ALOHA**算法设计文档**

|  |  |  |  |
| --- | --- | --- | --- |
| **提交人** | **曾婷燕** | **状态** | |
| **标识号** |  | **讨论稿** | **√** |
| **版本号** |  | **初 稿** |  |
| **前一版本** |  | **评审稿** |  |
| **发布日期** |  | **正式稿** |  |
| **修订人** | **曾婷燕** | **修订稿** |  |
| **修订日期** |  | **其 它** |  |

**（版权所有，不得传播）**

目录

[1、 纯ALOHA算法（pure-aloha） 2](#_Toc472084920)

[1.1 纯ALOHA算法原理 2](#_Toc472084921)

[1.2 纯ALOHA基本思想 2](#_Toc472084922)

[1.3 纯ALOHA识别过程 2](#_Toc472084923)

[1.4 纯ALOHA算法性能分析 2](#_Toc472084924)

[1.5 算法matlab仿真代码 3](#_Toc472084925)

[2、时隙ALOHA算法（slotted-aloha） 6](#_Toc472084926)

[2.1 时隙ALOHA算法原理 6](#_Toc472084927)

[2.2 时隙ALOHA性能分析 6](#_Toc472084928)

[2.3 算法matlab仿真代码 7](#_Toc472084929)

[3、纯ALOHA与时隙ALOHA算法的比较 9](#_Toc472084930)

[4、帧时隙 ALOHA 算法 10](#_Toc472084931)

[4.1 帧时隙 ALOHA 算法（Framed Slotted ALOHA）思想： 10](#_Toc472084932)

[4.2 帧时隙 ALOHA 算法识别过程： 10](#_Toc472084933)

[4.3 算法matlab仿真 11](#_Toc472084934)

[5、动态帧时隙ALOHA(Dynamic Framed Slotted ALOHA)算法 17](#_Toc472084935)

[5.1 动态帧时隙ALOHA算法思想 17](#_Toc472084936)

[5.2 动态帧时隙ALOHA算法识别过程 17](#_Toc472084937)

[5.3 动态帧时隙ALOHA算法经典标签估读模型 19](#_Toc472084938)

[5.3.1 算法思想及仿真 19](#_Toc472084939)

[5.3.2 算法仿真结果对比 22](#_Toc472084940)

[6、参考文献 23](#_Toc472084941)

1. 纯ALOHA算法（pure-aloha）
   1. 纯ALOHA算法原理

纯ALOHA算法是是最简单最基本的一种防碰撞算法。算法基于时分多路法(TDMA)思想，是一种概率算法。当标签进入阅读器的作用区域时，就主动向阅读器发送其自身的信息，不同的标签发送信息的时间也是随机的，当阅读器准确的识别出唯一的标签就开始与该标签通信。

* 1. 纯ALOHA基本思想

标签随机发送自身的信息给阅读器，由于这种随机性，发送的数据帧很有可能发生冲突，标签得不到确认响应，也就是可以鉴别出它发送的数据帧被破坏，会重新选择一个时隙发送，等待时间随机。直至所有标签被识别。也可能存在标签饿死的情况。适合用于待识别标签数目不多的情况。

* 1. 纯ALOHA识别过程

对于有接受功能的标签，标签发送信息的过程中有其他标签也在同时发送数据，信号就会产生重叠，导致部分冲突或者完全冲突。一旦产生冲突，阅读器就发送命令让标签停止发送信息，然后随机等待一段时间再发送信息，不同的标签可能选择不同的时延，从而避免了冲突。如果没有冲突，阅读器正确读取标签信息后，标签进入休眠状态。对于无接收功能的标签，由于标签收不到阅读器发送的信息，在检测期间一直重复发送自己的信息，直到识别结束。

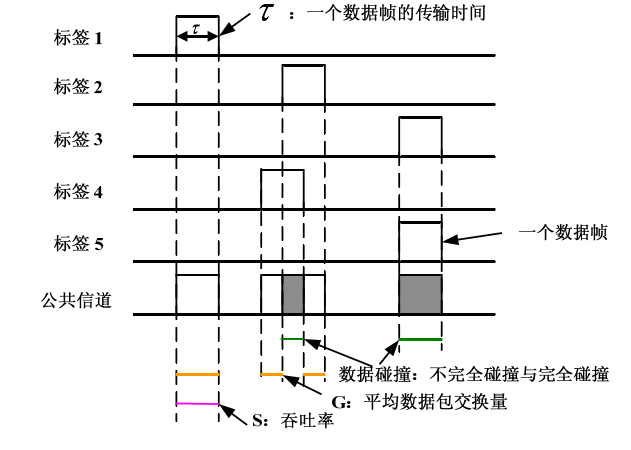


图1.1 纯ALOHA算法发生冲突情况

* 1. 纯ALOHA算法性能分析

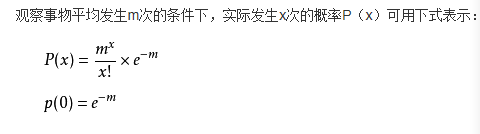
纯 ALOHA 算法的缺点是数据帧发送过程中碰撞发生的概率很大，冲突周期是2T0（T0表示一个标签发送数据包所用时间）。平均交换的数据包量G可以表示为





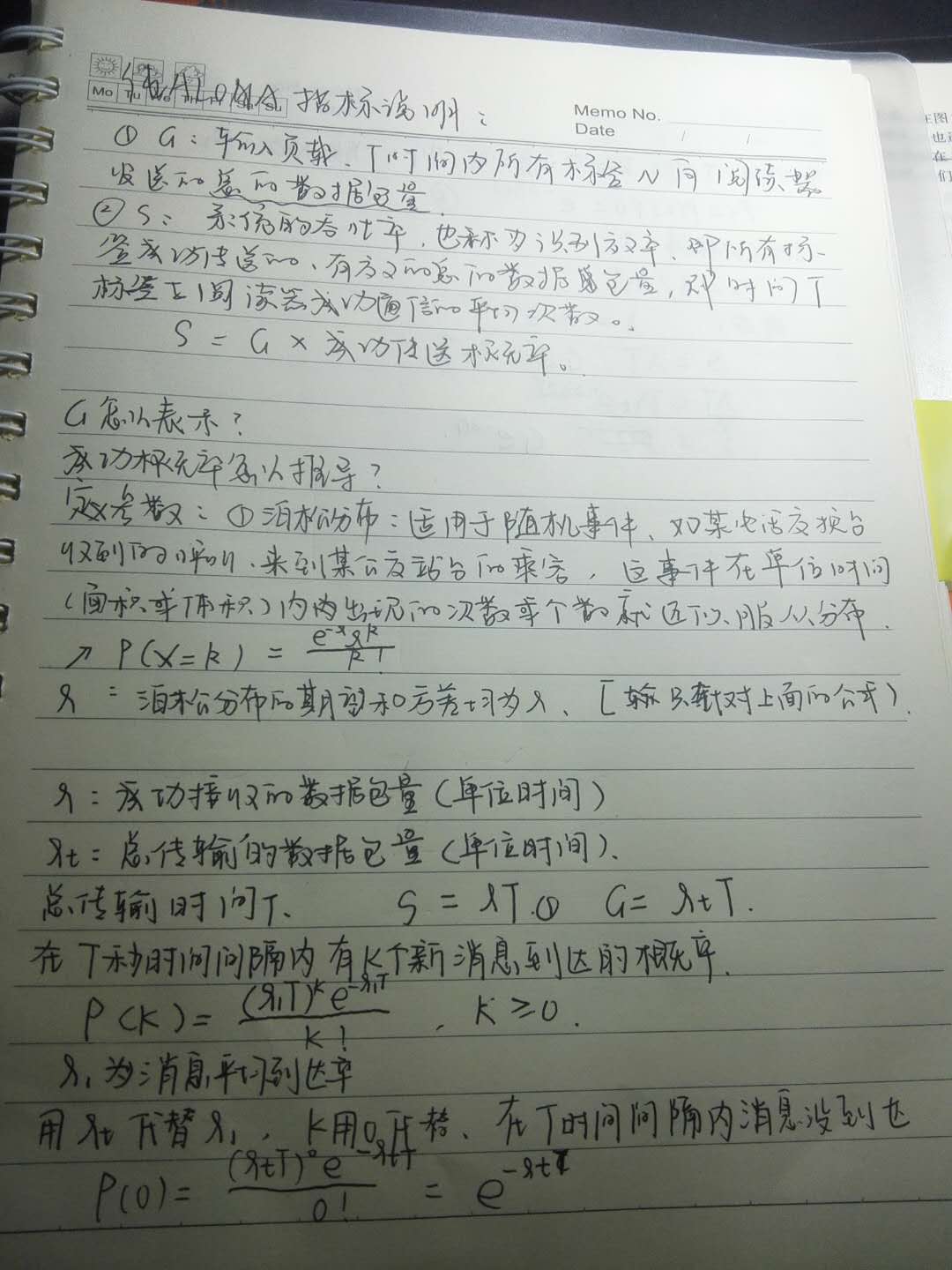
表示一个数据包的传输持续时间；n表示标签的数量；r 是在观察时间 T内由标签n发送的数据包量。

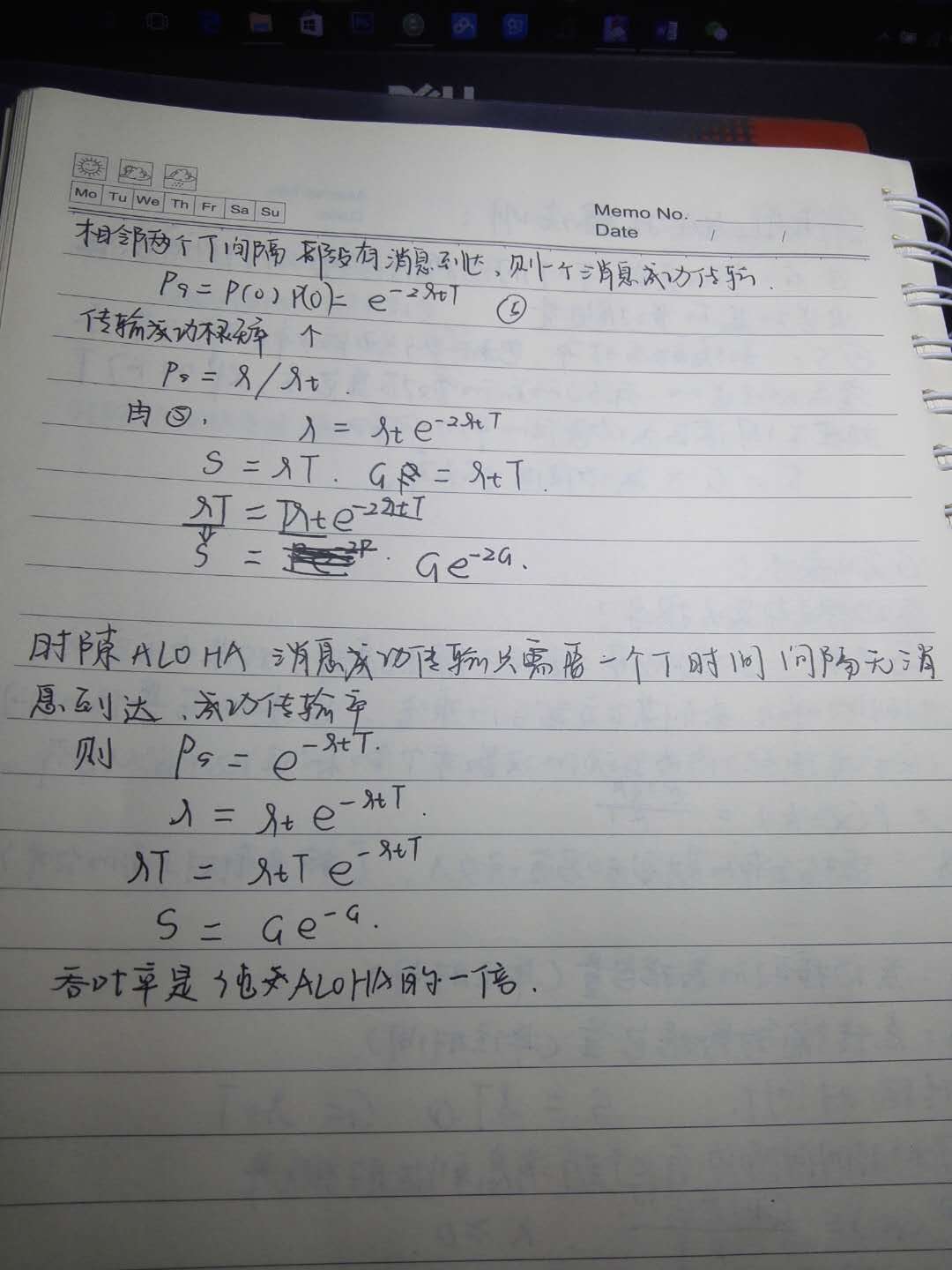
泊松分布适合于描述单位时间（或空间）内随机事件发生的次数。



吞吐率是关于数据通信系统数据传输率的指标。在通信系统中，这个指标通常是基于每秒能处理的数据位数或分组的数目，它依赖于网络的带宽和交换部件。在 RFID 系统中，当吞吐率 S等于 1，即是在传输期间无碰撞的传输数据包，等于0表示数据没有发送或者是由于碰撞不能准确读出传输的数据。传输通道的平均吞吐率为：







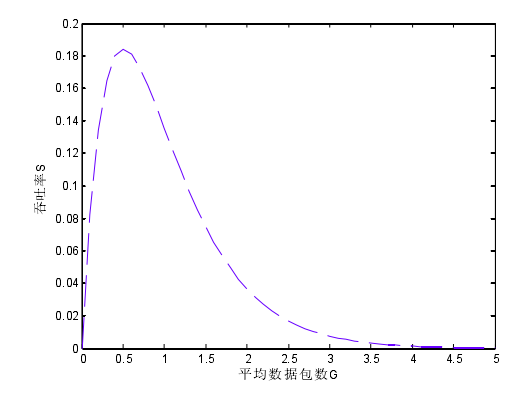
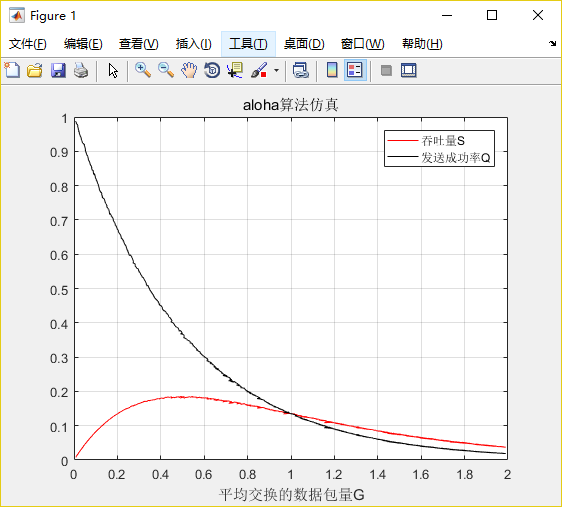


图1.2 纯ALOHA吞吐率S与平均数据包数G的关系（理论）

* 1. 算法matlab仿真代码

1. % pure aloha 算法仿真  
   % 注：帧=数据帧=数据包（定长）  
     
   function [G, S, Q] = aloha(hostNum, pktNum) % hostNum表示最大标签数，pktNum表示每个标签需传送数据包的个数  
    % pktNum也可理解为重发次数  
     
   if hostNum < 2 % 要发生标签碰撞事件，标签数必须>=2  
    error('hostNum must be bigger than 2'); % 若在main脚本文件里设置的 hostNum值小于2，仿真运行时则返回出错信息  
   end  
     
    G = zeros(1, hostNum - 1); % 负载（预分配内存空间）  
    S = zeros(1, hostNum - 1); % 平均每帧时的吞吐量，即为吞吐率（预分配内存空间）  
    Q = zeros(1, hostNum - 1); % 发送成功率（预分配内存空间）  
     
    factor = 1; % 假设随机发帧间隔等待时间服从0-1分布，故随机分布系数为1，可修改  
     
   for hosts = 2:hostNum % 函数运行时会产生一个临时工作区，函数运行完成后，自动将该临时工作区删除。中间变量hosts最终值等于hostNum  
    timePoint = cumsum(factor\*rand(pktNum, hosts)); % 随机矩阵表示随机发帧时间，矩阵‘cumsum’运算是列相加  
    observedTime = timePoint(pktNum, 1); % 设立观察时间  
     
    sequence = zeros(1, hosts\*pktNum); % 将矩阵 timePoint的所有数据存入行向量 sequence里，首先定义 sequence  
    for i = 1:pktNum % 类似C语言的int a，声明一个整形变量a，使其占用内存的一个空间（预分配内存）  
    sequence(1, (i-1)\*hosts+1 : i\*hosts) = timePoint(i, :); % timePoint 矩阵有 pktNum行，hostNum列  
    end % 上述赋值方法↑很简洁，适当做一下笔记 % 共 pktNum\*hostNum 个元素  
    sequence = sort(sequence); % 按时间顺序排列各个发帧时间（从左到右依次递增排序）  
     
    total = 0; % 总帧数  
    success = 0; % 无冲突帧数  
    frameTime = 0.004; % 帧时/帧长（设传输1帧需要4毫秒）（相当于1帧信息的长度为T0=0.004s）（即每个数据包的宽度）  
    interval = diff(sequence); % 帧间隔时间（对数组元素采用diff函数是差分的意思，就是说 dy(n)=y(n+1)-y(n)）  
    for i = 1:hosts\*pktNum - 1; % 所以得到的 interval数组会比 sequence数组少一个  
    if sequence(i+1) > observedTime  
    break; % 忽略观察时间以外的帧，break  
    end  
    total = total + 1; % 当 t0-T0<=t1<=t0+T0 时发生碰撞。设起始时间为t0，一帧信息的长度为T0，另一帧的起始时间设为t1  
    if (i == 1 || i == ( hosts\*pktNum -1 ) ) && interval(1, i) >= frameTime % 第一个和最后一个帧（数据包），只需考虑时间间隔大等于帧时  
    success = success + 1; %第一个数据帧只考虑尾部时间间隔；最后一个则只考虑前部时间间隔  
    end  
    if i ~= 1 && i ~= ( hosts\*pktNum -1 ) && interval(1, i) >= frameTime && interval(1, i-1) >= frameTime % 帧的前后间隔都应大等于帧时  
    success = success + 1;  
    end  
    end  
    G(hosts-1) = frameTime/observedTime\*total; % 计算负载  
    S(hosts-1) = frameTime/observedTime\*success; % 计算平均每帧时的吞吐量（吞吐率）  
    Q(hosts-1) = S/G; % 计算发送成功率  
   end  
   end % function aloha 结束



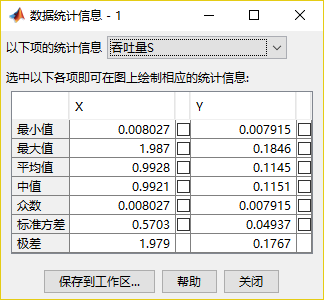


图1.3 算法仿真以及数据统计

可以看出，仿真结果与理论结果相近，对于纯ALOHA系统而言：当G>0.5时性能迅速恶化，系统进入不稳定状态。因为必须尽量使G不要太大，这样只能改小T0，同时也要注意由于标签自身的问题不能检测碰撞，因此只能重发，重发次数不一定越多越好，因为重发次数多了，会使得输入负载过大，碰撞率增加，因此要选适当的退避区间。

2、时隙ALOHA算法（slotted-aloha）

2.1 时隙ALOHA算法原理

为了避免像纯ALOHA 那样的不完全碰撞，为了提高系统的吞吐量，可将时间划分为一段一段等长的时隙，记为T0。规定数据帧只能在时隙的开始才能发送，这就是时隙ALOHA(Slotted ALOHA)算法。

每个时隙存在以下三种情况：

1. 空闲时隙
2. 成功时隙
3. 碰撞时隙

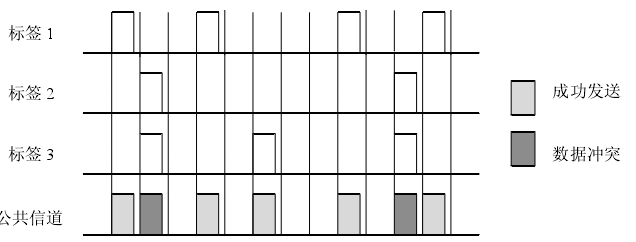


图2.1 时隙ALOHA算法碰撞情况

2.2 时隙ALOHA性能分析

时隙 ALOHA 法的吞吐量与平均交换的数据包含量 G的关系：



交换的数据包量在G=1时吞吐量Q取得最大值0.368，使得信道利用率增加一倍。如果有大量的标签处于阅读器的作用范围内，比如已存在标签在此时隙交换数据包，再加上另外到达的标签，那么吞吐率很快就接近于 0。在最不利的情况下，经过多次搜索也可能没有发现序列号，因为没有唯一的标签能单独处于一个时隙之中而发送成功。因此，需要准备足够大量的时隙，这样做法降低了防碰撞算法的性能。

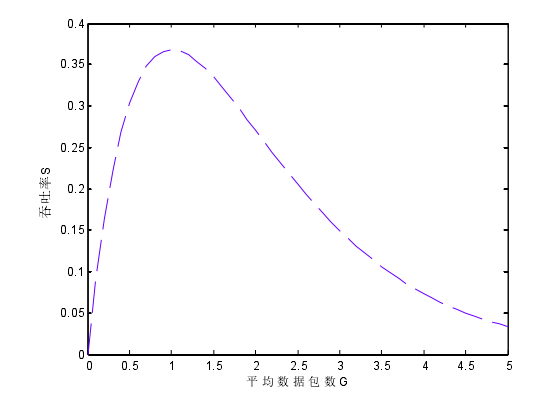


图2.2 时隙ALOHA吞吐量与平均发送数据包的关系（理论）

2.3 算法matlab仿真代码

分槽函数

% 时隙ALOHA算法是把时间分成多个离散的时隙，每个时隙的长度等于或稍大于一个帧，  
% 标签只能在每个时隙的开始处发送数据，这样标签或成功发送或者完全碰撞，  
% 避免了ALOHA算法的部分碰撞，使碰撞周期减半，提高了信道的利用率。  
  
function slot\_length = slotted(M, deta)  
s = size(M); % s=[pktNum,frameTime] %deta =0.005  
slot\_length = zeros(s(1), s(2));  
for i = 1:s(2)  
 for j = 1:s(1)  
 if M(j, i)/deta ~= floor(M(j, i)/deta) % 函数floor表示向下取整，如 floor(3.8) ans=3； "~"表示不等于，非  
 slot\_length(j, i) = (floor(M(j, i)/deta)+1)\*deta; % 每个时隙的长度要大等于标签回复的数据长度deta  
 end  
 end  
end  
end

% slotted aloha 算法仿真

% 数据帧要么成功发送，要么完全碰撞，避免了纯ALOHA算法中部分碰撞的发生，使碰撞周期变为T0

%重点在于采用diff函数

% slotted aloha 算法仿真  
% 数据帧要么成功发送，要么完全碰撞，避免了纯ALOHA算法中部分碰撞的发生，使碰撞周期变为T0  
  
function [G, S, Q] = s\_aloha(hostNum, pktNum)  
  
if hostNum < 2  
 error('hostNum must be bigger than 2');  
end  
  
G = zeros(1, hostNum - 1); % 负载  
S = zeros(1, hostNum - 1); % 平均每帧时的吞吐量，即为吞吐率  
Q = zeros(1, hostNum - 1); % 发送成功率  
  
factor = 1; % 随机分布系数  
frameTime = 0.005; % 帧时或者可认为是标签与读卡器的交换时间  
for hosts = 2:hostNum  
 % 随机矩阵，表示随机发帧时间，并分槽  
 timePoint = cumsum(slotted(factor\*rand(pktNum, hosts), frameTime)); % slotted函数功能  
 observedTime = timePoint(pktNum, 1); % 设立观察时间  
  
 sequence = zeros(1, hosts\*pktNum);  
 for i = 1:pktNum  
 sequence(1, (i-1)\*hosts+1 : i\*hosts) = timePoint(i, :);% 将timePoint的第i行元素赋给sequence的第(i-1)\*hosts+1和i\*hosts列元素  
 end  
 sequence = sort(sequence); % 按时间顺序排列各个发帧时间  
  
 total = 0; % 总帧数  
 success = 0; % 无冲突帧数  
 interval = diff(sequence); % 帧间隔时间  
 for i = 1: hosts\*pktNum - 1;  
 if sequence(i+1) > observedTime % 忽略观察时间以外的帧  
 break; % break  
 end  
 total = total + 1;  
 if interval(1, i) >= frameTime % 时间间隔大等于帧时，则无冲突  
 success = success + 1;  
 end  
 end  
 G(hosts-1) = frameTime/observedTime\*total; % 计算负载  
 S(hosts-1) = frameTime/observedTime\*success; % 计算平均每帧时的吞吐量（吞吐率）  
 Q(hosts-1) = S/G; % 计算发送成功率  
end  
end

main函数

close all;

clear;

clc;

hosts =500;

pkts = 1000;

[G2, S2, Q2] = s\_aloha(hosts, pkts);

plot(G2, S2, 'b', G2, Q2, 'g');

title('slotted ALOHA仿真');

legend('slotted ALOHA');

gtext('时隙ALOHA发送成功率');

xlabel('G(平均每帧时负载量)');

ylabel('S(平均每帧时吞吐量)');

axis([0 2 0 1]);

grid on;

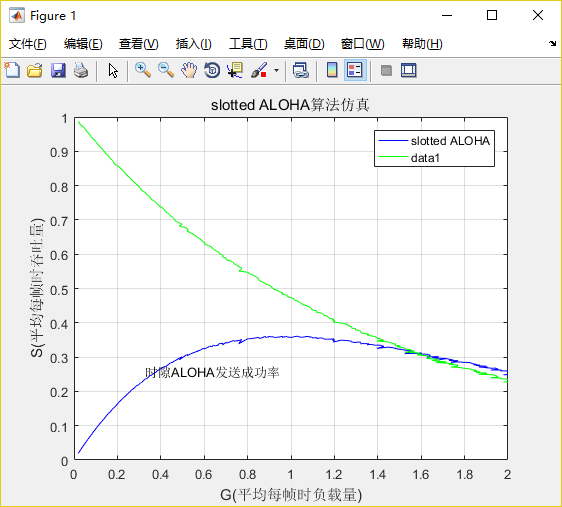


图2.3 算法仿真结果及数据分析

时隙ALOHA：当G=0.3612时，Smax=0.9313（大概值，每次均有些许的浮动）

（理论情况下：当G=1时，系统的吞吐率可以达到最大值0．368）

由多次仿真测试结果可得，影响吞吐率变化的因素主要有4个：标签数hostNum，发送数据帧数量pktNum，帧时frameTime以及随机发送间隔等待时间（退避时间）factor（算法中由随机分布系数factor确定）。

此外，标签数以及发包数越大，即离散测试点越多，所得到的图越平滑，越接近理想曲线。

3、纯ALOHA与时隙ALOHA算法的比较

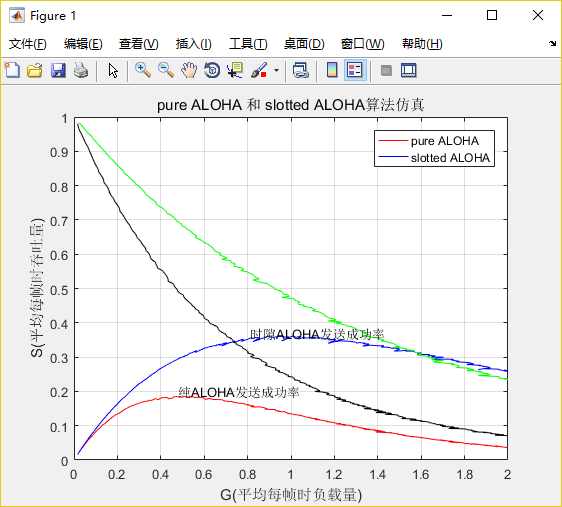


图3.1 纯ALOHA和时隙ALOHA算法比较

整体分析：

对于纯ALOHA系统而言：当G>0.5时性能迅速恶化，系统进入不稳定状态。因为必须尽量使G不要太大，这样只能改小T0，同时也要注意由于标签自身的问题不能检测碰撞，因此只能重发，重发次数不一定越多越好，因为重发次数多了，会使得输入负载过大，碰撞率增加，因此要选适当的退避区间。

对于时隙ALOHA系统而言：相关问题与纯ALOHA相仿。当G=1时，系统的吞吐率可以达到理论最大值0．368，由此可见时隙ALOHA算法的系统吞吐率比纯ALOHA算法提高了1倍。缺点是需要同步时钟，且标签能够计算时隙。

注意两种算法的相同与不同之处，其实有异曲同工之处，时隙ALOHA则更加巧妙，相比于纯ALOHA，大大降低了碰撞率，因为每个标签只能在时隙的开始发送数据包，如果判断出此时隙开端只有一个标签发送数据包，则为成功时隙，可以成功识别，成功通信。

4、帧时隙 ALOHA 算法

4.1 帧时隙 ALOHA 算法（Framed Slotted ALOHA）思想：

是时隙ALOHA上的改进,把多个时隙组成一个帧，标签在每帧内随机选择一个时隙发送信息，这种算法适用于传输信息量较大的场合,但是与时隙ALOHA算法相同，帧时隙 ALOHA 算法也需要一个同步开销。

4.2 帧时隙 ALOHA 算法识别过程：

算法识别过程：阅读器发送命令通知标签该帧内含有多少个时隙，标签在帧

时隙内随机选择一个时隙作为自己的发送时隙，然后阅读器发送命令通知标签目

前要读取的时隙，如果标签所选时隙与该时隙相同，那么标签就响应阅读器的命

令。如果该时隙有不止一个标签响应，就产生了冲突，无法读出任何一个标签信

息，阅读器通知标签在下一轮中重新选择发送时隙，直到标签被正确识别。如果

目前的时隙只有一个标签响应，就不会产生碰撞，能够正确识别出标签信息。

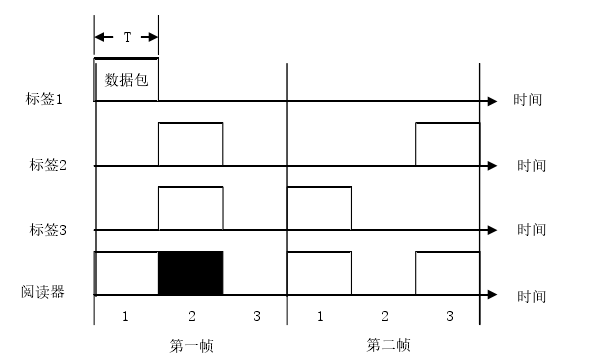


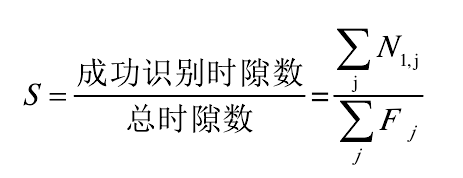
图4.1 帧时隙（FSA）算法碰撞原理图



图4.2 帧时隙（FSA）算法吞吐率与待识别标签数目关系（理论）

4.3 算法matlab仿真

说明：在这里的吞吐量（总的成功识别率）要和前面PA和SA的吞吐率区分开来，定义为总的成功识别的时隙数与总时隙数的比值，即：



4.3.1帧时隙ALOHA（Framed-Slotted ALOHA）算法主函数

% 帧时隙 ALOHA（Framed-Slotted ALOHA，FSA）算法仿真  
  
clear;  
close all;  
clc;  
  
N1 = 32; % 设置5种帧长情况  
N2 = 64;  
N3 = 96;  
N4 = 128;  
N5 = 256;  
  
S1 = zeros(500, 501); % 系统吞吐率  
S2 = zeros(500, 501);  
S3 = zeros(500, 501);  
S4 = zeros(500, 501);  
S5 = zeros(500, 501);  
  
S1\_avr = zeros(1, 501); % 系统平均吞吐率  
S2\_avr = zeros(1, 501);  
S3\_avr = zeros(1, 501);  
S4\_avr = zeros(1, 501);  
S5\_avr = zeros(1, 501);  
  
S1\_coll = zeros(500, 501); % 系统时隙碰撞率  
S2\_coll = zeros(500, 501);  
S3\_coll = zeros(500, 501);  
S4\_coll = zeros(500, 501);  
S5\_coll = zeros(500, 501);  
  
S1\_coll\_avr = zeros(1, 501); % 系统平均时隙碰撞率  
S2\_coll\_avr = zeros(1, 501);  
S3\_coll\_avr = zeros(1, 501);  
S4\_coll\_avr = zeros(1, 501);  
S5\_coll\_avr = zeros(1, 501);  
  
S1\_idle = ones(500, 501); % 初始帧长度为32时的空时隙率  
  
S1\_idle\_avr = ones(1, 501); % 初始帧长度为32时的平均空时隙率  
  
  
for cycle = 1:500 % 重复进行500次仿真测试  
 for TagsNum = 1:500 % 标签数目设置为1—500  
  
 [ succ1 , idle1 , coll1 ] = FSA\_anti ( TagsNum , N1 ); % succ1：成功时隙数；idle1：空闲时隙数；  
 S1(cycle, TagsNum+1) = succ1/N1; % coll1：碰撞时隙数  
 S1\_idle(cycle, TagsNum+1) = idle1/N1;  
 S1\_coll(cycle, TagsNum+1) = coll1/N1;  
  
 [ succ2 , idle2 , coll2 ] = FSA\_anti ( TagsNum , N2 );  
 S2(cycle, TagsNum+1) = succ2/N2;  
 S2\_coll(cycle, TagsNum+1) = coll2/N2;  
  
 [ succ3 , idle3 , coll3 ] = FSA\_anti ( TagsNum , N3 );  
 S3(cycle, TagsNum+1) = succ3/N3;  
 S3\_coll(cycle, TagsNum+1) = coll3/N3;  
  
 [ succ4 , idle4 , coll4 ] = FSA\_anti ( TagsNum , N4 );  
 S4(cycle, TagsNum+1) = succ4/N4;  
 S4\_coll(cycle, TagsNum+1) = coll4/N4;  
  
 [ succ5 , idle5 , coll5 ] = FSA\_anti ( TagsNum , N5 );  
 S5(cycle, TagsNum+1) = succ5/N5;  
 S5\_coll(cycle, TagsNum+1) = coll5/N5;  
  
 end  
end  
  
for i = 2:501 % 求平均值  
 S1\_avr(i)=sum(S1(:, i))/500;  
 S2\_avr(i)=sum(S2(:, i))/500;  
 S3\_avr(i)=sum(S3(:, i))/500;  
 S4\_avr(i)=sum(S4(:, i))/500;  
 S5\_avr(i)=sum(S5(:, i))/500;  
 S1\_coll\_avr(i)=sum(S1\_coll(:, i))/500;  
 S2\_coll\_avr(i)=sum(S2\_coll(:, i))/500;  
 S3\_coll\_avr(i)=sum(S3\_coll(:, i))/500;  
 S4\_coll\_avr(i)=sum(S4\_coll(:, i))/500;  
 S5\_coll\_avr(i)=sum(S5\_coll(:, i))/500;  
 S1\_idle\_avr(i)=sum(S1\_idle(:, i))/500;  
end  
  
i=1:501;  
figure(1)  
plot(i, S1\_avr, 'k', i, S2\_avr, 'r', i, S3\_avr, 'b', i, S4\_avr, 'g', i, S5\_avr, 'm');  
xlim([0 500]);  
title('Framed-Slotted ALOHA 算法仿真');  
legend('N=32', 'N=64', 'N=96', 'N=128', 'N=256', 'location', 'best');  
xlabel('n：待识别的标签数量');  
ylabel('S：吞吐率');  
grid on;  
  
figure(2)  
plot(i, S1\_coll\_avr, 'k', i, S2\_coll\_avr, 'r', i, S3\_coll\_avr, 'b', i, S4\_coll\_avr, 'g', i, S5\_coll\_avr, 'm');  
xlim([0 500]);  
title('Framed-Slotted ALOHA 算法仿真');  
legend('N=32', 'N=64', 'N=96', 'N=128', 'N=256', 'location', 'best');  
xlabel('n：待识别的标签数量');  
ylabel('时隙碰撞率');  
grid on;  
  
figure(3)  
plot(i, S1\_avr, 'k', i, S1\_coll\_avr, 'r', i, S1\_idle\_avr, 'b');  
xlim([0 500]);  
title('FSA 算法仿真（帧长=32）');  
legend('成功时隙率P(s)', '碰撞时隙率P(c)', '空闲时隙率P(i)', 'location', 'best');  
xlabel('标签数目');  
ylabel('效率');  
grid on;

4.3.2 帧时隙（Framed-Slotted ALOHA）碰撞处理函数

% 帧时隙 ALOHA（Framed-Slotted ALOHA，FSA）算法的碰撞处理函数  
  
function [ S\_succ , S\_idle , S\_coll ] = FSA\_anti ( TagsNum , FrameLength )  
  
S\_succ = 0; % 成功时隙数  
S\_idle = 0; % 空闲时隙数  
S\_coll = 0; % 碰撞时隙数  
  
SlotCounter = randi( [0 , FrameLength - 1] , 1 , TagsNum );  
% 标签从 0-（FrameLength-1）中随机选择一个整数存入时隙计数器，作为发送数据的时隙数  
  
SlotCheck = zeros(1, FrameLength); % 检测每帧中所有时隙的情况（成功/空闲/碰撞）  
  
for i = 1:FrameLength  
 for n = 1:TagsNum  
 if (i-1) == SlotCounter(n)  
 SlotCheck(i) = SlotCheck(i) + 1;  
 if SlotCheck(i) == 2  
 break  
 end  
 end  
 end  
end  
  
for i = 1:FrameLength  
 if SlotCheck(i) == 1  
 S\_succ = S\_succ +1;  
 elseif SlotCheck(i) == 0  
 S\_idle = S\_idle +1;  
 else  
 S\_coll = S\_coll + 1;  
 end  
end  
end



图4.3 FSA中各时隙状态概率比较结果图

从图中我们可以看出标签数越多，时隙的碰撞情况越明显，而空时隙和成功时隙的情况越来越少，其中成功识别时隙率也就是系统识别效率，增加到最大值 0.368后随着标签数增加而逐渐减小，当标签数远大于固定帧长度时，成功时隙率和空时隙率逐渐趋于0，而碰撞时隙率趋于 1，这说明此时每个时隙内都有标签碰撞情况发生，无空时隙和成功时隙的情况，此时系统的识别效率为 0，为最坏的识别情况，会导致RFID系统瘫痪。



图4.4 不同数量待识别标签数量下吞吐量与标签数量的关系



图4.5 不同数量待识别标签数量下时隙碰撞率与标签数量的关系

当帧长度与标签数相差较大时，不仅系统识别效率降低，时隙内的碰撞情况也很严重，当待识别标签数与初始帧长度接近时，系统效率达到最高点，时隙内的碰撞概率也可以降到最低点。因此，如果能在每一帧识别结束之后动态调整下一帧的长度使其与待识别标签数目接近，那么就能大大提高成功时隙率。在下一节中，就将介绍帧时隙ALOHA的改进版——动态帧时隙ALOHA。

5、动态帧时隙ALOHA(Dynamic Framed Slotted ALOHA)算法

5.1 动态帧时隙ALOHA算法思想

动态帧时隙ALOHA(Dynamic Framed Slotted ALOHA)算法是通过改变帧时隙的大小提高标签识别效率的。DFSA 算法能部分解决时隙ALOHA算法存在的识

别效率低的问题。动态帧时隙ALOHA 算法和时隙 ALOHA 算法基本相同，只是在时隙数的选择上根据具体情况进行设定。时隙数的选取对算法的效率有很大的影响，在时隙数少的情况下，标签选择同一个时隙发送数据的概率很大，容易产生碰撞。为了避免大量碰撞的发生，需要准备足够大量的时隙，但是降低了防碰撞算法的性能。因为所有时隙段的持续时间与可能存在的标签数有关，对于可能只有一个标签存在的特殊情况，过多的时隙数会造成浪费。弥补的方法是创建动态的时隙，即识别范围内的标签数较多的时候，提供相对多的时隙数，识别范围内的标签数较少的时候提供相对少的时隙数，达到优化算法的目的。

5.2 动态帧时隙ALOHA算法识别过程

1）阅读器在周期循环的开始时隙中发送请求命令，该命令包含了初始的时隙数L。

2)标签接收到阅读器的请求命令后进入识别状态，并在(1，L)中随机选择一个时隙，同时将自己的时隙计数器复位为 1。

3)如果标签的时隙计数器等于标签自己所选择的时隙数时，标签向阅读器发送信息；如果标签的时隙数不等于时隙计数器，那么标签将保留自己的时隙数等待下一个阅读器的命令。

4)阅读器收到的响应与发出的命令可能有以下三种情况：阅读器没有收到标签的响应，说明该时隙是空闲时隙(空闲时隙数记为E)。阅读器结束该时隙并发送下一个时隙命令。同时处于识别状态的标签收到命令后将时隙计数器1，然后转向步骤 3)重新开始执行。阅读器检测到多个标签响应，发生了碰撞(碰撞时隙数记为 C)。阅读器结束该时隙并发送下一个时隙命令。同时处于识别状态的标签收到命令后将时隙计数器加1，然后转向步骤3)重新开始执行。阅读器只收到一个标签响应，并且正确读取标签信息(成功时隙数记为S)。阅读器发送下一时隙命令，同时处于识别状态的标签收到命令后将时隙计数器加 1，已经正确读取的标签进入休眠状态，不再响应阅读器的命令。

5）如果目前的时隙数已经等于该帧的最大时隙数 L，那么结束本次循环，阅读器发送命令转到步骤 1)，开始下一帧的识别。新一帧的长度 L 是阅读器根据上一次循环中的碰撞情况动态调整后确定的。DFSA最关键的问题是如何调整帧长度。

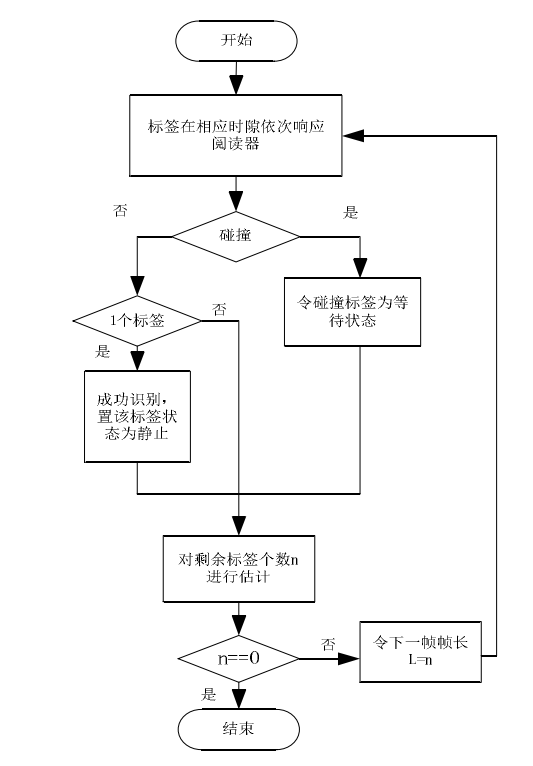


图5.1 DFSA算法识别过程

5.3 动态帧时隙ALOHA算法经典标签估读模型

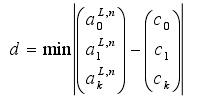
5.3.1 算法思想及仿真

1）Vogt算法

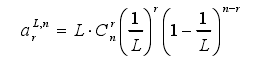
Vogt算法是通过上一帧的标签识别状况来估计剩余的标签个数。首先求得实际中一帧内所有时隙出现的成功识别的标签时隙数、标签碰撞时隙数以及空时隙数，然后计算出一帧内所有时隙出现的成功识别标签时隙数、标签碰撞时隙数以及空时隙数的期望值，将这两组数分别组成向量，通过求得这两组向量的空间距离求出剩余的标签个数，这两组分别为



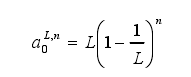
设帧长为L，标签个数为n，向量的空间距离为d。



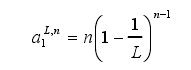
当一帧内有r个标签被读写器成功识别时，则成功识别标签时隙数期望值：



空隙数期望值：



成功识别标签时隙数期望值：



所以标签碰撞时隙数期望值：



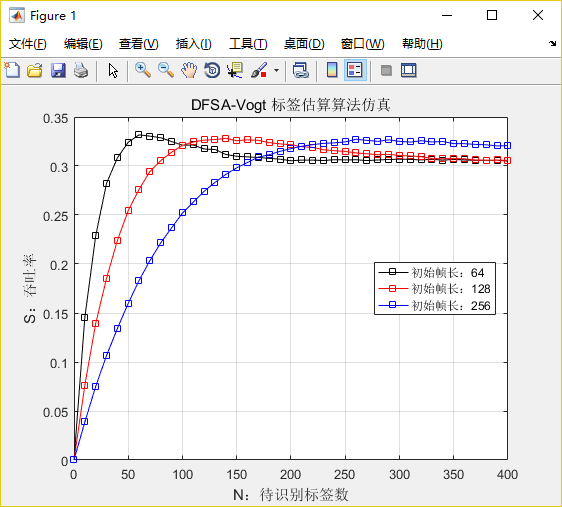


图5.2 DFSA-Vogt算法仿真

2）Lower Bound 估算算法

Lower Bound估算算法仅仅是对一帧内发生碰撞标签个数的下限值进行估算。当标签个数较小时，标签估算的准确率比较高。当标签个数很大时，标签估算值与实际剩余的标签个数相比存在很大的误差。（n = 2\*C(coll)）



图5.3 DFSA-Lower Bound 标签估算算法仿真

3)Schoute 估算算法

n = 2.3922\*C(coll)，仿真n=ceil(2.3922\*C(coll))



图5.4 DFSA-Schoute标签估算算法仿真





图5.5 Schoute 估算算法和Lower Bound 估算算法对比

5.3.2 算法仿真结果对比

Vogt算法无论在估算误差，还是消耗时隙总数上都是最优的。Vogt算法对标签估算相对稳定，但算法比较繁琐，从而会使系统变得复杂。Schoute算法能使系统达到最大吞吐率，但在标签个数不是足够大的情况下，估算误差比Vogt算法大。Lower Bound算法标签个数较小时，标签估算的准确率比较高。当标签个数很大时，标签估算值与实际剩余的标签个数相比存在很大的误差。碰撞率算法在帧长足够大的情况下，标签估算相对比较准确。帧长比较小时，标签估算结果误差较大。所以应该按照实际标签个数及其他因素正确选择合适的估算算法。

6、参考文献

[1] 煤矿安全预警系统中RFID防碰撞算法的研究\_赵跃

[2] 基于动态帧时隙ALOHA的标签防碰撞算法研究\_张晶

[3] 基于ALOHA的RFID系统防碰撞算法的研究与设计\_孟婕

7、思考和改进

7.1 关于纯ALOHA所说的吞吐量的计算，由于满足泊松分布，在一篇论文中在仿真的时候，就是对直接采用了泊松发生器，如何采用，如何实现，不得而知。