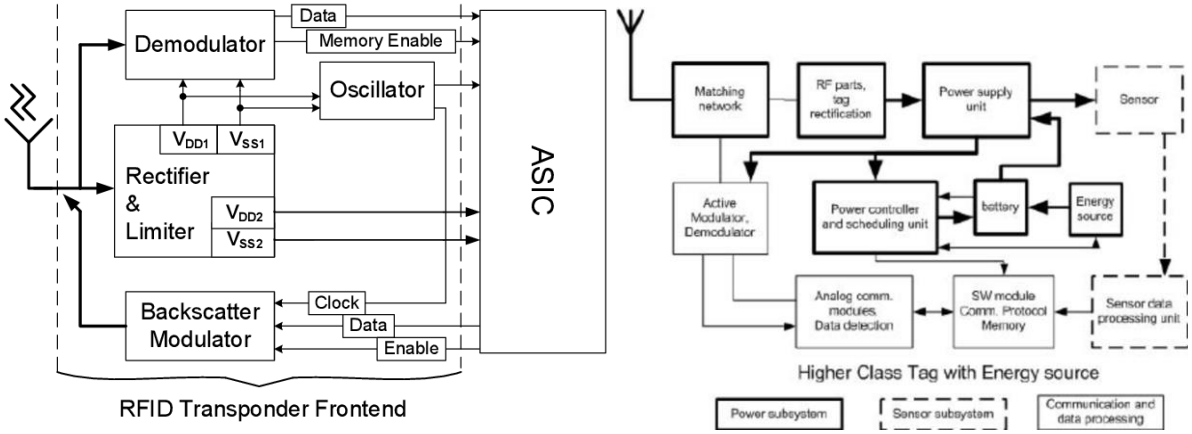
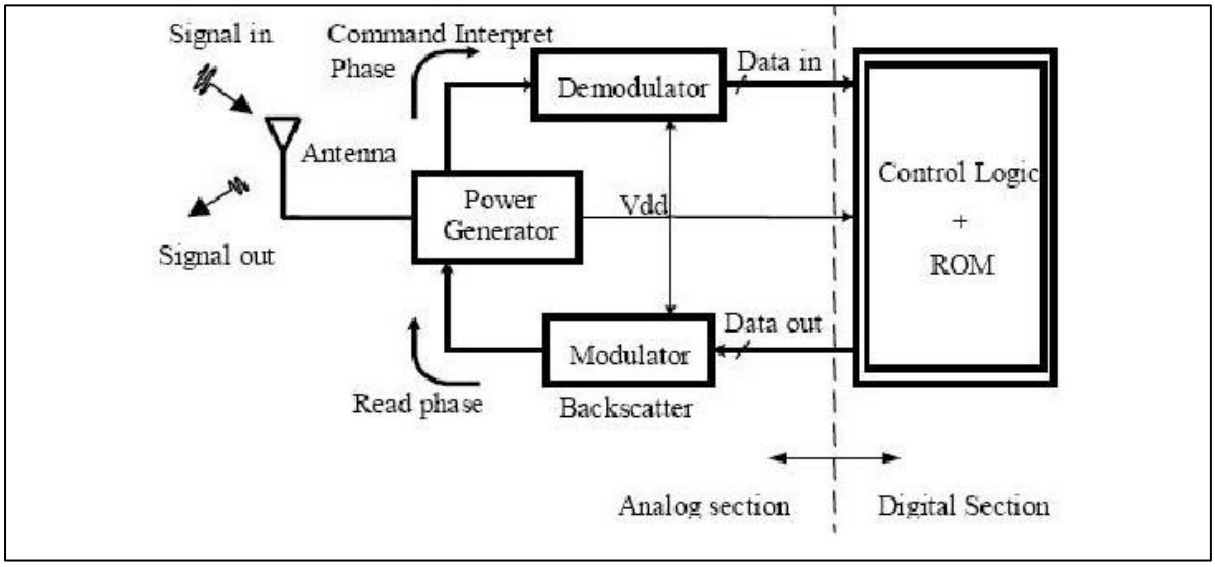
Internet of Everything (IoE)

RFID APPLICATIONS

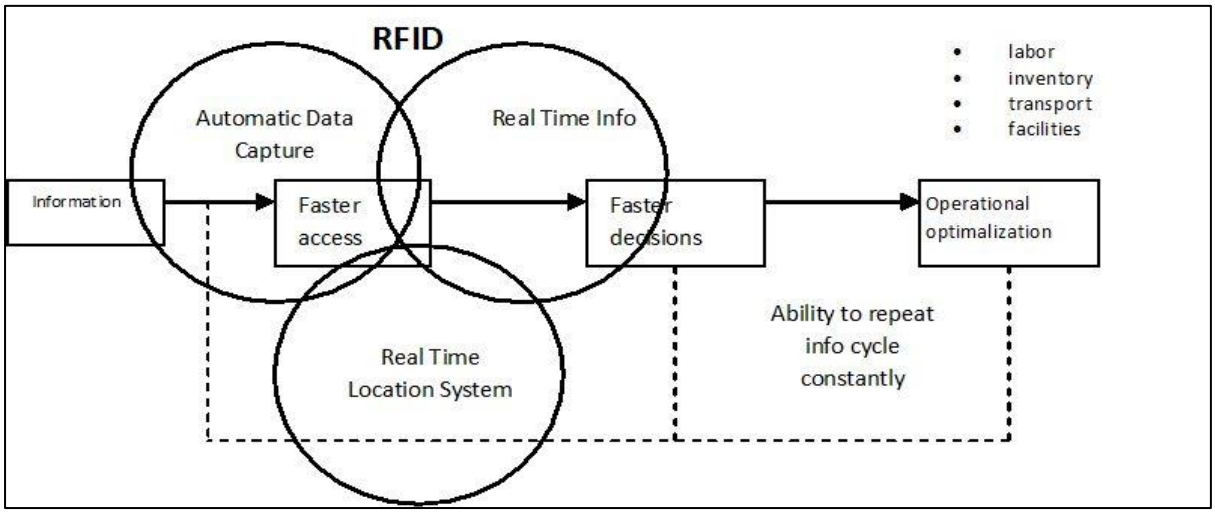
RFID Transponder



RFID Architecture:



RFID-enabling technology:



Types of ALOHA:

1. **Pure ALOHA:** A tag itself decides the data transmission time randomly as soon as it is activated. The transmission time is not synchronized with both the reader and the other tags at all. When the electricity is charged by the reader's electromagnetic wave tags transmit data after receiving the *REQUEST* command from the reader. If multiple tags transmit data imminently (whether earlier or later) then a complete or partial collision occurs. Retransmitting after random delay is the solution for a collision. During the read cycle the reader receives the data and identifies tags sent data without collision. When a read cycle is done then the reader broadcasts the *SELECT* command with the tag's unique identifier received from the tag. Once tags are selected the tags stop responding for the request command i.e. the selected tags keep silence until whether they receive other commands e.g. authenticate, read and write or the tag's power is off by being located out of the reader's power range. When the tag is reentered into the reader's interrogation range it restarts transmitting its data to the reader. The advantage of this algorithm is simplicity.

2. **Slotted ALOHA:** It is obtained by the addition of a constraint to the (Pure) ALOHA. The read cycle is divided into discrete time intervals called *slot* and which is synchronized with the entire tags by the reader. Thus, tags must choose one of the slots randomly and transmit data within a single slot. Transmission begins right after a slot delimiter. This causes that packets either collide completely or don't collide at all i.e. there is no partial collision in the Slotted ALOHA algorithm. This reduces wasting the read cycle relatively as compared with the (Pure) ALOHA algorithm. However, the empty slot can be occurred in the read cycle and the disadvantage is that it requires a synchronization mechanism in order for the slot-begin to occur simultaneously at all tags.

3. **Framed slotted ALOHA:** Framed Slotted ALOHA algorithm uses the frame which is the discrete time interval of the read cycle and each frame is divided into the same number of slots. There are multiple frames in a single read cycle and the frame size is decided by the reader. There is a constraint that the tags can transmit data only once in each frame. It may reduce the number of collided slots and it shows the best performance among them.
   FSA (Framed Slotted Aloha) can be classified into the BFSA (Basic Framed Slotted Aloha) and the DFSA (Dynamic Framed Slotted Aloha) according to whether which uses fixed frame size or variable frame size [Klair04]. If the number of actual tags is unknown DFSA can identify tags efficiently rather than BFSA by changing frame size since BFSA uses fixed frame size. In addition, BFSA and DFSA can be further classified based on whether they support muting or/and early-end features [Klair04]. The muting makes tags remain silent after being identified by the reader while the early-end allows a reader close an idle slot early when no response is detected.
   a. BFSA:
      i. BFSA-Non-Muting
      ii. BFSA-Muting
      iii. BFSA-Non muting-early-end
      iv. BFSA-Muting-early-end
   b. DFSA:
      i. DFSA-Non-Muting
      ii. DFSA-Muting
      iii. DFSA-Non muting-early-end
      iv. DFSA-Muting-early-end

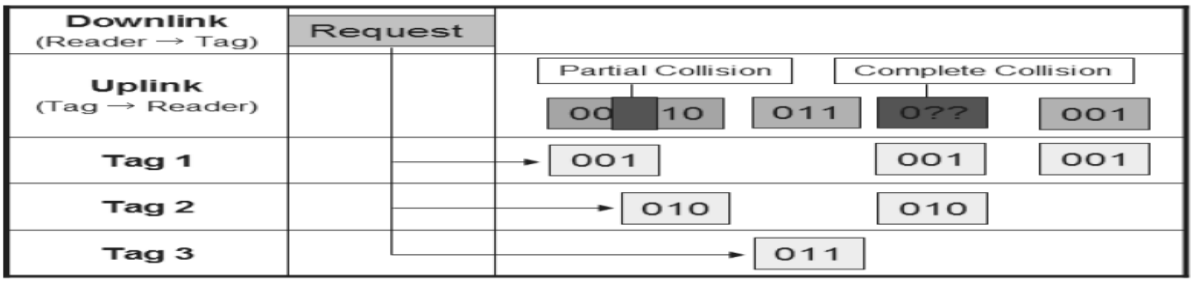


Fig: Pure ALOHA

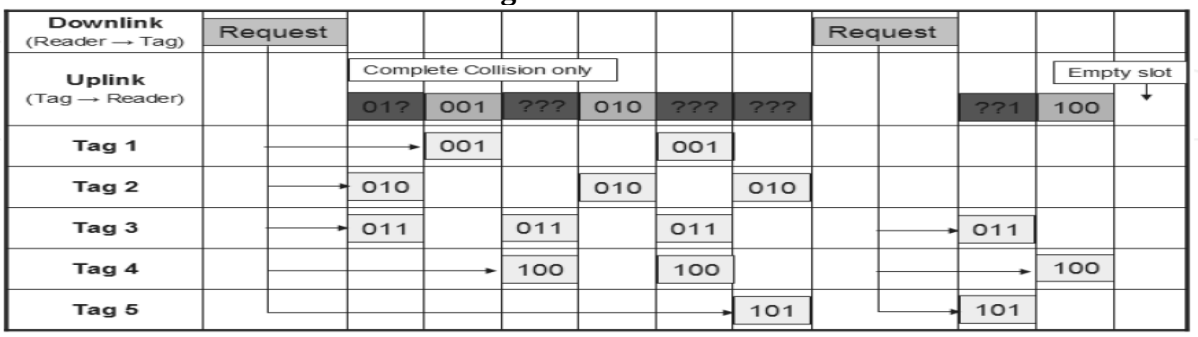


Fig: Slotted ALOHA

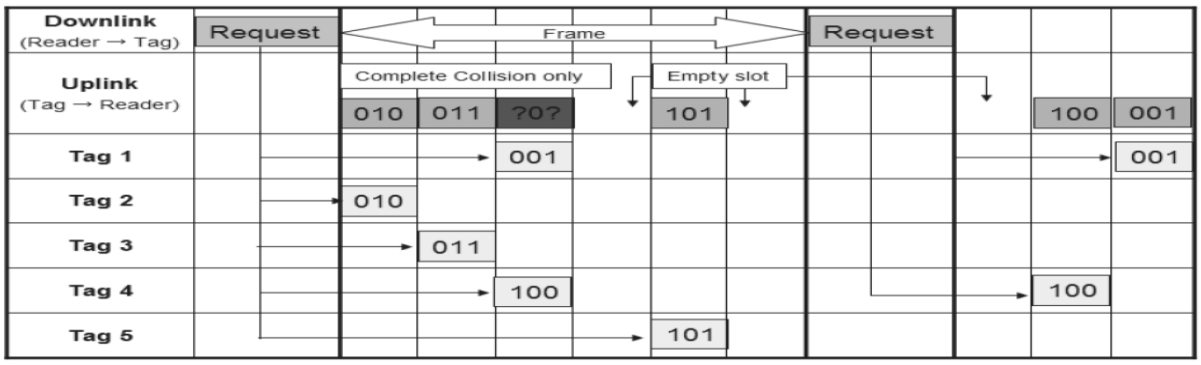


Fig: Frame Slotted ALOHA

Multi-State Query Tree Protocol:

To identify tags, we suggest multiple state query tree protocol, which is variation of query tree protocol. The query tree algorithm consists of rounds of queries and response. In each round the reader asks the tags whether and of their IDs contains a certain prefix. If more than one tag answer, then the reader knows that there are at least two tags having that prefix. The reader then appends symbol 1, 2, … or 20 to the prefix, and continue to query for longer prefix. When a prefix matches a tag uniquely, that tag can be identified. Therefore, by extending the prefixes until only one tag's ID matches, the algorithm can discover all the tags.

In the query tree protocol, a reader detects collision bit by bit. But in our scheme can detect collision with 16-bit vector symbols which have twenty symbols. And all tags which are matched the prefix, transmit their remained bits in query tree protocol, but in multiple states query tree protocol, they transmit their next one symbol which is 16 bits. The following describes the protocol:

```
Set the prefix empty
Begin until
rx-signal = request (with the prefix)
If (rx-signal is no response ) then
If (the prefix is not empty) then
delete last symbol in the prefix
```

```
Else
no response with empty prefix
Endif
Else
Symbol = decode (the rx-signal)
add symbol in to end of the prefix
Endif
If (size of prefix == size of tags symbol) then
ensure that existence of the tag and
make it not response
delete last symbol in the prefix
Endif
Until (there are no response with empty prefix)
```

Suppose that the RFID system use 48 bits for IDs, which consist of three symbols and supports 8000 tags. Each tag has unique path in the query tree and its depth is 3. Therefore, we can identify one tag at most 3 times transmission. When a reader request next symbol with prefix, the tags transmit their next 16-bit symbols and the prefix matches with one tag's all symbol, the tag must send conform message. For example, there 4 tags whose ID are [4 18 5], [4, 18, 7], [8, 9, 2], [6 8 3] in the reader, the reader's requests command bellows:

iteration	Reader request	Tags response	Memo
1	null	[4]	
2	[4]	[18]	
3	[4 18]	[5]	Identified
4	[4 18]	[7]	Identified
5	[4 18]	null	
6	[4]	Null	
7	null	[8]	
8	[8]	[9]	
9	[8 9]	[2]	Identified
10	null	[6]	
11	[6]	[8]	
12	[6 8]	[3]	Identified
13	[6 8]	Null	
14	[6]	Null	
15	null	Null	

To support 8000 tags, the other protocol needs 13 bits (8192 tags) and 13 iterations to identify one tag in worst case but our scheme needs only 3 iterations in worst case.

Tree Protocols:

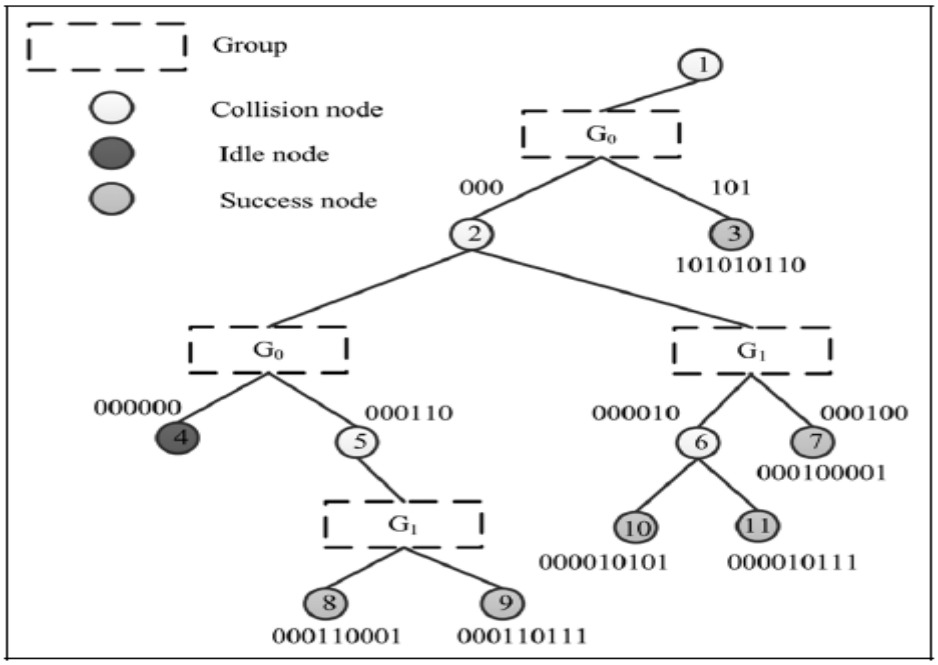
REQUEST(e): a request command. All tags, existing within the scope of a reader, automatically respond and send their IDs to the reader.

REQUEST(PRE): a request command. Within the query process, the value of PRE has to be updated.

Here, PRE is a query prefix. Tags respond if their prefixes are the same as PRE after receiving the command. And then they compute GC. When RFID reader broadcasts REQUEST(PRE), the responded tags transmit their GCs and remaining ID except PRE to the reader.

REQUEST(PRE,G): a request command. Here, PRE is also a query prefix and G denotes group number, 0 or 1. When an RFID reader broadcasts REQUEST(PRE,G), the tags of which IDs' prefixes are the same as PRE respond and are divided into two groups according to definition

1, grouping rule based on characteristic value. And then tags in G send their remaining IDs except PRE to the reader.
PUSH(PRE): a read/write command. PRE, as a new prefix, is pushed to the bottom of the tack.
POP(PRE): PRE pulls from the top of the stack. SELECT(ID): a select command to select a tag with the same ID.
READDATA: a read command to read tag IDs.
SILENCE: a silent command. The tag not responds to the newly queued queries in the tag identification process.
'||' represents a concatenation operation, such as 01 || 11 means 0111.
Example of BAT:
It is assumed that there are six tags: tag 1 (000110111), tag 2 (000110001), tag 3 (000010111), tag 4 (0001 00001), tag 5 (000010101) and tag 6 (101010110). The details of the identification procedure with BAT are shown in Table. Fig. shows the query tree structure using BAT.



Slot	Request(PRE)	Tag respond	Bit string	Stack
1	ε	Tags 1, 2, 3, 4, 5, 6	Collision, X0X, RGC = 0	000, 101
2	000	Tags 1, 2, 3, 4, 5	Collision, XX0, RGC = X	101, 000000, 000110, 000010, 000100
3	101	Tag 6	Identify	000000, 000110, 000010, 000100
4	000000		Idle	000110, 000010, 000100
5	000110	Tags 1, 2	Collision, XX1, RGC = 1	000010, 000100, 000110001, 000110111
6	000010	Tags 3, 5	Collision, X1	000100, 000110001, 000110111, 000010101, 000010111
7	000100	Tag 4	Identify	000110001, 000110111, 000010101, 000010111
8	000110001	Tag 2	Identify	000110111, 000010101, 000010111
9	000110111	Tag 1	Identify	000010101, 000010111
10	000010101	Tag 5	Identify	000010111
11	000010111	Tag 3	Identify	Empty