sub_channel_state$ 是忙碌状态。因此只需要在主程序对传入节点信道状态进行改变，便可解决互听/不互听问题。

并发时能否成功传输：当并发状态存在时，会根据系统的 SIR，决定此时并发状态下能否成功传输。对于代码也只需要做出极小的修改：我们在主程序里在碰撞状态下，为碰撞节点传入不同的信号，若 SIR 较高，碰撞的节点都视为成功传输；若 SIR 较低，碰撞的节点都视为失败传输。

综上，上述仿真代码设置具有简便性与通用性。通过将节点的状态转移进行封装，我们只需要在主程序中对不同情况做出细微改变，便可实现仿真模型的建立。

5.4 仿真代码

我们使用 Python 语言对仿真程序进行编写，具体代码可见附录 F，以及代码附件。

6. 模型建立与求解

6.1 问题一

6.1.1 模型建立

在该问题中，我们的研究对象为一个多 BSS 同频系统，其中包含了 2 个 AP 与 2 个 STA，在题目中 AP 之间的 RSSI 为-70dBm，大于 CAA 阈值-84dBm，那么二者是互听的，即可以知道信道当前状态，两个 AP 分别向自己对应的 STA 进行数据传输，在传输过程中由于是同频，那么他们共享一个信道，分析该信道可能存在的状态：

- 第一，空闲，即无 AP 向 STA 传输；
- 第二，传输成功，只有其中一个 AP 向 STA 发送信息，另一个 AP 监测到信道繁忙，不进行发送；
- 第三，并发，当信道处于空闲时，两个 AP 经过 DIFS 后，进入随机退避阶段，那么二者就存在并发的情况，这个时候两个 AP 同时向对应的 STA 发出信号（经过调制），经过传输之后，STA 接收到信号，然后进行解调，但是由于题目中的条件给出这个时候 SIR 较低，即信号和其干扰（对 STA1 来说 AP2 发出的信号为干扰，反之）在解调器中无法区分开，那么无法解调成功，所以造成信号传输失败，接收端 STA 无法向 AP 发出 ACK 信号，这个时候即碰撞，退避计数器将按照二进制指数退避算法升阶，再此进行随机回避。

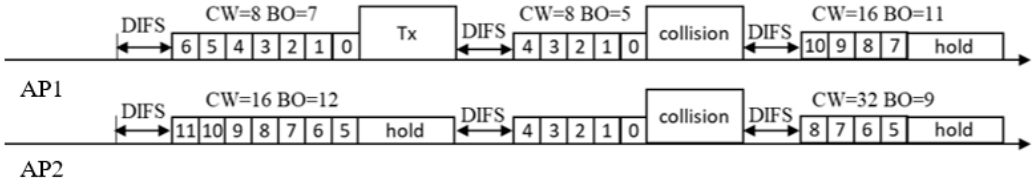

图 6.1　当退避器同时达到 0 时形成并发

此处，在该多 BSS 系统中，两个 AP 节点是对称的。即不论分析哪个节点，二者都处于同样的状况下，即与另一个互听，同频传输（在同一个信道上传输数据），且各种参数相同，所以二者对称，在后续分析中任意选一个节点进行分析即可。

综上所述，信道存在三个状态：空闲、传输成功、传输失败（并发、碰撞），所以，据此我们建立类似于 Bianchi 模型的二维 Markov chain，将上述的每个状态都看作一个虚拟时隙，那么信道在三种虚拟时隙中转化。建模时，我们对于其中一个节点 AP，将退避器所处的阶数和随机回退数 $\{b(t), s(t)\}$ 作为 Markov chain 中的状态，与上面叙述基本模型一致，**需要特别注意的是，在此处的 Markov chain 退避器升阶的概率 p，即是条件并发传输概率（对应 Bianchi 模型中的条件碰撞概率 p），且该节点发送概率依然为 τ**。

6.1.2 模型求解

依据上述基本模型章节的推导，该二维 Markov chain 具有稳态解，所以在稳态时，有以下 Markov chain 稳态解 b_{00}，发送概率 τ，条件传输失败（碰撞）概率 p:

$$\mathrm{b}_{00}=\begin{cases}\frac{2(1-p)(1-2p)}{(1-2p)(1-p^{r+1})+W_0(1-p)(1-(2p)^{r+1})} & r\leq m\\ \frac{2(1-p)(1-2p)}{W_0(1-(2p)^{m+1})(1-p)+(1-2p)(1-p^{r+1})+W_0 2^m p^{m+1}(1-p^{r-m})(1-2p)} & m<r\end{cases} \tag{6-1}$$

$$\tau=\sum_{0}^{r}b_{i,0}=b_{0,0}*\frac{1-p^{r+1}}{1-p} \tag{6-2}$$

$$p=1-(1-\tau)^{n-1} \tag{6-3}$$

在本问题中退避器最高阶数为 $m=log_2\frac{1024}{16}=6$，最大重传次数 $r=32$，$r>m$, 那么取

$$b_{00}=\frac{2(1-p)(1-2p)}{W_0\left(1-(2p)^{m+1}\right)(1-p)+(1-2p)\left(1-p^{r+1}\right)+W_0 2^m p^{m+1}\left(1-p^{r-m}\right)(1-2p)} \tag{6-4}$$

代入 τ，至此式 (6-2) 与 (6-3) 构成非线性二元方程。我们可以在 Matlab 中利用 fsolve 函数，将题目中的参数 $W_0=16$ 和 $n=2$ 代入，求解该非线性二元方程。

最终得到结果，$\tau=p=0.1046$，从而根据上述基本模型的分析，得到以下:

传输概率，至少有一个节点在发送的概率:

$$P_{tr}=1-(1-\tau)^2=0.1983 \tag{6-5}$$

条件传输成功概率，有节点发送时，由于并发会失败，所以是有且仅有一个节点在发送的概率:

$$P_s=\frac{2\tau(1-\tau)^{2-1}}{P_{tr}}=0.9448 \tag{6-6}$$

在这之后，代入题目中的通用参数 ACK、SIFS 等，以及问题中的载荷长度为 1500Bytes（1Bytes = 8bits），PHY 头时长为 13.6µs，MAC 头为 30Bytes，MAC 头和有效载荷采用物理层速率 455.8Mbps。从而求解出:

$$T_s=H+E[P]+SIFS+ACK+DIFS=131.4539\mu s \tag{6-7}$$

$$T_c=H+E^*[P]+DIFS+ACKTimeout=148.4539\mu s \tag{6-8}$$

所以在此问题中建立的数学模型中，归一化吞吐 S^*:

$$S^*=\frac{P_s P_{tr} E[P]}{(1-P_{tr})\,\sigma+P_s P_{tr} T_s+P_{tr}\,(1-P_s)\,T_c}=0.14738 \tag{6-9}$$

在本问题中，由于在信道上传输数据的始终只有一个节点，不存在多节点同时传输的情况，故吞吐为：$S = S^* \times 455.8 = 67.17442$Mbps

6.1.3 仿真验证

在仿真算法中，我们设置节点个数为 $num_nodes = 2$,$sim_time = 10000000$(一千万微秒)。由于系统状态为 AP1 与 AP2 互听，AP1 和 AP2 可以监听整个信道的状态，因此在节点状态更新时传递整个信道的状态；并且并发时的 SIR 较低会导致传输失败，我们设置传输失败时节点会进入 WAIT_ACKTIMEOUT_STATE。通过运行如上仿真代码，我们可以得到归一化吞吐 S^* 与最终吞吐 S 大小：

$$S^* = 0.14298, S = 65.1702$$

我们可以看到仿真算法计算结果与数学模型计算结果吻合的极好，**归一化吞吐相对误差仅为** 3.075%, 为了进一步验证数学模型的准确性，我们在此题参数设置下，改变发送包的载荷长度，从 1500Bytes，变为 [100,200,300,...,1400,1500]Bytes, 以 100 为间隔，共 15 组参数。两者运行结果如图 6.3 所示。

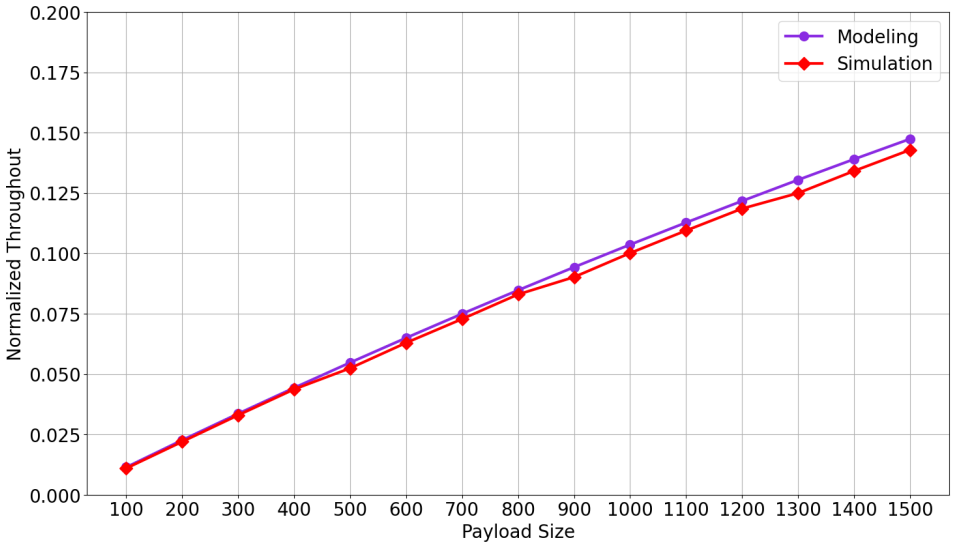

图 6.2　问题一不同载荷长度下仿真与数学模型结果对比

在不同载荷长度设置下，**归一化吞吐的平均相对误差仅为** 3.100%, 进一步说明数学模型与仿真结果极好吻合。

6.2 问题二

6.2.1 模型建立

在问题二中，类似于上述问题一中的分析，两个 AP 同样会出现退避器同时到达 0 的状态，此时他们同时向自己的 STA 发送信号，这个时候由于并发传输时两个终端 STA 接收到的信号 SIR 较高，终端 STA 能够从并发的信号中解调出自己的信号，在接收完成后向各自的 AP 发送 ACK 信号，所以两个 AP 的数据传输都能成功。从而此时信道无论是否处于并发状态，都可以视作传输成功状态，也即在问题二中信道可能存在的状态只有空闲与传输成功两个。

综上所述，信道中的状态存在空闲和传输成功两个状态，值得注意的是，此处的传输成功包含两种情况：1. 只有一个 AP 向其对应的 STA 发送信号，2. 两个 AP 出现并发状况，同时向对应的 STA 发送信号。那么我们可以做出结论，在问题二的情况下，是不会发生退避器升阶的情况的，即不会存在因为碰撞导致信号传输失败，无法收到 ACK，从而进入下一次的随机退避阶段。从而我们在问题一建立的二维 Markov chain 转移状态 $\{b(t), s(t)\}$ 中的退避器阶数 $s(t)$ 固定为 0，即竞争窗口 CW 一直维持初始大小，从而二维 Markov chain 退化为一维 Markov chain。

所以，我们将上述的两个状态都看作一个虚拟时隙，那么信号将在两种虚拟时隙中转化，对于其中一个节点 AP，只将随机回退数 $b(t)$ 作为一维 Markov chain 中的状态，建立以下模型。

我们建立一维 Markov chain，令 $b(t)$ 代表 t 时刻一个节点退避随机过程的退避计数，此处的 t 是一个离散的虚拟时隙的开始时刻。CW 保持初始大小不变：

$$CW = W_0 \tag{6-10}$$

一维 $\{b(t)\}$ 随机过程可以用一维 Markov chain 表示，如图 6.2 所示。

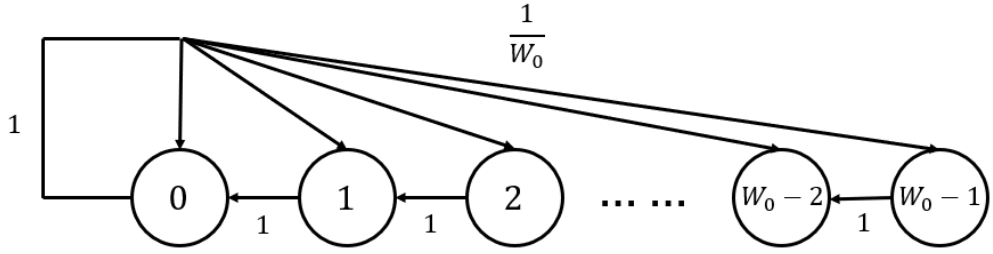

图 6.3　问题二的一维 Markov chain 模型

此处一维的 Markov chain 一步状态转移概率为：

$$P(k|k+1) = 1 \quad k \in [0, W_0 - 2]$$

$$P(k|0) = 1/W_0 \quad k \in [0, W_0 - 1]$$

上述等式都具有一定物理意义，其中第一个等式代表，退避计数器在每个空闲时隙的开始时刻减一的概率是 1；第二个等式代表，退避计数器退避至 0 时不论是否发送并发传输，都会传输成功，新到达的数据在 $[0, W_0 - 1]$ 中等概率选一个随机数进行回退。

6.2.2 模型求解

分析该一维的 Markov chain，仍然可以发现，其依旧具备不可约、非周期、常返的性质，新的 Markov chain 也有稳态解。对新 Markov chain 模型求稳态解。根据 Markov chain 的性质，所有稳态的概率之和为 1。不妨令 $b_k = \lim_{t\to\infty} P\{b(t) = k\}, k \in [0, W_0 - 1]$ 表示 Markov chain 的稳态解，从图 6.2 中可以看出，在退避器回退至 0 时，不论怎么样都发送成功，然后进行下一次退避。以 b_1 为例，所有状态 b_0 到 b_1 的步长包括：$b_0 \to b_1, b_0 \to b_2 \to b_1, b_0 \to b_3 \to b_2 \to b_1, \cdots\cdots, b_0 \to b_{W_0-1} \to \cdots \to b_1$，共 $W_0 - 1$ 种，对于 $b_2, b_3, \cdots, b_{W_0-1}$ 同理，所以求和可得

$$b_k = \frac{W_0 - k}{W_0} b_0 \tag{6-11}$$

根据 Markov chain 的性质，所有稳态的概率之和为 1，因此有，

$$\sum_{k=0}^{W_0-1} b_k = \sum_{k=0}^{W_0-1} \frac{W_0 - k}{W_0} b_0 = \frac{W_0(W_0 + 1)}{2} b_0 = 1 \tag{6-12}$$

从而有

$$b_0 = \frac{2}{W_0 + 1} \tag{6-13}$$

节点随机回退到 0 时发送数据，因此节点在一个时隙发送数据帧的概率为

$$\tau = b_0 = \frac{2}{W_0 + 1} \tag{6-14}$$

同时，经上述分析不存在退避器升阶的情况，故碰撞（并发）概率

$$p = 0 \tag{6-15}$$

下面进行吞吐的计算，对于该题，值得注意的是，当并发存在时，该题条件下传输成功，那么可以认为此时单位传输的比特数是非并发情况的 2 倍，所以在计算实际吞吐量的时候应该分开计算。在传输的时候，尽管是并发传输，但是其时间是相同的，所以对于归一化吞吐，其计算仍然按照先前的算法。

所以，在此情况下计算归一化吞吐，传输概率为至少有一个节点在发送的概率，即：

$$P_{tr} = 1 - (1 - \tau)^2 = 0.2215 \tag{6-16}$$

条件传输成功概率，有节点发送时，由于并发能成功，所以传输必定成功：

$$P_s = 1 \tag{6-17}$$

同时，由于不论是如何发送的情况，发送都不会失败，所以不会存在接受不到 ACK，导致 ACKTimeout 的出现，所以在此处 $T_c = T_s$ 从而归一化吞吐为：

$$S^* = \frac{P_s P_{tr} E[P]}{(1 - P_{tr})\,\sigma + P_{tr} T_s} = 0.24122 \tag{6-18}$$

在求实际吞吐时，并发时传输成功，单位时间内传输的比特数是不并发时的两倍，故而此时的物理速率可以看作为单发时的 2 倍，同理我们将 P_{tr} 看作两部分，分别由只有一个节点发送概率 P_{tr1}（两个节点中选一个节点发送）和两个节点并发概率 P_{tr2}（两个节点同时发送）构成，此时有：

$$\begin{aligned} P_{tr1} &= C_2^1 \tau (1 - \tau) \\ P_{tr2} &= \tau^2 \\ P_{tr} &= P_{tr1} + P_{tr2} \end{aligned} \tag{6-19}$$

也就是说，在信道有效信息传输的时间中，一部分时间时单发传输，一部分是并发传输，而总的虚拟时隙时间不会因为是否并发而变化，那么实际的信道吞吐量为：

$$\begin{aligned} S =& \frac{P_s P_{tr1} E[P]}{(1 - P_{tr})\,\sigma + P_{tr} T_s} \times SPEED + \\ & \frac{P_s P_{tr2} E[P]}{(1 - P_{tr})\,\sigma + P_{tr} T_s} \times 2SPEED \\ =& 70.558 Mbps \end{aligned} \tag{6-20}$$

6.2.3 仿真验证

在仿真算法中，我们设置节点个数为 $num_nodes = 2$,$sim_time = 10000000$(一千万微秒)。与题目一不同之处并发时的 SIR 较低会导致传输失败，我们设置传输失败时节点虽然失败，系统会进入碰撞状态，但是节点仍然会进入 WAIT_ACK_STATE，这段时间也会被统计算入成功传输时间。通过运行如上仿真代码，我们可以得到归一化吞吐 S^*：

$$S^* = 0.23326$$

在此处，由于在建模阶段考虑了并发时传输量为双倍，而在仿真中，我们只考虑了信道上的有效数据传输时间，而并发时的传输时间不会变化，所以此处利用求解仿真的实际传输吞吐无意义，不作讨论。以建模求解的系统吞吐为准。

我们可以看到仿真算法计算结果与数学模型计算结果吻合的极好，**归一化吞吐相对误差仅为** 3.299%。我们同样改变发送包的载荷长度，变为 [100,200,300,...,1400,1500]Bytes, 以 100 为间隔，共 15 组参数。两者运行结果如图 6.4 所示。

在不同载荷长度设置下，**归一化吞吐的平均相对误差仅为** 3.487%, 进一步说明数学模型与仿真结果极好吻合。

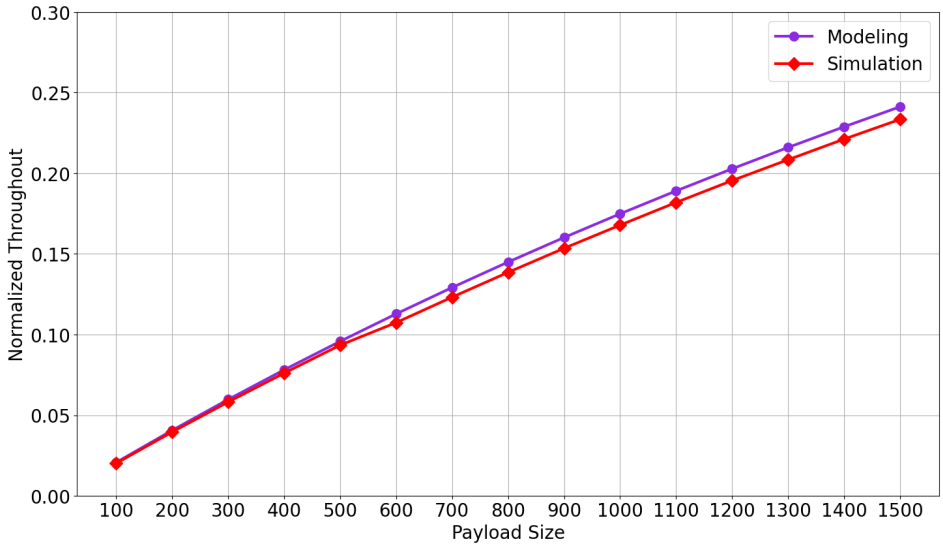

图 6.4　问题二不同载荷长度下仿真与数学模型结果对比

6.3 问题三

6.3.1 模型建立

在问题三中，两个 AP 之间的 RSSI 被设定为-90dBm，小于 CAA 阈值-84dBm，这意味着两个 AP 之间不互听，那么 AP 会一直认为信道为空闲状态，其会不断的尝试发送信号，这就意味着，此时和互听状况不同，当一个 AP 向其对应的 STA 传输数据的时候，另一个 AP 也可能向它对应的 STA 发送信号，那么就会存在交叠的状况。另外，在基本模型中，我们假设了理想信道，不会因为信道质量而丢包，在问题三的要求中，我们需要考虑实际的传输情况下，当有遮挡物或者人走动时，无线信道都可能会快速发生比较大的变化。当仅有一个 AP 发送数据时，即便不存在邻 BSS 干扰，也会有 10% 以内不同程度的丢包，丢包时我们认为此时可视作一次需要重传的失败传输而不是直接丢弃数据帧，从而重新回到升阶后的随机回避阶段。该丢包的含义为数据包在发送的过程中，STA 由于自身接受机的异常导致包的解调失败，也就是说如果数据包丢失，该包也在信道中存在，从而可以得出结论：丢包过程是在接收端发生的，不影响传输过程，如果传输过程中发生了交叠，那么在接收端依然是交叠的情况。在该问题中，两个 AP 间 SIR 较小，一旦**两个 AP 发包在时间上有交叠**时，由于 SIR 比较小，会导致两个 AP 的发包均失败。

交叠过程分析

对于该问题，我们依然沿用前面的研究方法，首先分析该条件下的信道状态，第三问最大的区别就是，当两个 AP 不互听的时候，AP 总是认为信道空闲，从而不断尝试发送信息，导致所有 AP 在传输数据的时候，不进行 hold，从而导致信号交叠。在此问题中，来

自 AP1 和 AP2 的电磁波信号混叠，当混叠信号到达 STA 处时，由于 SIR 较低，解码失败，导致传输错误。考虑到混叠的情况，我们在此进行分析。对于特定的一个 STA1，其对应 AP1 向其发送的视为信号，AP2 发送给 STA2，视为干扰。那么在时间上可能存在如下四种情况。

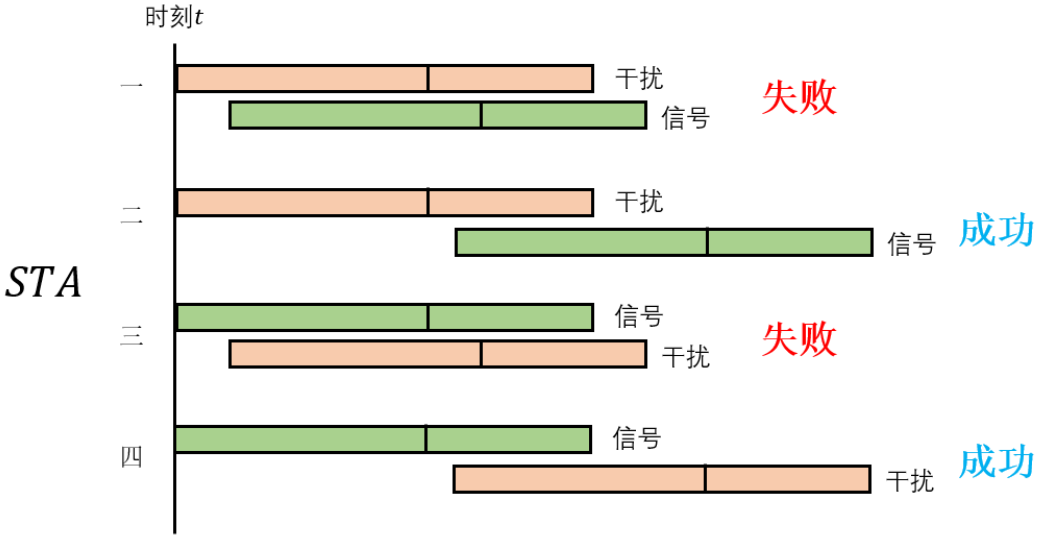

图 6.5　信号传输情况示意图

由题意，默认两个 AP 发包在时间上有交叠时，假设 SIR 比较小，会导致两个 AP 的发包均失败。当信号包先到时，接收机锁定 Preamble，干扰包被视为干扰，此时 SIR 较低，失败；当干扰包先到时，接收机先锁定干扰包的 Preamble，导致错过信号包的 Preamble，则一定接收失败。同时又有小信号屏蔽算法，接收机在信号包到达时转为锁定 RSSI 更大的信号包，此时同样会由于 SIR 较低，导致接受失败。

所以对于**情况**一，干扰包先到，先锁定干扰包 Preamble，随后信号包到达，由小信号屏蔽算法锁定信号 Preamble，但是由于 SIR 较低，失败。对于**情况**二，STA 先接受完干扰的数据包（图中每一条的竖线之前代表数据包：$PHY + MAC + PAYLOAD$），发现不是给自己的数据，然后在后面接收到来自自己 AP 的数据包，成功。对于**情况**三，先接收到信号数据包，但是在数据包传输阶段，接收到了干扰信号，同样由于 SIR 较低，失败。对于**情况四**，数据包接受完成后，才传来干扰，所以成功。

综上所述，当且仅当信号和干扰发生交叠的时候，会导致传输失败。

二维 Markov chain 建模

所以，我们可以分析出信道可能所处的状态为：空闲，传输成功（不交叠），传输失败（交叠）。类似的，我们将信道的每个状态看作一个虚拟时隙，信道在三种虚拟时隙中转化，由于节点的对称性，我们将其中一个节点的退避器所处阶数和随机回退数用二维 Markov chain 表示。我们在这里构建出和基本模型结构一致，但参数含义略有不同，将在后续进行

分析。

6.3.2 模型求解

p 的重构

在基本模型中，二维 Markov chain 描述了退避器的阶数和随机回退数，其中退避器的阶数以概率 p 转移至下一阶，即因为传输失败导致重新进入升阶的退避器。而在问题三中，由于信道可能存在的缺陷，以及不互听导致的交叠等多个引起传输失败的条件相互交织，我们需要考察新情况下 p 的含义及其表达式。

对于问题三，p 的含义依旧为传输失败，退避器升阶的概率，但是由两部分组成：此处考察信道状况和信号交叠之间的影响。对于本题的两个节点 1 和 2 来讲，节点 1 与节点 2 信号交叠情况和节点 2 信道干扰无关，反之亦然。

$$p = P_e + (1 - P_e)(1 - p_h) \tag{6-21}$$

其中 P_e 为信道丢包概率，p_h 为信号不发生交叠的概率，则上述表达式是指，对于所研究的节点来讲，其对应的 AP 发送的信号传输失败只有两种情况，第一，信道丢包时，无论是否在 STA 发生交叠，传输一定失败，即第一项；第二，信道不丢包的情况下，还要考虑在 STA 交叠发生的概率，即第二项。

不交叠概率 p_h 求解

在问题三的叙述中，重点强调了交叠的定义，我们可以理解为如图 5.4 所示，即交叠的区间为 $PHY + MAC + PAYLOAD$。

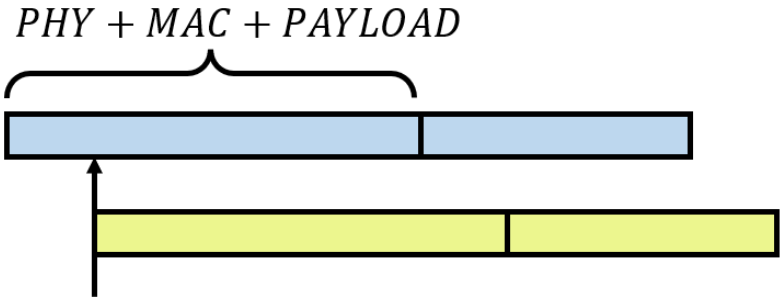

图 6.6　信号交叠示意图

仿照文献 [4] 对隐藏节点的研究方法，由于系统是在传输的时刻发生交叠，所以选择节点 1 作为研究对象，并且二维 Markov chain 中的单位时间为 1 个 SLOTTIME，在一个单位时间中从一个状态转移到另一个状态，那么计算在整个数据传输阶段，所需要的单位时间数 k：

$$k = \frac{PHY + MAC + PAYLOAD}{\sigma} \tag{6-22}$$

令 l_p 表示满足 $2^{l_p}W_0 - 1 \geq k$ 的最小退避阶数。在本文假设下，节点 1、2 不互听，所以总是在不断的试图传输数据。下面分析出现交叠的情况，即当研究对象节点 1 退避计数归零时，节点 2 处于退避状态，并且在数据传输阶段的时间中的某一时刻，节点 2 退避至 0，节点 2 也发出数据，那么这个时候就会出现交叠的情况，从而导致传输失败。这种分析是对称的，对节点 2 亦然，我们只需要研究一种即可。

因此，节点 1 退避至 0 时，通过分析节点 2 处于的退避状态即可求出不交叠的概率。如果节点 2 的退避值小于等于上述计算出的 k，那么在传输阶段过程中，节点 2 就会出现退避至 0 的情况，导致交叠。但是如果此时节点 2 的退避值大于 k，那么在数据传输结束之前，都不会出现节点 2 退避至 0，即不会交叠。

综上所述，不交叠的概率即是节点 2 的退避值大于 k 的概率：

$$p_h = \sum_{i=l_p}^{r} \sum_{j=k+1}^{W_i-1} b_{i,j} \tag{6-23}$$

P_s 的重构

在基本模型中，P_s 代表对于整个信道来说，在有节点发送的条件下，成功传输的概率。由于本模型假设为单一信道，我们可以分开考虑不同节点发送的情况，从而将 P_s 以如下形式表达：

$$P_s = \frac{\tau_1[(1-\tau_2)+\tau_2]p_h(1-P_e) + \tau_2[(1-\tau_1)+\tau_1]p_h(1-P_e)}{P_{tr}} \tag{6-24}$$

现在对上式做出说明，我们假设两个 AP 各自的发送概率分别为 τ_1 和 τ_2，在 AP1 发送的情况下，要保证传输成功，既要求 AP1 传输包不受信道干扰，又要求 AP1 信息不与 AP2 信息发生交叠，由于 P_s 代表系统处于一个时刻时的相关概率，在这个时刻瞬间，AP2 是否发送不影响具体状况，故在公式中我们将其写作 $[(1-\tau_2)+\tau_2]$。对于 AP2 发送的情况，与 AP1 发送的情况是对称的，由此我们就构成了完整的 P_s 的新的表达式。

在本问题中，两个 AP 各自的发送概率是相等的，即 $\tau_1 = \tau_2$，因此，式 5-20 可以简化为如下形式：

$$P_s = \frac{C_2^1 \tau p_h(1-P_e)}{P_{tr}} \tag{6-25}$$

式 5-21 即最终的本问题中的 P_s 表达式。

S 的重构

在问题一和问题二中，都是互听的，从信道时序上来看，信道在传输成功时是以 T_s 的稳定间隔运行的，如图 6.5 中 (a) 所示，也就是说在互听的情况下，信道状态中传输成功是离散的进行转移。但是我们发现，在第三问中，由于引入了交叠现象，信道在传输成功时，其运行的时间长度将不再是一个常数，如图 6.5 中 (b) 所示，在传输成功的情况中，当传输两个包成功时，存在有传输时间中的 SIFS 和 ACK 段重叠的可能性（我们在此称之为无关

交叠），使得此状态下的时隙长度小于 $2T_s$。从而导致在原有的归一化吞吐公式计算中，可能将信道的一个时隙长度高估。

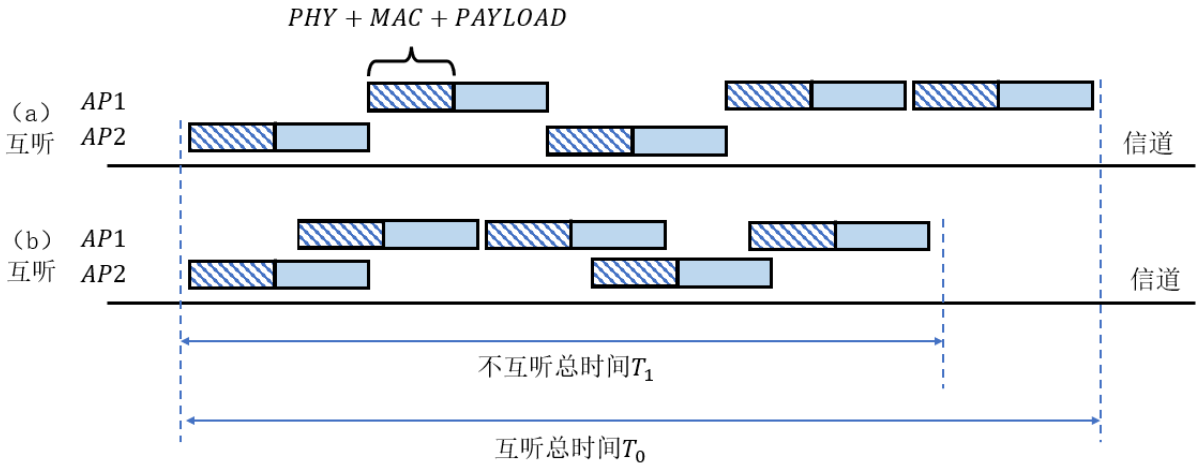

图 6.7　不互听时长变化示意图

因此，需要重构 S 的表达式，以适应不同情况，我们的思路是，对于式 4-10 中展示的 S^* 表达式。

其中 T_s 可能有两种情况，第一，存在无关交叠；第二，类似于互听情况下的依次传输。那么分别计算这两种情况出现的概率。情况一，无关交叠是指如下图所示的时间段内发生的交叠，所以仿照前文求解 p_h 的方法，我们只需要计算当一个节点退避至 0 时，另一个节点的退避计数位于此时间段内的概率 p_u，即退避计数器值处于 $[k, k_2]$，值得一提的是，此概率包含的事件为 p_h 的子集。

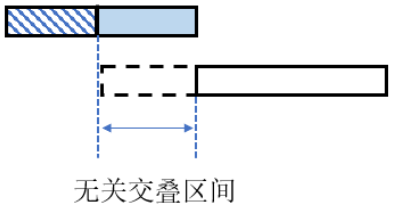

图 6.8　无关交叠时间段

.

从而有：

$$k_2 = \frac{T_s}{\delta}$$
$$p_u = \sum_{i=l_p}^{r} \sum_{j=k+1}^{k_2} b_{i,j} \tag{6-26}$$

无关交叠对应的时长，在此处是难以计算的，因为无法得知无关交叠的程度具体为多少，所以我们在这里使用无关交叠的两个极端情况的均值进行近似，即：

$$\begin{aligned} T_{s_u} =& \frac{1}{2}[(PHY + MAC + PAYLOAD + T_s) + 2T_s] \\ =& \frac{PHY + MAC + DATA}{2} + \frac{3}{2}T_s \end{aligned} \tag{6-27}$$

对于情况二，互听情况下依次传输的概率，即为 $p_h - p_u$，此时的对应时间依然是 T_S。

对于 $E[P]$，可以从图中看出当发生如图 6.5(b) 情况时，载荷的有效传输时长变为原来的两倍。在其他情况下则不变，依据这一条件分子的表达式也可以得到。

$$E[P]' = \frac{p_h * 2EP}{p_h} + \frac{(p_h - p_u) * EP}{p_h} \tag{6-28}$$

我们定义当一个时隙中成功传输一个数据包时，产生的有效载荷时长为 EP，在问题三中，正如图 6.5 所示，发生无关交叠时产生的有效载荷传输时长为 $2EP$，所以公式 6-28 的物理含义就是：一次成功传输中不同传输情况的概率乘以此时产生的有效载荷传输时长，相加得到总的有效载荷传输时长的均值。

求解计算

经过上述分析，求解过程如下，首先依据基本模型，有：

$$\mathrm{b}_{00} = \begin{cases} \frac{2(1-p)(1-2p)}{(1-2p)(1-p^{r+1})+W_0(1-p)(1-(2p)^{r+1})} & r \le m \\ \frac{2(1-p)(1-2p)}{W_0(1-(2p)^{m+1})(1-p)+(1-2p)(1-p^{r+1})+W_0 2^m p^{m+1}(1-p^{r-m})(1-2p)} & m < r \end{cases} \tag{6-29}$$

另外有：

$$b_{i,k} = \frac{W_i - k}{W_i} * b_{i,0} \quad 0 \le i \le r, 0 \le k \le W_i - 1 \tag{6-30}$$

故可以得出不交叠概率 p_h

$$\begin{aligned} p_h =& \sum_{i=l_p}^{r} \sum_{j=k+1}^{W_i-1} b_{i,j} \\ =& \sum_{i=l_p}^{r} b_{i,0} \sum_{n=k+1}^{W_i-1} \frac{W_i - n}{W_i} \\ =& \sum_{i=l_p}^{r} b_{i,0} \frac{(W_i - k - 1)(W_i - k)}{2W_i} \\ =& \sum_{i=l_p}^{r} p^i b_{0,0} \frac{(W_i - k - 1)(W_i - k)}{2W_i} \end{aligned} \tag{6-31}$$

同时有 $p = P_e + (1 - P_e)(1 - p_h)$，$\tau = b_{0,0} * \frac{1-p^{r+1}}{1-p}$，在 MATLAB 中联立这些式子，额外注意 r, m, k 的相对关系，利用 fsolve 函数求解该非线性方程组，既可以求得答案。这里需要注意 r 与 m 的相对值，具体过程可以在 MATLAB 相应代码中查看。

则此时 Markov chain 上的每一个状态的稳态概率可以求得，那么可以求解无关交叠概率 p_u，额外注意 r, m, k k_2 的相对关系。

$$\begin{aligned} p_u &= \sum_{i=l_p}^{r} \sum_{j=k+1}^{k_2} b_{i,j} \\ &= \sum_{i=l_p}^{r} b_{i,0} \sum_{n=k+1}^{k_2} \frac{W_i - n}{W_i} \\ &= \sum_{i=l_p}^{r} p^i b_{0,0} \frac{(2W_i - k - k_2 - 1)(k_2 - k)}{2W_i} \end{aligned} \tag{6-32}$$

综上所述，进行归一化吞吐的求解：

$$S^* = \frac{P_s P_{tr} (\frac{p_h * 2E[P]}{p_h} + \frac{(p_h - p_u) * E[P]}{p_h})}{(1 - P_{tr})\,\sigma + P_s P_{tr}((p_u T_{s_u} + (p_h - p_u) T_s + P_{tr}(1 - P_s) T_c))} \tag{6-33}$$

此处 P_{tr}，P_s 由上述重构中可知，从而按照原式（4-14）可以求解出归一化吞吐，然后考虑物理层速率 $SPEED$ 则可以求出吞吐。按照题目附录 6 所给的数据，我们求出结果如Table 6.1所示。

表 6.1　问题三数学模型计算不同参数下的吞吐

附录参数序号	归一化吞吐	实际吞吐/Mbps
1	0.14442	44.41043
2	0.13430	36.67841
3	0.14368	44.51861
4	0.19007	30.45981
5	0.18413	26.04794
6	0.18961	34.67287

6.3.3 仿真验证

在仿真算法中，我们设置节点个数为 $num_nodes = 2$,$sim_time = 10000000$(一千万微秒)。由于 AP1 与 AP2 间处于不互听的状态，两者会始终认为信道始终处于空闲状态，因此在每个循环会直接在给节点传输信道状态时，直接传递 IDLE_STATE。对于 SIR 较小导

致数据包交叠时的发包失败，我们会通过检测两个节点发送数据包的起始和结束时间实现。运行如上仿真代码，我们可以得到归一化吞吐 S^* 与最终吞吐 S 大小如Table 6.2所示：

表 6.2　问题三仿真实验计算不同参数下的吞吐

附录参数序号	归一化吞吐	实际吞吐/Mbps	归一化吞吐相对误差
1	0.15794	45.29719	9.361%
2	0.13057	37.44748	2.777%
3	0.15769	45.30293	5.575%
4	0.20025	31.71960	5.356%
5	0.17053	27.01195	7.386%
6	0.22264	35.26618	10.689%

我们可以看到仿真算法计算结果与数学模型计算结果吻合的较好。我们选择附录参数序号为 1 的参数，同样设置不同载荷长度进行对比试验。两者运行结果如图 6.9 所示。

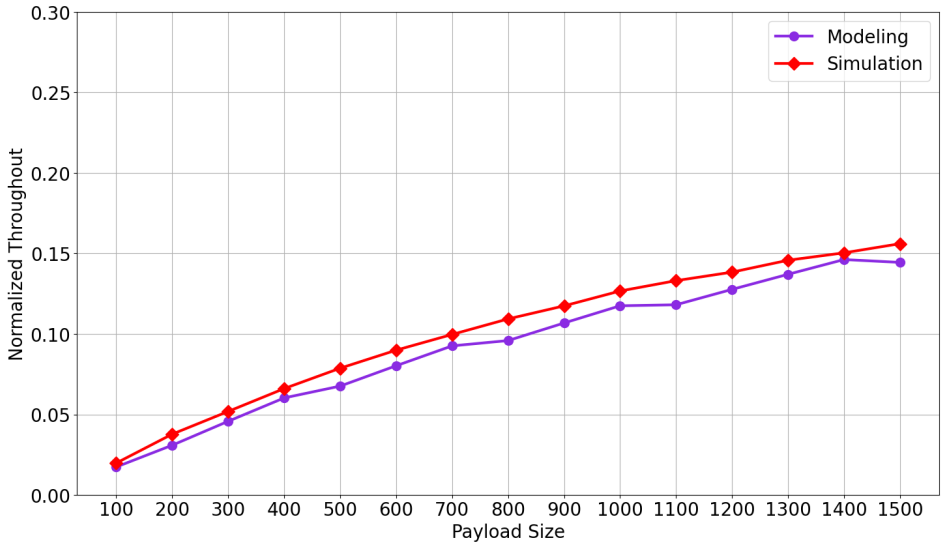

图 6.9　问题三不同载荷长度下仿真与数学模型结果对比

在不同载荷长度设置下，**归一化吞吐的平均相对误差为** 9.787%, 仍然说明数学模型与仿真结果具有较高的吻合度。

6.4 问题四

6.4.1 模型建立

在问题四中，我们需要考虑 3BSS 模型，其中 AP1 与 AP2 之间，AP2 与 AP3 之间 RSSI 均为-70dBm，AP1 与 AP3 之间 RSSI 为-96dBm。这意味着 AP1 与 AP3 不互听，AP2 与两者都互听，如图所示。题目中假设 AP1 和 AP3 发包时间交叠时，SIR 较大，两者发送均成功。而并没有给出 AP2 与两者之间的 SIR 关系，所以，我们需要按照 SIR 的大小来分类考虑。当 AP2 与两者之间的 SIR 较高时，意味着模型中无论是碰撞还是交叠都不会导致信号传输的失败，此时模型趋近于问题二中模型。当 AP2 与两者之间的 SIR 较低时，意味着若 AP2 与 AP1 或者 AP3 发生并行传输时，传输一定失败，该情况下传输模型需要重新建立，也是我们重点考虑的情况。以下模型建立只针对于 AP2 与两者间 SIR 较低的情况。

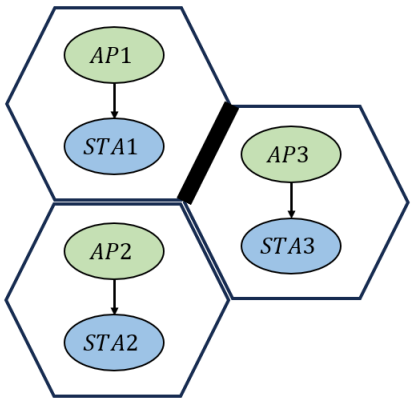

图 6.10　互听不互听关系示意图

此时仍然适用前文的方法，首先分析问题四条件下的信道状态。本题目中对于系统而言，信道的状态依然只有空闲、传输成功、传输失败三种，分析信道传输失败时三个 AP 可能存在的状态，有几种情况：

1. AP1 和 AP2 出现并行传输
2. AP3 和 AP2 出现并行传输
3. AP1、AP2、AP3 三者并行传输

在此时，我们可以仿照前文的做法，通过对节点建模，然后求出传输概率 P_{tr} 与条件传输成功概率 P_s 从而求解，但是这时，针对第四问，我们注意到节点之间不存在对称性了，即无法通过分析其中一个节点就得出另一个节点的情况。分析此问题，我们发现由于三个 AP 之间的互听情况不一致，其中 AP1 和 AP3 为等价的，因为他们都和 AP2 互听，但是二者之间不互听；AP2 则是另一种情况，它与 AP1、AP3 互听，所以在这里我们需要分析两种类型的节点。可以预见的是 AP2 与其他两者之间 τ，p 不同，AP2 的发送机会被 AP1 和 AP3 挤占。

所以，不妨设 AP1、AP3 为节点类型 A，AP2 为节点类型 B。由于信道还是只在三种状态中转移，对他们分别建立二维 Markov chain，节点 A 的状态为 $b_{i,k}^{A}$，节点 B 为 $b_{i,k}^{B}$，利用两个 Markov chain 之间的关系进行求解相关的参数。

对于最终的归一化吞吐的求解，可以仿照问题三中 S 的建模思路，针对问题四的条件求解相关的参数，得到新的 S 表示。

6.4.2 模型求解

建模 Markov chain 的目的就是通过求解条件传输失败概率 p 和发送概率 τ，从而求解出信道中的 P_{tr} 与 P_s，在这里由于两个节点的不对称性，不妨设节点 A 中的参数为 p_A 和 τ_A，节点 B 中的参数为 p_B 和 τ_B。

那么在问题四中，对于节点 A 进行分析，以 AP1 为例，其与 AP2 互听，所以当其与 AP2 并发时会传输失败，同时其与 AP3 不互听，可能出现与 AP3 交叠的情况，但是由于交叠时,SIR 较高，不会造成传输失败，所以当且仅当与 AP2 并发时会导致传输失败。故节点 A 的 Markov chain 中，退避器升阶的概率 p_A 有：

$$p_A = 1 - (1 - \tau_B) \tag{6-34}$$

对节点 B 进行分析，其与 AP1、AP3 都互听，所以其与 AP1 和 AP3 任意一个并发的时候会导致传输失败，故对节点 B，其 p_B 有：

$$p_B = 1 - (1 - \tau_A)^2 \tag{6-35}$$

同时，在两者自身的 Markov chain 中，分别有：

$$\tau_A = b_{0,0}^{A} * \frac{1 - p_A^{r+1}}{1 - p_A} \qquad \tau_B = b_{0,0}^{B} * \frac{1 - p_B^{r+1}}{1 - p_B} \tag{6-36}$$

从而可以联立上述四个式子（5-25）(5-26)（5-27），在 matlab 中求解非线性方程，可以得到答案。在这之后可以建立出信道上的三种状态的分布概率。

对于传输概率 P_tr，其为至少有一个节点在发送的概率：

$$P_{tr} = 1 - (1 - \tau_A)^2(1 - \tau_B) \tag{6-37}$$

对于条件传输成功概率，可以看作以下三种情况：1.AP1、AP3 只有一个节点发送，AP2 不发送；2.AP1 与 AP3 都不发送，AP2 发送；3.AP1 和 AP3 都发送，AP2 不发送。从而 P_s 有：

$$P_s = \frac{C_2^1 \tau_A (1 - \tau_A)(1 - \tau_B) + (1 - \tau_A)^2 \tau_B + \tau_A^2 (1 - \tau_B)}{P_{tr}} \tag{6-38}$$

但是，在这里我们同样需要考虑如问题三中，图6.7所示的由于交叠产生的总时长减少的现象。

在本问题中，由于交叠信号的 SIR 较高，所以传输能成功，从而我们考虑交叠影响的时间，此处令 p_o 表示交叠区间从数据包头，即 PHY 的第一刻，到成功传输结束即 ACK 的最后一刻的概率，如下图所示时间段，为了便于后续讨论，我们令与数据段交叠的概率为 p_{o_1}，无关交叠段为 p_{o_2}。

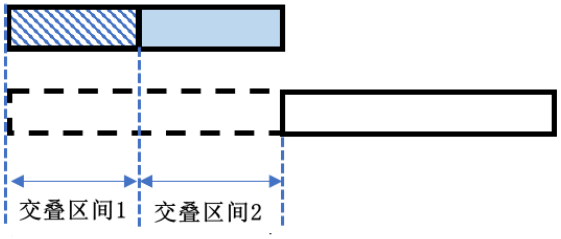

图 6.11　交叠时间示意图

类似的，由于我们在上方已经求解出 Markov chain 中的每一个状态的稳态解。所以对于节点类型 A（AP1、AP3），来说 p_{o_1} 与 p_{o_2} 分别为：

$$p_{o_1}=\sum_{i=0}^{r}\sum_{j=k}^{k_2}b_{i,j}\qquad p_{o_2}=\sum_{i=l_p}^{r}\sum_{j=k+1}^{k_2}b_{i,j} \tag{6-39}$$

其中 k 与 k_2 和问题三中意义一致。

那么类似的，对于归一化吞吐的计算中，对于 T_s 的各个概率，有以下：

$$\begin{cases} T_s & ,1-p_{o_1}-p_{o_2} \\ \frac{1}{2}(0+T_s+EP) & ,p_{o_1} \\ \frac{1}{2}(EP+T_s+2T_s) & ,p_{o_2} \end{cases} \tag{6-40}$$

对于 $E[P]'$，同理可得：

$$\begin{cases} EP & ,1-p_{o_1}-p_{o_2} \\ \frac{3}{2}EP & ,p_{o_1} \\ 2EP & ,p_{o_2} \end{cases} \tag{6-41}$$

从而可以利用式4-14计算出归一化吞吐 S^* 为：

$$S^*=\frac{P_sP_{tr}[\frac{3}{2}EPp_{o_1}+2EPp_{o_2}+(1-p_{o_1}-p_{o_2})EP]}{(1-P_{tr})\,\sigma+P_sP_{tr}[T_s(1-p_{o_1}-p_{o_2})+\frac{1}{2}(T_s+EP)p_{o_1}+\frac{3}{2}(EP+T_s)p_{o_2}]+P_{tr}\,(1-P_s)\,T_c} \tag{6-42}$$

从而系统吞吐量为 $S=S^*\times SPEED$。

在 Matlab 中计算可得结果为：

表 6.3　问题四不同参数下的吞吐

附录参数序号	归一化吞吐	实际吞吐/Mbps
1	0.35196	96.03677
2	0.28294	77.04132
3	0.35197	96.03752
4	0.51349	77.16336
5	0.44696	64.17043
6	0.51351	78.16417

6.4.3 仿真验证

在仿真算法中，我们设置节点个数为 $num_nodes = 3$,$sim_time = 10000000$(一千万微秒)。此时 AP1 与 AP3 不互听，AP2 与两者都互听，我们可以认为在每个循环传递给 AP2 的是真实的信道状态 channel_state，而传递给 AP1 和 AP3 的是一个“不完整”的信道状态 sub_channel_state，这种状态仅取决于 AP2 此时的状态。针对 AP1 和 AP3 发包时间可能的交叠，处理与问题 2、3 一致。运行如上仿真代码，我们可以得到归一化吞吐 S^* 与最终吞吐 S 大小如Table 6.4所示：

表 6.4　问题四仿真实验计算不同参数下的吞吐

附录参数序号	归一化吞吐	实际吞吐/Mbps	归一化吞吐相对误差
1	0.33549	96.21853	4.680%
2	0.27023	77.50196	4.492%
3	0.33478	96.01490	4.884%
4	0.49221	77.96606	4.323%
5	0.40941	64.85054	8.401%
6	0.49268	78.51571	4.056%

我们可以看到仿真算法计算结果与数学模型计算结果吻合的很好。同样如问题 3 中所示，选择附录参数序号为 1 的参数，设置不同载荷长度进行对比试验。两者运行结果如图 6.12 所示。

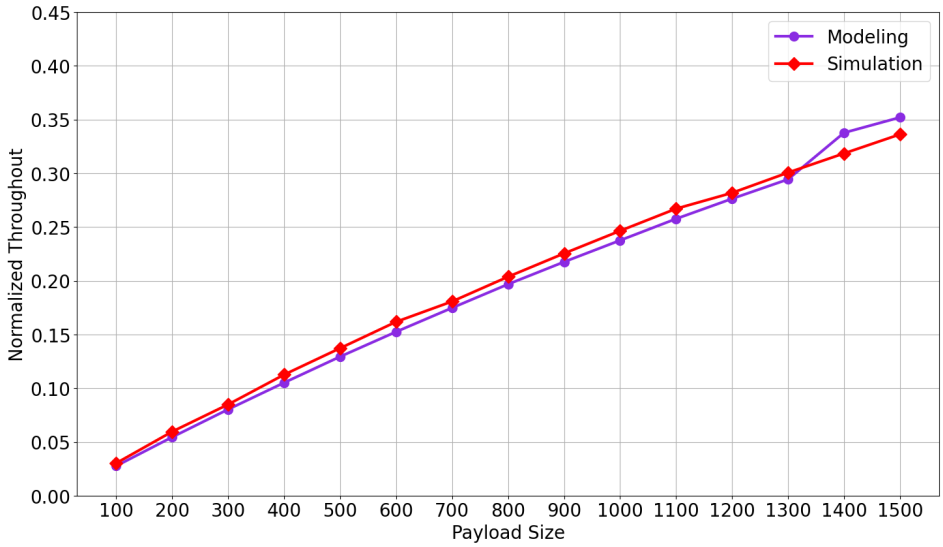

图 6.12　问题四不同载荷长度下仿真与数学模型结果对比

在不同载荷长度设置下，**归一化吞吐的平均相对误差仅为** 4.846%, 进一步说明数学模型与仿真结果具有极好的吻合度。

7. 模型评价

7.1 优点

1. 利用 Markov 链精确描述了 802.11 DCF 机制中节点竞争信道的过程。模型考虑了节点退避时间的递增, 以及竞争失败后重传的过程, 可以准确反映实际网络的行为;
2. 建立了状态转移矩阵, 给出了状态转移概率的表达式。这使得模型可以进行数学分析, 能够推导出网络饱和吞吐量的精确表达式;
3. 扩展模型考虑了信道错误率、节点数等因素的影响, 使模型更贴近实际情况。还讨论了隐节点问题对性能的影响。

7.2 缺点

1. 问题三、问题四中处于交叠情况下信道传输时长的分析，应考虑结合实际传输进行进一步优化;
2. 当传输发生交叠时，本模型简化了传输过程，没有从 δ 时间对齐角度进行考虑，可以考虑结合这一角度进一步优化;
3. 问题四只考虑了 AP1、AP2 之间 SIR 较低，导致并行传输一定失败的情况，没有考虑实际传输中 SIR 不为定值，可能导致并行传输具有一定成功率的情况。

7.3 展望

多 BSS 组网形成的 WLAN 信道接入机制建模是通信领域中一个重要的研究课题，如家庭环境中多个发送端传输的准确性决定着网络环境的体验感，在实际通信场景中会包含多个 AP 与 STA 的交叉传输，同时信道也会存在一定的干扰，可以设计一种模型从简单的多 BSS 组网开始研究在大环境下复杂组网的问题。

在系统 BSS 节点较多的情况下对这些节点进行信道的接入机制设计较为复杂，希望可以基于本模型，建立更具体的复杂模型，通过将更复杂情况从基础的节点组网出发，扩大模型所能包含的节点数目，最终完成实际情况下的复杂模型建立。

.

参考文献

[1] Bianchi Giuseppe. IEEE 802.11-Saturation Throughput Analysis [J]. IEEE Communications Letters, 1998, 2(12):318-320.

[2] P. Chatzimisios, V. Vitsas and A. C. Boucouvalas, ”Throughput and delay analysis of IEEE 802.11 protocol,” Proceedings 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2002, pp. 168-174, doi: 10.1109/IWNA.2002.1241355.

[3] Hung, Fu-Yi, and Ivan Marsic. ”Performance analysis of the IEEE 802.11 DCF in the presence of the hidden stations.” Computer Networks 54.15 (2010): 2674-2687.

[4] 付丽. 隐藏站点对非饱和 IEEE802.11WLANs 吞吐量的影响. Diss. 山东大学, 2014.

[5] M. Ni, Z. Zhong, J. Pan and D. Zhao, ”Saturation performance analysis of IEEE 802.11 broadcast in 2-D mobile ad hoc networks,” 2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC), Limassol, Cyprus, 2012, pp. 1069-1073, doi: 10.1109/IWCMC.2012.6314354.

附录 A 问题一求解程序

```
global r;
global m;
global node_num;

%参数定义
r = 32;
CW_min = 16;
CW_max = 1024;
m = log2(CW_max/CW_min);
node_num = 2;
ACK = 32;
SIFS = 16;
DIFS = 43;
SLOT = 9;
ACKTimeout = 65;

PAYLOAD = 1500;
MAC = 30;
PHY = 13.6;
SPEED = 455.8;

p_t = fsolve(@bianchi_p_tau,[0, 0],optimset('Display','off'));
   %求解马尔科夫链中的转移概率，以及发送概率
p = p_t(1);
tau = p_t(2);

P_tr = 1 - (1-tau)^node_num;
P_s = node_num * tau * (1-tau)^(node_num-1) / (1 -
   (1-tau)^node_num);

H = MAC * 8 / SPEED + PHY;
EP = PAYLOAD * 8 / SPEED;

T_s = H + EP +SIFS + ACK + DIFS;
T_c = H + EP + DIFS + ACKTimeout;

S = P_s*P_tr*EP / ((1-P_tr)*SLOT + P_tr*P_s*T_s + P_tr*(1-P_s)*T_c);
```

```
disp('归一化吞吐量为:')
disp(S)

disp('实际吞吐量为:')
disp(S*SPEED)
```

附录 B 问题二求解程序

```
%参数定义
r = 32;
CW_min = 16;
CW_max = 1024;
m = log2(CW_max/CW_min);
node_num = 2;
ACK = 32;
SIFS = 16;
DIFS = 43;
SLOT = 9;
ACKTimeout = 65;

PAYLOAD = 1500;
MAC = 30;
PHY = 13.6;
SPEED = 275.3;

% 一维马尔科夫链求解
b0 = 2 / (1 + CW_min);
tau = b0;

%计算吞吐量
P_tr = 1 - (1-tau)^node_num;
P_tr1 = 2*tau*(1-tau);
P_tr2 = tau^2;
%P_s = node_num * tau * (1-tau)^(node_num-1) / (1 -
   (1-tau)^node_num);
P_s = 1;
H = MAC * 8 / SPEED + PHY;
```

```
EP = PAYLOAD * 8 / SPEED;

T_s = H + EP +SIFS + ACK + DIFS;
T_c = T_s;

S = P_s*P_tr*EP / ((1-P_tr)*SLOT + P_tr*P_s*T_s + P_tr*(1-P_s)*T_c);
S1 = P_s*P_tr1*EP / ((1-P_tr)*SLOT + P_tr*P_s*T_s +
   P_tr*(1-P_s)*T_c);
S2 = P_s*P_tr2*EP / ((1-P_tr)*SLOT + P_tr*P_s*T_s +
   P_tr*(1-P_s)*T_c);

disp('归一化吞吐量为:')
disp(S)

disp('实际吞吐量为:')
disp(S*SPEED)

disp('实际吞吐量为:')
disp(S1*SPEED+S2*2*SPEED)
```

附录 C 问题三求解程序

```
global CW_max
global CW_min;
global r;
global m;
global node_num;
global ACK;
global SIFS;
global DIFS;
global SLOT;
global ACKTimeout;
global PAYLOAD;
global MAC;
global PHY;
global SPEED;
global P_e;
global p_nonoverlap;
```

```
global b00;
global l_p;
global k;

node_num = 2;
ACK = 32;
SIFS = 16;
DIFS = 43;
SLOT = 9;
ACKTimeout = 65;
PAYLOAD = 1500;
MAC = 30;
PHY = 13.6;
P_e = 0.1;


fprintf(' 参数序号  改进归一化吞吐量   实际吞吐量  \n');
rs = [6 5 32 6 5 32];
CW_mins = [16 32 16 16 32 16];
CW_maxs = [1024, 1024, 1024, 1024, 1024, 1024];
SPEEDs = [286.8 286.8 286.8 158.4 158.4 158.4];
for idx = 1:6
   r = rs(idx);
   CW_min = CW_mins(idx);
   CW_max = CW_maxs(idx);
   SPEED = SPEEDs(idx);
   m = log2(CW_max/CW_min);


   if r <= m
      p_t = fsolve(@q3_markov_solver_1,[0.1,
         0.1],optimset('Display','off'));
   else
      p_t = fsolve(@q3_markov_solver_2,[0.1,
         0.1],optimset('Display','off'));
   end

   p = p_t(1);
   tau = p_t(2);
```

```
P_tr = 1 - (1-tau)^node_num;
P_s = node_num * tau * (1-tau) * (1-P_e) * p_nonoverlap / (1 -
   (1-tau)^node_num);

H = MAC * 8 / SPEED + PHY;

EP = PAYLOAD * 8 / SPEED;

T_s = H + EP +SIFS + ACK + DIFS;

k_2 = T_s / SLOT;

p_new_h = 0;

if r <= m
   for i=l_p:r
      if 2^i*CW_min <= k_2
         p_new_h = p_new_h + p^i * b00 * (2^i*CW_min - k -
            1)*(2^i*CW_min - k) / (2*2^i*CW_min);
      else
         p_new_h = p_new_h + p^i * b00 * (k_2 - k)*(2^i*CW_min -
            k - 1 + 2^i*CW_min - k_2) / (2*2^i*CW_min);
      end
   end
else
   for i=l_p:m
      if 2^i*CW_min <= k_2
         p_new_h = p_new_h + p^i * b00 * (2^i*CW_min - k -
            1)*(2^i*CW_min - k) / (2*2^i*CW_min);
      else
         p_new_h = p_new_h + p^i * b00 * (k_2 - k)*(2^i*CW_min -
            k - 1 + 2^i*CW_min - k_2) / (2*2^i*CW_min);
      end
   end
   for i=m+1:r
      p_new_h = p_new_h + p^i * b00 * (k_2 - k)*(2^i*CW_min - k
         - 1 + 2^i*CW_min - k_2) / (2*2^i*CW_min);
   end
```

```
    end


    T_c = H + EP + DIFS + ACKTimeout;
    T_s_new = (H + EP) / 2 + 3*T_s / 2;

    S = P_s*P_tr*EP / ((1-P_tr)*SLOT + P_tr*P_s*T_s +
        P_tr*(1-P_s)*T_c);
    S_new =
        P_s*P_tr*((p_new_h*2*EP/p_nonoverlap)+((p_nonoverlap-p_new_h)*EP/p_nonove
        / ((1-P_tr)*SLOT + P_tr*P_s*((p_new_h * T_s_new) +
        (p_nonoverlap-p_new_h)*T_s) + P_tr*(1-P_s)*T_c);

    fprintf('  %6d   %13.5f   %9.5f  \n', idx, S_new, S*SPEED);
end
```

附录 D 问题四求解代码

```
global CW_max
global CW_min;
global r;
global m;
global node_num;
global ACK;
global SIFS;
global DIFS;
global SLOT;
global ACKTimeout;
global PAYLOAD;
global MAC;
global PHY;
global SPEED;
global b001;
global p1;

node_num = 3;
ACK = 32;
SIFS = 16;
```

```
DIFS = 43;
SLOT = 9;
ACKTimeout = 65;

PAYLOAD = 1500;
MAC = 30;
PHY = 13.6;

fprintf('  参数序号  改进归一化吞吐量    实际吞吐量  \n');
rs = [6 5 32 6 5 32];
CW_mins = [16 32 16 16 32 16];
CW_maxs = [1024, 1024, 1024, 1024, 1024, 1024];
SPEEDs = [286.8 286.8 286.8 158.4 158.4 158.4];
for idx = 1:6
   r = rs(idx);
   CW_min = CW_mins(idx);
   CW_max = CW_maxs(idx);
   SPEED = SPEEDs(idx);
   m = log2(CW_max/CW_min);

   k = (PHY + MAC * 8 / SPEED + PAYLOAD * 8 / SPEED) / SLOT;
   k = ceil(k);
   l_p = ceil(log2((k+1)/CW_min));
   if l_p < 0
      l_p = 0;
   end

   if (r <= m)
      p_t = fsolve(@q4_markov_solver_1,[0.1, 0.1, 0.1,
         0.1],optimset('Display','off'));
   else
      p_t = fsolve(@q4_markov_solver_2,[0.1, 0.1, 0.1,
         0.1],optimset('Display','off'));
   end

   p1 = p_t(1);
   p2 = p_t(2);
   tau1 = p_t(3);
   tau2 = p_t(4);
```

```
P_tr = 1-(1-tau1)^2*(1-tau2);

P_s = (2*tau1*(1-tau1)*(1-tau2) + (1-tau1)^2*tau2 +
   tau1^2*(1-tau2)) / P_tr;

H = MAC * 8 / SPEED + PHY;
EP = PAYLOAD * 8 / SPEED;

T_s = H + EP +SIFS + ACK + DIFS;
T_s_new = 3*T_s / 2;
T_c = H + EP + DIFS + ACKTimeout;

k_2 = T_s / SLOT;

p_old_h = 0; %p1
p_new_h = 0; %p2

if r<=m
   for i=0:r
      if 2^i*CW_min <= k_2
         p_old_h = p_old_h + p1^i * b001 * (2^i*CW_min + 1) / 2;
      else
         p_old_h = p_old_h + p1^i * b001 * (k + 1) * (2^i*CW_min
            + 2^i*CW_min - k) / (2*2^i*CW_min);
      end
   end
else
   for i=0:m
      if 2^i*CW_min <= k_2
         p_old_h = p_old_h + p1^i * b001 * (2^i*CW_min + 1) / 2;
      else
         p_old_h = p_old_h + p1^i * b001 * (k + 1) * (2^i*CW_min
            + 2^i*CW_min - k) / (2*2^i*CW_min);
      end
   end
   for i=m+1:r
      p_old_h = p_old_h + p1^i * b001 * (k + 1) * (2^i*CW_min +
         2^i*CW_min - k) / (2*2^i*CW_min);
```

```
    end
end


if r <= m
    for i=l_p:r
        if 2^i*CW_min <= k_2
            p_new_h = p_new_h + p1^i * b001 * (2^i*CW_min - k -
                1)*(2^i*CW_min - k) / (2*2^i*CW_min);
        else
            p_new_h = p_new_h + p1^i * b001 * (k_2 - k)*(2^i*CW_min
                - k - 1 + 2^i*CW_min - k_2) / (2*2^i*CW_min);
        end
    end
else
    for i=l_p:m
        if 2^i*CW_min < k_2
            p_new_h = p_new_h + p1^i * b001 * (2^i*CW_min - k -
                1)*(2^i*CW_min - k) / (2*2^i*CW_min);
        else
            p_new_h = p_new_h + p1^i * b001 * (k_2 - k)*(2^i*CW_min
                - k - 1 + 2^i*CW_min - k_2) / (2*2^i*CW_min);
        end
    end
    for i=m+1:r
        p_new_h = p_new_h + p1^i * b001 * (k_2 - k)*(2^i*CW_min -
            k - 1 + 2^i*CW_min - k_2) / (2*2^i*CW_min);
    end
end


S = P_s*P_tr*EP / ((1-P_tr)*SLOT + P_tr*P_s*T_s +
    P_tr*(1-P_s)*T_c);
S_new = P_s*P_tr*((3/2*EP*p_old_h + 2*EP*p_new_h +
    (1-p_old_h-p_new_h)*EP)) / ((1-P_tr)*SLOT +
    P_tr*P_s*((p_new_h * T_s_new) + (1-p_new_h)*T_s) +
    P_tr*(1-P_s)*T_c);

fprintf('  %6d   %13.5f   %9.5f  \n', idx, S_new, S*SPEED);
```

```
end
```

附录 E 问题三、四求解函数

5.1 问题三求解函数 $r \leq m$

```
% r <= m
function output = q3_markov_solver_1(input)
   global CW_max
   global CW_min;
   global r;
   global m;
   global node_num;
   global ACK;
   global SIFS;
   global DIFS;
   global SLOT;
   global ACKTimeout;
   global PAYLOAD;
   global MAC;
   global PHY;
   global SPEED;
   global P_e;
   global p_nonoverlap
   global b00;
   global l_p;
   global k;

   k = (PHY + MAC * 8 / SPEED + PAYLOAD * 8 / SPEED) / SLOT;
   k = ceil(k);
   l_p = ceil(log2((k+1)/CW_min));
   if l_p < 0
      l_p = 0;
   end

   p = input(1);
   tau = input(2);

   b00 = 2*(1-p)*(1-2*p) / ((1-2*p)*(1-p^(r+1)) +
```

```
      CW_min*(1-p)*(1-(2*p)^(r+1)));
   %b00 = 2*(1-p)*(1-2*p) / (CW_min*(1-(2*p)^(m+1))*(1-p) +
      (1-2*p)*(1-p^(r+1)) +
      CW_min*(2^m)*p^(m+1)*(1-p^(r-m))*(1-2*p));
   p_nonoverlap = 0;

   for i=l_p:r
      p_nonoverlap = p_nonoverlap + p^i * b00 * (2^i*CW_min - k -
         1)*(2^i*CW_min - k) / (2*2^i*CW_min);
   end

   %output(1) = (1 - p_nonoverlap) + p_nonoverlap * P_e - p;
   output(1) = P_e + (1-P_e)*(1-p_nonoverlap) - p;
   %output(1) = (1 - p_nonoverlap)- p_nonoverlap*P_e - p;
   %output(1) = 1 - p_nonoverlap*(1-P_e) - p;
   %output(1) = P_e + tau*(1-p_nonoverlap)*(1-P_e) - p;

   output(2) = b00 * (1-p^(r+1)) / (1-p) - tau;

end
```

5.2 问题三求解函数 $r > m$

```
% r > m
function output = q3_markov_solver_2(input)
   global CW_max
   global CW_min;
   global r;
   global m;
   global node_num;
   global ACK;
   global SIFS;
   global DIFS;
   global SLOT;
   global ACKTimeout;
   global PAYLOAD;
   global MAC;
   global PHY;
   global SPEED;
```

```
    global P_e;
    global p_nonoverlap;
    global b00;
    global l_p;
    global k;

    k = (PHY + MAC * 8 / SPEED + PAYLOAD * 8 / SPEED) / SLOT;
    k = ceil(k);
    l_p = ceil(log2((k+1)/CW_min));
    if l_p < 0
        l_p = 0;
    end
    p = input(1);
    tau = input(2);

    %b00 = 2*(1-p)*(1-2*p) / ((1-2*p)*(1-p^(r+1)) +
        CW_min*(1-p)*(1-(2*p)^(r+1)));
    b00 = 2*(1-p)*(1-2*p) / (CW_min*(1-(2*p)^(m+1))*(1-p) +
        (1-2*p)*(1-p^(r+1)) +
        CW_min*(2^m)*p^(m+1)*(1-p^(r-m))*(1-2*p));

    p_nonoverlap = 0;
    for i=l_p:m
        p_nonoverlap = p_nonoverlap + p^i * b00 * (2^i*CW_min - k -
            1)*(2^i*CW_min - k) / (2*2^i*CW_min);
    end

    for i=m+1:r
        p_nonoverlap = p_nonoverlap + p^i * b00 * (CW_max - k -
            1)*(CW_max - k) / (2*CW_max);
    end

    output(1) = P_e + (1-P_e)*(1-p_nonoverlap) - p;
    output(2) = b00 * (1-p^(r+1)) / (1-p) - tau;

end
```

5.3 问题四求解函数 $r \leq m$

```
% r <= m
function output = q4_markov_solver_1(input)
   global CW_max
   global CW_min;
   global r;
   global m;
   global node_num;
   global ACK;
   global SIFS;
   global DIFS;
   global SLOT;
   global ACKTimeout;
   global PAYLOAD;
   global MAC;
   global PHY;
   global SPEED;
   global P_e;
   global p_nonoverlap
   global b001;
   global p1;

   p1 = input(1);
   p2 = input(2);
   tau1 = input(3);
   tau2 = input(4);

   b002 = 2*(1-p2)*(1-2*p2) / ((1-2*p2)*(1-p2^(r+1)) +
      CW_min*(1-p2)*(1-(2*p2)^(r+1)));
   b001 = 2*(1-p1)*(1-2*p1) / ((1-2*p1)*(1-p1^(r+1)) +
      CW_min*(1-p1)*(1-(2*p1)^(r+1)));

   output_tmp_tau2 = b002 * (1-p2^(r+1)) / (1-p2);
   output_tmp_tau1 = b001 * (1-p1^(r+1)) / (1-p1);

   output_tmp_p2 = 1 - (1 - tau1) ^ 2;
   output_tmp_p1 = tau2;

   output(1) = output_tmp_p1 - p1;
   output(2) = output_tmp_p2 - p2;
```

```
    output(3) = output_tmp_tau1 - tau1;
    output(4) = output_tmp_tau2 - tau2;
end
```

5.4 问题四求解函数 $r > m$

```
% r > m
function output = q4_markov_solver_2(input)
    global CW_max
    global CW_min;
    global r;
    global m;
    global node_num;
    global ACK;
    global SIFS;
    global DIFS;
    global SLOT;
    global ACKTimeout;
    global PAYLOAD;
    global MAC;
    global PHY;
    global SPEED;
    global P_e;
    global p_nonoverlap
    global b001;
    global p1;

    p1 = input(1);
    p2 = input(2);
    tau1 = input(3);
    tau2 = input(4);

    b002 = 2*(1-p2)*(1-2*p2) / (CW_min*(1-(2*p2)^(m+1))*(1-p2) +
        (1-2*p2)*(1-p2^(r+1)) +
        CW_min*(2^m)*p2^(m+1)*(1-p2^(r-m))*(1-2*p2));
    b001 = 2*(1-p1)*(1-2*p1) / (CW_min*(1-(2*p1)^(m+1))*(1-p1) +
        (1-2*p1)*(1-p2^(r+1)) +
        CW_min*(2^m)*p1^(m+1)*(1-p1^(r-m))*(1-2*p1));
```

```
    output_tmp_tau2 = b002 * (1-p2^(r+1)) / (1-p2);
    output_tmp_tau1 = b001 * (1-p1^(r+1)) / (1-p1);

    output_tmp_p2 = 1 - (1 - tau1) ^ 2;
    output_tmp_p1 = tau2;

    output(1) = output_tmp_p1 - p1;
    output(2) = output_tmp_p2 - p2;
    output(3) = output_tmp_tau1 - tau1;
    output(4) = output_tmp_tau2 - tau2;
end
```

附录 F 仿真程序代码

6.1 节点类定义

```
from state_enum import TimerState,NodeState,ChannelState
import random
import math

class Node:

    def __init__(self, node_num, ACK, SIFS, DIFS, SLOT, ACKTimeout,
        CW_min, CW_max, max_retry, payload_length,
        phy_header_duration, mac_header_length, phy_rate,drop_rate):

        # ----------------------------------------#
        # 原始参数传递
        self.node_num = node_num
        self.ACK = ACK
        self.SIFS = SIFS
        self.DIFS = DIFS
        self.SLOT = SLOT
        self.ACKTimeout = ACKTimeout
        self.CW_min = CW_min
        self.CW_max = CW_max
        self.max_retry = max_retry
        self.payload_length = payload_length
        self.phy_header_duration = phy_header_duration
```

```
        self.mac_header_length = mac_header_length
        self.phy_rate = phy_rate
        self.drop_rate = drop_rate
        # ----------------------------------------#

        # ----------------------------------------#
        # 传输各时段时长设置
        # @formulation:这里没有考虑DIFS
        # Ts = H + E[P] + SIFS + ACK (+ DIFS)
        # T_c = H + E_P_collision (+ DIFS) + ACKTimeout

        # @param:

        # idel_time:载波侦听时长

        # backoff_time: 回退时隙长度

        # send_data_time:数据帧时长，这里为了方便仿真进行取整

        # wait_ACK_time: 若传输成功需要等待的时间

        # wait_ACKTIMEOUT_time: 若传输失败需要等待的时间
        self.idle_time = self.DIFS

        self.backoff_time = self.SLOT

        self.send_data_time = self.phy_header_duration +
            (self.mac_header_length + self.payload_length) * 8 /
            self.phy_rate
        self.send_data_time = math.ceil(self.send_data_time)

        self.wait_ACK_time = self.SIFS + self.ACK

        self.wait_ACKTIMEOUT_time = self.ACKTimeout
        # ----------------------------------------#

        # ----------------------------------------#
        # 状态机所需要参数设置
        # @param:
```

```
        # current_state:状态机当前状态

        # next_state: 状态机下一时刻应该的状态

        # CW: 当前时刻竞争窗口数

        # CW_counter:用于计数的竞争窗口数

        # prev_CW_counter:前一时刻竞争窗口数(wait状态使用)

        # current_try: 失败次数

        # TX_start_time: 发送数据包开始时间

        # TX_end_time: 发送数据包结束时间
        self.current_state = NodeState.IDLE
        self.next_state = NodeState.IDLE
        self.CW = self.CW_min
        self.CW_counter = 0
        self.prev_CW_counter = 0
        self.current_try = 0
        self.TX_start_time = 0
        self.TX_end_time = 0

        self.timer = self.Timer(self)
        # ----------------------------------------#


    class Timer:

        def __init__(self,outer_node):
            self.outer = outer_node
            self.state = TimerState.READY
            self.counter = 0
            self.prev_counter = 0


        def update(self,channel_state):
            # 当处于wait状态并且信道转为空闲时:
```

```
        if self.state == TimerState.WAIT and channel_state ==
           ChannelState.IDLE:

           # 当之前处于IDLE状态，接着DIFS计数
           if self.outer.current_state == NodeState.IDLE:
              self.state = TimerState.COUNTING
              self.counter = self.outer.DIFS

           # 当之前处于BACKOFF状态，接着BACKOFF计数
           if self.outer.current_state == NodeState.BACKOFF:
              self.prev_counter = self.counter
              self.outer.prev_CW_counter = self.outer.CW_counter

              self.state = TimerState.COUNTING
              self.counter = self.outer.DIFS

        # 节点计数时突然信道繁忙
        if self.state == TimerState.COUNTING and \
          channel_state in [ChannelState.TX,
             ChannelState.COLLISION]:

           # 转为等待状态
           if self.outer.current_state == NodeState.IDLE or
              self.outer.current_state == NodeState.BACKOFF:
              self.state = TimerState.WAIT

           # 自身处于传输状态，保持计数
           elif self.outer.current_state == NodeState.SEND_DATA or
              \
               self.outer.current_state == NodeState.WAIT_ACK or \
               self.outer.current_state ==
                  NodeState.WAIT_ACKTIMEOUT:

              self.counter = self.counter - 1
              if self.counter == 0:
                 self.state = TimerState.READY

        # 计数完成，可以状态机转换
        elif self.state == TimerState.COUNTING and channel_state
```

```
            == ChannelState.IDLE:

            self.counter = self.counter - 1
            if self.counter == 0:
               self.state = TimerState.READY


    def set(self, timer_state, timer_counter, next_state):
       self.timer.state = timer_state
       self.timer.counter = timer_counter
       self.next_state = next_state

    def run(self, channel_state, bus_clock):

       self.timer.update(channel_state)

       if self.timer.state == TimerState.READY:
          self.current_state = self.next_state

          # IDLE状态：计数DIFS
          if self.current_state == NodeState.IDLE:

             self.CW_counter = random.randint(0, self.CW - 1) + 1
             self.set(timer_state = TimerState.COUNTING,
                   timer_counter = self.idle_time,
                   next_state = NodeState.BACKOFF)

          # BACKOFF状态：计数SLOTTIME
          if self.current_state == NodeState.BACKOFF:

             if self.prev_CW_counter !=0 and self.timer.prev_counter
                !=0:
                self.set(timer_state = TimerState.COUNTING,
                      timer_counter = self.timer.prev_counter,
                      next_state = NodeState.BACKOFF)
                self.prev_CW_counter = 0
                self.timer.prev_counter = 0

             else:
```

```
            if self.CW_counter > 0:
                self.set(timer_state = TimerState.COUNTING,
                      timer_counter = self.backoff_time,
                      next_state = NodeState.BACKOFF)

            if self.CW_counter == 0 and self.timer.counter == 0:
                self.current_state = NodeState.SEND_DATA

            if self.CW_counter > 0:
                self.CW_counter = self.CW_counter - 1

        # SEND_DATA状态：计数数据包时长
        if self.current_state == NodeState.SEND_DATA:
            self.TX_start_time = bus_clock
            self.set(timer_state = TimerState.COUNTING,
                  timer_counter = self.send_data_time,
                  next_state = NodeState.UNKNOWN)

        if self.current_state == NodeState.UNKNOWN:
            # 根据丢包率判断传输是否成功
            random_number = random.random() # 生成一个0到1之间的随机数
            if random_number < self.drop_rate:
                # 代表丢包
                self.current_state = NodeState.WAIT_ACKTIMEOUT
            else:
                self.current_state = NodeState.WAIT_ACK

        # WAIT_ACK状态：传输成功
        if self.current_state == NodeState.WAIT_ACK:
            self.TX_end_time = bus_clock
            self.set(timer_state = TimerState.COUNTING,
                  timer_counter = self.wait_ACK_time,
                  next_state = NodeState.IDLE)

            self.current_try = 0
            self.CW = self.CW_min

        # WAIT_ACKTIMEOUT状态：传输失败
        if self.current_state == NodeState.WAIT_ACKTIMEOUT:
```

```
            self.TX_start_time = 0
            self.TX_end_time = 0
            self.set(timer_state = TimerState.COUNTING,
                  timer_counter = self.wait_ACKTIMEOUT_time,
                  next_state = NodeState.IDLE)

            self.current_try = self.current_try + 1
            if self.current_try == self.max_retry:
               self.CW = self.CW_min
               self.current_try = 0

            self.CW = min(self.CW *2,self.CW_max)
```

6.2 状态定义

```
from enum import Enum

class TimerState(Enum):
   WAIT = 1
   COUNTING = 2
   READY = 3


class NodeState(Enum):
   IDLE= 1
   BACKOFF = 2
   SEND_DATA = 3
   WAIT_ACK = 4
   WAIT_ACKTIMEOUT = 5
   UNKNOWN = -1

class ChannelState(Enum):
   IDLE = 1
   TX = 2
   COLLISION= 3
```

6.3 问题一主程序

```
import random
```

```
import math
from node import Node
from state_enum import TimerState, NodeState, ChannelState

if __name__ == "__main__":
    # ----------------------------------------#
    # 基本仿真参数设置
    num_nodes = 2 # 设置节点数量

    sim_time = 10000000 # 设置仿真时间
    # ----------------------------------------#

    # ----------------------------------------#
    # 通用参数设置
    ACK = 32 # ACK帧长度,单位微秒

    SIFS = 16 # SIFS间隔时长,单位微秒

    DIFS = 43 # DIFS间隔时长,单位微秒

    SLOT = 9 # 时隙长度,单位微秒

    ACKTimeout = 65 # ACK超时长度,单位微秒

    CW_min = 16 # 冲突窗口最小值

    CW_max = 1024 # 冲突窗口最大值

    max_retry = 32 # 最大重传次数
    # ----------------------------------------#

    # ----------------------------------------#
    # 题目具体参数设置
    payload_length = 1500 # AP发送数据包的载荷长度,单位字节

    phy_header_duration = 13.6 # PHY头时长,单位微秒

    mac_header_length = 30 # MAC头长度,单位字节
```

```
phy_rate = 455.8 # 物理层速率,单位Mbps
# phy_rate = 275.3 # 物理层速率,单位Mbps

drop_rate = 0 # 丢包率
# ----------------------------------------#

# ----------------------------------------#
# 仿真所需元素初始化
nodes = []

send_num = 0

send_index1 = []

send_index2 = []

TX_total_time = 0

channel_state = ChannelState.IDLE

log_messages = []

send_data_time = phy_header_duration + (mac_header_length +
   payload_length) * 8 / phy_rate

scale = (payload_length * 8 / phy_rate) / send_data_time

# 初始化每个节点并添加到节点列表中
for i in range(num_nodes):
   node = Node(node_num=i + 1,
               ACK=ACK,
               SIFS=SIFS,
               DIFS=DIFS,
               SLOT=SLOT,
               ACKTimeout=ACKTimeout,
               CW_min=CW_min,
               CW_max=CW_max,
               max_retry=max_retry,
               payload_length=payload_length,
```

```
                phy_header_duration=phy_header_duration,
                mac_header_length=mac_header_length,
                phy_rate=phy_rate,
                drop_rate=drop_rate)
        nodes.append(node)

    for bus_clock in range(sim_time + 1):

        for node in nodes:
            node.run(channel_state, bus_clock)

        if send_index1:
            send_index1 = [
                node for node in send_index1 if node.current_state ==
                    NodeState.SEND_DATA]

        for node in nodes:
            if node.current_state == NodeState.SEND_DATA:
                if node not in send_index1:
                    send_index1.append(node)
                    send_index2.append(node)

        if send_index1:
            if len(send_index1) == 1:
                channel_state = ChannelState.TX

            elif len(send_index1) >= 2:
                for node in send_index1:
                    node.next_state = NodeState.WAIT_ACKTIMEOUT
                    # node.next_state = NodeState.WAIT_ACK
                    channel_state = ChannelState.COLLISION

        if send_index2:
            send_index2 = [
                node for node in send_index2 if node.current_state !=
                    NodeState.IDLE]
            if not send_index2:
                channel_state = ChannelState.IDLE
```

```
    # for node in nodes:
    #    if node.TX_start_time > 0 and node.TX_end_time > 0:
    #       TX_total_time = TX_total_time + (node.TX_end_time -
       node.TX_start_time) * scale
    #       node.TX_start_time = 0
    #       node.TX_end_time = 0

    node1 = None
    node2 = None

    # 遍历节点列表
    for node in nodes:
       if node.node_num == 1:
          node1 = node
       if node.node_num == 2:
          node2 = node

    if node1.TX_start_time > 0 and node1.TX_end_time > 0:
       if node2.TX_start_time == 0 and node2.TX_end_time == 0:
          TX_total_time = TX_total_time + \
             (node1.TX_end_time - node1.TX_start_time) * scale
          node1.TX_start_time = 0
          node1.TX_end_time = 0

       if node2.TX_start_time > 0 and node2.TX_end_time == 0:
          if node2.TX_start_time >= node1.TX_start_time +
             math.ceil(payload_length * 8 / phy_rate):
             TX_total_time = TX_total_time + \
                (node1.TX_end_time - node1.TX_start_time) * scale
             node1.TX_start_time = 0
             node1.TX_end_time = 0

          elif node2.TX_start_time >= node1.TX_start_time and
             node2.TX_start_time < node1.TX_start_time +
             math.ceil(payload_length * 8 / phy_rate):
             node1.TX_start_time = node1.TX_start_time
             node1.TX_end_time = node1.TX_end_time

    if node2.TX_start_time > 0 and node2.TX_end_time > 0:
```

```
        if node1.TX_start_time == 0 and node1.TX_end_time == 0:
            TX_total_time = TX_total_time + \
                (node2.TX_end_time - node2.TX_start_time) * scale
            node2.TX_start_time = 0
            node2.TX_end_time = 0

        if node1.TX_start_time > 0 and node1.TX_end_time == 0:
            if node1.TX_start_time >= node2.TX_start_time +
               math.ceil(payload_length * 8 / phy_rate):
               TX_total_time = TX_total_time + \
                   (node2.TX_end_time - node2.TX_start_time) * scale
               node2.TX_start_time = 0
               node2.TX_end_time = 0

            elif node1.TX_start_time >= node2.TX_start_time and
               node1.TX_start_time < node2.TX_start_time +
               math.ceil(payload_length * 8 / phy_rate):
               node2.TX_start_time = node2.TX_start_time
               node2.TX_end_time = node2.TX_end_time

    if node1.TX_start_time > 0 and node1.TX_end_time > 0 and \
    node2.TX_start_time > 0 and node2.TX_end_time > 0:
        TX_total_time = TX_total_time + (node1.TX_end_time -
            node1.TX_start_time) * scale + \
            math.fabs(node1.TX_start_time - node2.TX_start_time)
        node1.TX_start_time = 0
        node1.TX_end_time = 0
        node2.TX_start_time = 0
        node2.TX_end_time = 0

    # # 添加以下代码来打印信息
    # print(f"Bus Clock: {bus_clock}")
    # for i, node in enumerate(nodes):
    #     print(f"Node {i + 1}:")
    #     print(f" Current State: {node.current_state}, Next State:
        {node.next_state}, CW: {node.CW}, CW_counter:
        {node.CW_counter}")
    #     print(f" Timer State: {node.timer.state}, Timer Counter:
        {node.timer.counter}")
```

```
    # # 打印send_index1和send_index2的信息
    # print(f"Send Index 1 (Count: {len(send_index1)}):")
    # for i, node in enumerate(send_index1):
    #     print(f" Node {node.node_num}")

    # print(f"Send Index 2 (Count: {len(send_index2)}):")
    # for i, node in enumerate(send_index2):
    #     print(f" Node {node.node_num}")

    # print(f"Channel State: {channel_state}\n")

    # # 添加每个循环中的 print 消息到 log_messages 列表中
    # log_messages.append(f"Bus Clock: {bus_clock}")
    # for i, node in enumerate(nodes):
    #     log_messages.append(f"Node {i + 1}:")
    #     log_messages.append(f" Current State:
        {node.current_state}, Next State: {node.next_state}, CW:
        {node.CW}, CW_counter: {node.CW_counter}")
    #     log_messages.append(f" Timer State: {node.timer.state},
        Timer Counter: {node.timer.counter}")
    # log_messages.append(f"Channel State: {channel_state}")

    # # 添加send_index1和send_index2的信息
    # log_messages.append(f"Send Index 1 (Count:
        {len(send_index1)}):")
    # for i, node in enumerate(send_index1):
    #     log_messages.append(f" Node {node.node_num}")

    # log_messages.append(f"Send Index 2 (Count:
        {len(send_index2)}):")
    # for i, node in enumerate(send_index2):
    #     log_messages.append(f" Node {node.node_num}")
    # log_messages.append(f"TX_total_time: {TX_total_time}")
    # log_messages.append(f"\n")

throughput_all = TX_total_time / sim_time
print(throughput_all)
```

```
    # with open("simulation_log.txt", "w") as log_file:
    #     log_file.write("\n".join(log_messages))
```

6.4 问题二主程序

```
import random
import math
from node import Node
from state_enum import TimerState,NodeState,ChannelState

if __name__ == "__main__":
    # ----------------------------------------#
    # 基本仿真参数设置
    num_nodes = 2 # 设置节点数量

    sim_time = 10000000 # 设置仿真时间
    # ----------------------------------------#

    # ----------------------------------------#
    # 通用参数设置
    ACK = 32 # ACK帧长度,单位微秒

    SIFS = 16 # SIFS间隔时长,单位微秒

    DIFS = 43 # DIFS间隔时长,单位微秒

    SLOT = 9 # 时隙长度,单位微秒

    ACKTimeout = 65 # ACK超时长度,单位微秒

    CW_min = 16 # 冲突窗口最小值

    CW_max = 1024 # 冲突窗口最大值

    max_retry = 32 # 最大重传次数
    # ----------------------------------------#

    # ----------------------------------------#
    # 题目具体参数设置
```

```
# payload_length = 1500 # AP发送数据包的载荷长度,单位字节

payload_length = 1500 # AP发送数据包的载荷长度,单位字节

phy_header_duration = 13.6 # PHY头时长,单位微秒

mac_header_length = 30 # MAC头长度,单位字节

# phy_rate = 455.8 # 物理层速率,单位Mbps
phy_rate = 275.3 # 物理层速率,单位Mbps

drop_rate = 0 # 丢包率
# ----------------------------------------#


nodes = []
send_num = 0
send_index1 = []
send_index2 = []

TX_total_time = 0
channel_state = ChannelState.IDLE
log_messages = []

send_data_time = phy_header_duration + (mac_header_length +
   payload_length) * 8 / phy_rate
scale = (payload_length * 8 / phy_rate) / send_data_time

# 初始化每个节点并添加到节点列表中
for i in range(num_nodes):
   node = Node(node_num = i + 1,
               ACK = ACK,
               SIFS = SIFS,
               DIFS = DIFS,
               SLOT = SLOT,
               ACKTimeout = ACKTimeout,
               CW_min = CW_min,
               CW_max = CW_max,
               max_retry = max_retry,
```

```
                payload_length = payload_length,
                phy_header_duration = phy_header_duration,
                mac_header_length = mac_header_length,
                phy_rate = phy_rate,
                drop_rate = drop_rate)
        nodes.append(node)

    for bus_clock in range(sim_time + 1):

        for node in nodes:
            node.run(channel_state,bus_clock)

        if send_index1:
            send_index1 = [node for node in send_index1 if
                node.current_state == NodeState.SEND_DATA]

        for node in nodes:
            if node.current_state == NodeState.SEND_DATA:
                if node not in send_index1:
                    send_index1.append(node)
                    send_index2.append(node)

        if send_index1:
            if len(send_index1) == 1:
                channel_state = ChannelState.TX

            elif len(send_index1) >= 2:
                for node in send_index1:
                    # node.next_state = NodeState.WAIT_ACKTIMEOUT
                    node.next_state = NodeState.WAIT_ACK
                    channel_state = ChannelState.COLLISION

        if send_index2:
            send_index2 = [node for node in send_index2 if
                node.current_state != NodeState.IDLE]
            if not send_index2:
                channel_state = ChannelState.IDLE

        # for node in nodes:
```

```
#    if node.TX_start_time > 0 and node.TX_end_time > 0:
#       TX_total_time = TX_total_time + (node.TX_end_time -
   node.TX_start_time)
#       node.TX_start_time = 0
#       node.TX_end_time = 0

node1 = None
node2 = None

# 遍历节点列表
for node in nodes:
   if node.node_num == 1:
      node1 = node
   if node.node_num == 2:
      node2 = node

if node1.TX_start_time > 0 and node1.TX_end_time > 0 :
   if node2.TX_start_time == 0 and node2.TX_end_time == 0:
      TX_total_time = TX_total_time + (node1.TX_end_time -
         node1.TX_start_time) * scale
      node1.TX_start_time = 0
      node1.TX_end_time = 0

   if node2.TX_start_time > 0 and node2.TX_end_time == 0:
      if node2.TX_start_time >= node1.TX_start_time +
         math.ceil(payload_length * 8 / phy_rate):
         TX_total_time = TX_total_time + (node1.TX_end_time -
            node1.TX_start_time) * scale
         node1.TX_start_time = 0
         node1.TX_end_time = 0

      elif node2.TX_start_time >= node1.TX_start_time and
         node2.TX_start_time < node1.TX_start_time +
         math.ceil(payload_length * 8 / phy_rate):
         node1.TX_start_time = node1.TX_start_time
         node1.TX_end_time = node1.TX_end_time

if node2.TX_start_time > 0 and node2.TX_end_time > 0 :
   if node1.TX_start_time == 0 and node1.TX_end_time == 0:
```

```
            TX_total_time = TX_total_time + (node2.TX_end_time -
                node2.TX_start_time) * scale
            node2.TX_start_time = 0
            node2.TX_end_time = 0

        if node1.TX_start_time > 0 and node1.TX_end_time == 0:
            if node1.TX_start_time >= node2.TX_start_time +
                math.ceil(payload_length * 8 / phy_rate):
                TX_total_time = TX_total_time + (node2.TX_end_time -
                    node2.TX_start_time) * scale
                node2.TX_start_time = 0
                node2.TX_end_time = 0

            elif node1.TX_start_time >= node2.TX_start_time and
                node1.TX_start_time < node2.TX_start_time +
                math.ceil(payload_length * 8 / phy_rate):
                node2.TX_start_time = node2.TX_start_time
                node2.TX_end_time = node2.TX_end_time

    if node1.TX_start_time > 0 and node1.TX_end_time > 0 and \
    node2.TX_start_time > 0 and node2.TX_end_time > 0 :
        TX_total_time = TX_total_time + (node1.TX_end_time -
            node1.TX_start_time) * scale + \
                    math.fabs(node1.TX_start_time -
                        node2.TX_start_time)
        node1.TX_start_time = 0
        node1.TX_end_time = 0
        node2.TX_start_time = 0
        node2.TX_end_time = 0









    # # 添加以下代码来打印信息
    # print(f"Bus Clock: {bus_clock}")
    # for i, node in enumerate(nodes):
```

```
#     print(f"Node {i + 1}:")
#     print(f" Current State: {node.current_state}, Next State:
    {node.next_state}, CW: {node.CW}, CW_counter:
    {node.CW_counter}")
#     print(f" Timer State: {node.timer.state}, Timer Counter:
    {node.timer.counter}")

# # 打印send_index1和send_index2的信息
# print(f"Send Index 1 (Count: {len(send_index1)}):")
# for i, node in enumerate(send_index1):
#     print(f" Node {node.node_num}")

# print(f"Send Index 2 (Count: {len(send_index2)}):")
# for i, node in enumerate(send_index2):
#     print(f" Node {node.node_num}")

# print(f"Channel State: {channel_state}\n")

# # 添加每个循环中的 print 消息到 log_messages 列表中
# log_messages.append(f"Bus Clock: {bus_clock}")
# for i, node in enumerate(nodes):
#     log_messages.append(f"Node {i + 1}:")
#     log_messages.append(f" Current State:
    {node.current_state}, Next State: {node.next_state}, CW:
    {node.CW}, CW_counter: {node.CW_counter}")
#     log_messages.append(f" Timer State: {node.timer.state},
    Timer Counter: {node.timer.counter}")
# log_messages.append(f"Channel State: {channel_state}")

# # 添加send_index1和send_index2的信息
# log_messages.append(f"Send Index 1 (Count:
    {len(send_index1)}):")
# for i, node in enumerate(send_index1):
#     log_messages.append(f" Node {node.node_num}")

# log_messages.append(f"Send Index 2 (Count:
    {len(send_index2)}):")
# for i, node in enumerate(send_index2):
#     log_messages.append(f" Node {node.node_num}")
```

```
        # log_messages.append(f"TX_total_time: {TX_total_time}")
        # log_messages.append(f"\n")

    throughput_all = TX_total_time / sim_time
    print(throughput_all)


    # with open("simulation_log.txt", "w") as log_file:
    #     log_file.write("\n".join(log_messages))
```

6.5 问题三主程序

```
import random
import math
from node import Node
from state_enum import TimerState,NodeState,ChannelState

if __name__ == "__main__":
    # ----------------------------------------#
    # 基本仿真参数设置
    num_nodes = 2 # 设置节点数量

    sim_time = 10000000 # 设置仿真时间
    # ----------------------------------------#

    # ----------------------------------------#
    # 通用参数设置
    ACK = 32 # ACK帧长度,单位微秒

    SIFS = 16 # SIFS间隔时长,单位微秒

    DIFS = 43 # DIFS间隔时长,单位微秒

    SLOT = 9 # 时隙长度,单位微秒

    ACKTimeout = 65 # ACK超时长度,单位微秒

    CW_min = 16 # 冲突窗口最小值
```

```
CW_max = 1024 # 冲突窗口最大值

max_retry = 32 # 最大重传次数
# ----------------------------------------#

# ----------------------------------------#
# 题目具体参数设置
payload_length = 1500 # AP发送数据包的载荷长度,单位字节

phy_header_duration = 13.6 # PHY头时长,单位微秒

mac_header_length = 30 # MAC头长度,单位字节

# phy_rate = 455.8 # 物理层速率,单位Mbps
# phy_rate = 286.8 # 物理层速率,单位Mbps
phy_rate = 158.4 # 物理层速率,单位Mbps

drop_rate = 0.1 # 丢包率
# ----------------------------------------#


nodes = []
send_num = 0
send_index1 = []
send_index2 = []

TX_total_time = 0
channel_state = ChannelState.IDLE
log_messages = []

send_data_time = phy_header_duration + (mac_header_length +
   payload_length) * 8 / phy_rate
scale = (payload_length * 8 / phy_rate) / send_data_time

# 初始化每个节点并添加到节点列表中
for i in range(num_nodes):
   node = Node(node_num = i + 1,
               ACK = ACK,
               SIFS = SIFS,
```

```
                DIFS = DIFS,
                SLOT = SLOT,
                ACKTimeout = ACKTimeout,
                CW_min = CW_min,
                CW_max = CW_max,
                max_retry = max_retry,
                payload_length = payload_length,
                phy_header_duration = phy_header_duration,
                mac_header_length = mac_header_length,
                phy_rate = phy_rate,
                drop_rate = drop_rate)
        nodes.append(node)

    for bus_clock in range(sim_time + 1):

        for node in nodes:
            node.run(ChannelState.IDLE,bus_clock)

        if send_index1:
            send_index1 = [node for node in send_index1 if
                node.current_state == NodeState.SEND_DATA]

        for node in nodes:
            if node.current_state == NodeState.SEND_DATA:
                if node not in send_index1:
                    send_index1.append(node)
                    send_index2.append(node)

        if send_index1:
            if len(send_index1) == 1:
                channel_state = ChannelState.TX

            elif len(send_index1) >= 2:
                for node in send_index1:
                    node.next_state = NodeState.WAIT_ACKTIMEOUT
                    # node.next_state = NodeState.WAIT_ACK
                    channel_state = ChannelState.COLLISION

        if send_index2:
```

```
        send_index2 = [node for node in send_index2 if
            node.current_state != NodeState.IDLE]
        if not send_index2:
            channel_state = ChannelState.IDLE

    node1 = None
    node2 = None

    # 遍历节点列表
    for node in nodes:
        if node.node_num == 1:
            node1 = node
        if node.node_num == 2:
            node2 = node

    if node1.TX_start_time > 0 and node1.TX_end_time > 0 :
        if node2.TX_start_time == 0 and node2.TX_end_time == 0:
            TX_total_time = TX_total_time + (node1.TX_end_time -
                node1.TX_start_time) * scale
            node1.TX_start_time = 0
            node1.TX_end_time = 0

        if node2.TX_start_time > 0 and node2.TX_end_time == 0:
            if node2.TX_start_time >= node1.TX_start_time +
                math.ceil(payload_length * 8 / phy_rate):
                TX_total_time = TX_total_time + (node1.TX_end_time -
                    node1.TX_start_time) * scale
                node1.TX_start_time = 0
                node1.TX_end_time = 0

            elif node2.TX_start_time >= node1.TX_start_time and
                node2.TX_start_time < node1.TX_start_time +
                math.ceil(payload_length * 8 / phy_rate):
                node1.TX_start_time = node1.TX_start_time
                node1.TX_end_time = node1.TX_end_time

    if node2.TX_start_time > 0 and node2.TX_end_time > 0 :
        if node1.TX_start_time == 0 and node1.TX_end_time == 0:
            TX_total_time = TX_total_time + (node2.TX_end_time -
```

```
            node2.TX_start_time) * scale
        node2.TX_start_time = 0
        node2.TX_end_time = 0

    if node1.TX_start_time > 0 and node1.TX_end_time == 0:
        if node1.TX_start_time >= node2.TX_start_time +
            math.ceil(payload_length * 8 / phy_rate):
            TX_total_time = TX_total_time + (node2.TX_end_time -
                node2.TX_start_time) * scale
            node2.TX_start_time = 0
            node2.TX_end_time = 0

        elif node1.TX_start_time >= node2.TX_start_time and
            node1.TX_start_time < node2.TX_start_time +
            math.ceil(payload_length * 8 / phy_rate):
            node2.TX_start_time = node2.TX_start_time
            node2.TX_end_time = node2.TX_end_time

if node1.TX_start_time > 0 and node1.TX_end_time > 0 and \
node2.TX_start_time > 0 and node2.TX_end_time > 0 :
    TX_total_time = TX_total_time + (node1.TX_end_time -
        node1.TX_start_time) * scale + \
                math.fabs(node1.TX_start_time -
                    node2.TX_start_time)
    node1.TX_start_time = 0
    node1.TX_end_time = 0
    node2.TX_start_time = 0
    node2.TX_end_time = 0



# # 添加以下代码来打印信息
# print(f"Bus Clock: {bus_clock}")
# for i, node in enumerate(nodes):
#     print(f"Node {i + 1}:")
#     print(f" Current State: {node.current_state}, Next State:
    {node.next_state}, CW: {node.CW}, CW_counter:
    {node.CW_counter}")
#     print(f" Timer State: {node.timer.state}, Timer Counter:
```

```
        {node.timer.counter}")

    # # 打印send_index1和send_index2的信息
    # print(f"Send Index 1 (Count: {len(send_index1)}):")
    # for i, node in enumerate(send_index1):
    #     print(f" Node {node.node_num}")

    # print(f"Send Index 2 (Count: {len(send_index2)}):")
    # for i, node in enumerate(send_index2):
    #     print(f" Node {node.node_num}")

    # print(f"Channel State: {channel_state}\n")

    # # 添加每个循环中的 print 消息到 log_messages 列表中
    # log_messages.append(f"Bus Clock: {bus_clock}")
    # for i, node in enumerate(nodes):
    #     log_messages.append(f"Node {i + 1}:")
    #     log_messages.append(f" Current State:
        {node.current_state}, Next State: {node.next_state}, CW:
        {node.CW}, CW_counter: {node.CW_counter}")
    #     log_messages.append(f" Timer State: {node.timer.state},
        Timer Counter: {node.timer.counter}")
    # log_messages.append(f"Channel State: {channel_state}")

    # # 添加send_index1和send_index2的信息
    # log_messages.append(f"Send Index 1 (Count:
        {len(send_index1)}):")
    # for i, node in enumerate(send_index1):
    #     log_messages.append(f" Node {node.node_num}")

    # log_messages.append(f"Send Index 2 (Count:
        {len(send_index2)}):")
    # for i, node in enumerate(send_index2):
    #     log_messages.append(f" Node {node.node_num}")
    # log_messages.append(f"TX_total_time: {TX_total_time}")
    # log_messages.append(f"\n")

throughput_all = TX_total_time / sim_time
print(throughput_all)
```

```
    # with open("simulation_log.txt", "w") as log_file:
    #     log_file.write("\n".join(log_messages))
```

6.6 问题四主程序

```
import random
import math
from node import Node
from state_enum import TimerState,NodeState,ChannelState

if __name__ == "__main__":
    # ----------------------------------------#
    # 基本仿真参数设置
    num_nodes = 3 # 设置节点数量

    sim_time = 10000000 # 设置仿真时间
    # ----------------------------------------#

    # ----------------------------------------#
    # 通用参数设置
    ACK = 32 # ACK帧长度,单位微秒

    SIFS = 16 # SIFS间隔时长,单位微秒

    DIFS = 43 # DIFS间隔时长,单位微秒

    SLOT = 9 # 时隙长度,单位微秒

    ACKTimeout = 65 # ACK超时长度,单位微秒

    CW_min = 16 # 冲突窗口最小值

    CW_max = 1024 # 冲突窗口最大值

    max_retry = 32 # 最大重传次数
    # ----------------------------------------#
```

```
# ----------------------------------------#
# 题目具体参数设置
payload_length = 1500 # AP发送数据包的载荷长度,单位字节

phy_header_duration = 13.6 # PHY头时长,单位微秒

mac_header_length = 30 # MAC头长度,单位字节

# phy_rate = 455.8 # 物理层速率,单位Mbps
# phy_rate = 286.8 # 物理层速率,单位Mbps
phy_rate = 158.4 # 物理层速率,单位Mbps

drop_rate = 0 # 丢包率
# ----------------------------------------#

nodes = []
send_num = 0
send_index1 = []
send_index2 = []

TX_total_time = 0
channel_state = ChannelState.IDLE
log_messages = []

send_data_time = phy_header_duration + (mac_header_length +
   payload_length) * 8 / phy_rate
scale = (payload_length * 8 / phy_rate) / send_data_time

sub_channel_state = ChannelState.IDLE

# 初始化每个节点并添加到节点列表中
for i in range(num_nodes):
   node = Node(node_num = i + 1,
               ACK = ACK,
               SIFS = SIFS,
               DIFS = DIFS,
               SLOT = SLOT,
               ACKTimeout = ACKTimeout,
               CW_min = CW_min,
```

```
                CW_max = CW_max,
                max_retry = max_retry,
                payload_length = payload_length,
                phy_header_duration = phy_header_duration,
                mac_header_length = mac_header_length,
                phy_rate = phy_rate,
                drop_rate = drop_rate)
        nodes.append(node)

    for bus_clock in range(sim_time + 1):

        for node in nodes:
            if node.node_num == 1 or node.node_num == 3 :
                node.run(sub_channel_state,bus_clock)

        for node in nodes:
            if node.node_num == 2 :
                node.run(channel_state,bus_clock)

        if send_index1:
            send_index1 = [node for node in send_index1 if
                node.current_state == NodeState.SEND_DATA]

        for node in nodes:
            if node.current_state == NodeState.SEND_DATA:
                if node not in send_index1:
                    send_index1.append(node)
                    send_index2.append(node)

        if send_index1:
            if len(send_index1) == 1:
                channel_state = ChannelState.TX

            elif len(send_index1) >= 2:
                for node in send_index1:
                    if node.node_num == 1 or node.node_num == 3 :
                        node.next_state = NodeState.WAIT_ACK
                        channel_state = ChannelState.COLLISION
```

```
    if send_index2:
        send_index2 = [node for node in send_index2 if
            node.current_state != NodeState.IDLE]
        if not send_index2:
            channel_state = ChannelState.IDLE

    node1 = None
    node2 = None
    node3 = None

    # 遍历节点列表
    for node in nodes:
        if node.node_num == 1:
            node1 = node
        if node.node_num == 2:
            node2 = node
        if node.node_num == 3:
            node3 = node

    if node2.current_state == NodeState.IDLE or
        node2.current_state == NodeState.BACKOFF:
        sub_channel_state = ChannelState.IDLE

    if node2.current_state == NodeState.SEND_DATA or
        node2.current_state == NodeState.WAIT_ACK or \
    node2.current_state == NodeState.WAIT_ACKTIMEOUT:
        sub_channel_state = ChannelState.TX


    if node1.TX_start_time > 0 and node1.TX_end_time > 0 :
        if node3.TX_start_time == 0 and node3.TX_end_time == 0:
            TX_total_time = TX_total_time + (node1.TX_end_time -
                node1.TX_start_time) * scale
            node1.TX_start_time = 0
            node1.TX_end_time = 0

        if node3.TX_start_time > 0 and node3.TX_end_time == 0:
            if node3.TX_start_time >= node1.TX_start_time +
                math.ceil(payload_length * 8 / phy_rate):
```

```
            TX_total_time = TX_total_time + (node1.TX_end_time -
               node1.TX_start_time) * scale
            node1.TX_start_time = 0
            node1.TX_end_time = 0

         elif node3.TX_start_time >= node1.TX_start_time and
            node3.TX_start_time < node1.TX_start_time +
            math.ceil(payload_length * 8 / phy_rate):
            node1.TX_start_time = node1.TX_start_time
            node1.TX_end_time = node1.TX_end_time

   if node3.TX_start_time > 0 and node3.TX_end_time > 0 :
      if node1.TX_start_time == 0 and node1.TX_end_time == 0:
         TX_total_time = TX_total_time + (node3.TX_end_time -
            node3.TX_start_time) * scale
         node3.TX_start_time = 0
         node3.TX_end_time = 0

      if node1.TX_start_time > 0 and node1.TX_end_time == 0:
         if node1.TX_start_time >= node3.TX_start_time +
            math.ceil(payload_length * 8 / phy_rate):
            TX_total_time = TX_total_time + (node3.TX_end_time -
               node3.TX_start_time) * scale
            node3.TX_start_time = 0
            node3.TX_end_time = 0

         elif node1.TX_start_time >= node3.TX_start_time and
            node1.TX_start_time < node3.TX_start_time +
            math.ceil(payload_length * 8 / phy_rate):
            node3.TX_start_time = node3.TX_start_time
            node3.TX_end_time = node3.TX_end_time

   if node1.TX_start_time > 0 and node1.TX_end_time > 0 and \
   node3.TX_start_time > 0 and node3.TX_end_time > 0 :
      TX_total_time = TX_total_time + (node1.TX_end_time -
         node1.TX_start_time) * scale + \
                  math.fabs(node1.TX_start_time -
                     node3.TX_start_time)
      node1.TX_start_time = 0
```

```
        node1.TX_end_time = 0
        node3.TX_start_time = 0
        node3.TX_end_time = 0

    if node2.TX_start_time > 0 and node2.TX_end_time > 0 :
        TX_total_time = TX_total_time + (node2.TX_end_time -
            node2.TX_start_time) * scale
        node2.TX_start_time = 0
        node2.TX_end_time = 0



    # # 添加以下代码来打印信息
    # print(f"Bus Clock: {bus_clock}")
    # for i, node in enumerate(nodes):
    #     print(f"Node {i + 1}:")
    #     print(f" Current State: {node.current_state}, Next State:
        {node.next_state}, CW: {node.CW}, CW_counter:
        {node.CW_counter}")
    #     print(f" Timer State: {node.timer.state}, Timer Counter:
        {node.timer.counter}")

    # # 打印send_index1和send_index2的信息
    # print(f"Send Index 1 (Count: {len(send_index1)}):")
    # for i, node in enumerate(send_index1):
    #     print(f" Node {node.node_num}")

    # print(f"Send Index 2 (Count: {len(send_index2)}):")
    # for i, node in enumerate(send_index2):
    #     print(f" Node {node.node_num}")

    # print(f"Channel State: {channel_state}\n")

    # # 添加每个循环中的 print 消息到 log_messages 列表中
    # log_messages.append(f"Bus Clock: {bus_clock}")
    # for i, node in enumerate(nodes):
    #     log_messages.append(f"Node {i + 1}:")
    #     log_messages.append(f" Current State:
        {node.current_state}, Next State: {node.next_state}, CW:
```

```
        {node.CW}, CW_counter: {node.CW_counter}")
    #    log_messages.append(f" Timer State: {node.timer.state},
        Timer Counter: {node.timer.counter}")
    # log_messages.append(f"Channel State: {channel_state}")

    # # 添加send_index1和send_index2的信息
    # log_messages.append(f"Send Index 1 (Count:
        {len(send_index1)}):")
    # for i, node in enumerate(send_index1):
    #    log_messages.append(f" Node {node.node_num}")

    # log_messages.append(f"Send Index 2 (Count:
        {len(send_index2)}):")
    # for i, node in enumerate(send_index2):
    #    log_messages.append(f" Node {node.node_num}")
    # log_messages.append(f"TX_total_time: {TX_total_time}")
    # log_messages.append(f"\n")

throughput_all = TX_total_time / sim_time
print(throughput_all)
# print("payload_length:{},
    throughput_all:{}".format(payload_length,throughput_all))

# with open("simulation_log.txt", "w") as log_file:
#    log_file.write("\n".join(log_messages))
```