CHAPTER 5
POWER MANAGEMENT AT NETWORK INTERFACE LEVEL

The key mechanism to reduce the power consumption of wireless interfaces is to put the interface into sleep mode whenever possible. In the context of games, the challenge is to do this without affecting the game play. As described in Section 5.1, the games become not playable if the round trip network latency increases beyond acceptable levels. The acceptable level varies according to game genre and type - up to 180ms for high speed FPS games and up to 400ms for MMOG games. Therefore, the basic decision made by our system is: *When and for how long can the wireless interface be put into sleep mode without affecting game play?*

An obvious answer would be to sleep when there is minimum game state change and/or during unimportant, from the player’s perspective, game events. However, in order to find these situations, our system needs to gather sufficient information to estimate the current game state and then decide on the appropriate action. Non-critical game moment is defined as the duration in game that the player has no interaction with the other players. During non-critical game moment, the other players’ information is trivial to the player (*current player*) thus we can stop updating those information for the duration. We have game state estimation algorithms run at server-side obtaining necessary information from server. Server is responsible for running state estimation algorithm and sending *sleep* command to clients. Clients
need minimum changes just to listen to *sleep* command and run a procedure to put the wireless interface to sleep mode.

In this chapter, first we introduce mobile games and game maps in Section 5.1 that forms the background knowledge to understand our approaches for power saving. Then, we describe three different approaches (in Sections 5.2, 5.3 and 5.4) to determine non-critical game state and execute power saving operations. Each approach has its own pros and cons. We describe a scheme to select the algorithms according to the game map and game genre.

### 5.1 Mobile Games and Game Maps

In this work, our objective is to create power management algorithms for some popular and challenging game genres. We focus on FPS (First Person Shooting) games and MMOGs (Massively Multiplayer Online Games). FPS games usually have small, highly occluded and complex game maps, and requires swift user actions during the game play. In spite of high processing loads for rendering the complex graphics, these games should respond to user actions instantaneously. In modern networked games, the game server maintains the game state and does most of the processing for cheat prevention. Client actions are transferred to the server where the decision about the client’s new game state is made and transferred back to the client. For instantaneous response, the network latency should be very low. Though it is very specific to game design and game mechanics, our observations with Quake III and the results reported in the literature [43] show that round trip network delays beyond 180ms makes the game not playable for twitch games such as FPS. Any power saving
effort should seriously consider this delay requirements. MMOGs usually have huge open maps with less occlusions. These games typically have several 1000s of players playing concurrently hence load balancing and bandwidth management are the key challenges here. These games are usually slow speed games and our observations with Ryzom shows that players can tolerate upto 400ms round trip delay.

5.1.1 Binary Space Partitioning

Binary Space Partitioning (BSP) is one of the common spatial subdivision schemes used to represent game maps (eg. Quake III, Unreal Engine based games, Half Life 2, Call of Duty, Medal of Honor...). Though it needs more memory and pre-processing,
if clearly represented it is a good option for highly occluded static maps. As it is generated during pre-processing phase, it provides high run-time efficiency. BSP is a method for partitioning map such that each small part can be easily managed. It builds a tree structure that uses a polygon plane as node to divide the space into two subspaces. One is in front of the splitting plane, while the other is behind. Each of the 2 nodes is applied with same process recursively until a space that contains no polygon. This final space is a leaf of BSP tree. The leaf is a *convex hull* that no point exists inside. It can be an object enclosed with polygons or just an open space with partial or no faces. All the leaves add up to form the whole space. A detailed description of BSP is available in [137].

5.1.1.1 Potentially Visible Set

With BSP tree, the world is split into small convex hulls. Given the view point, we can easily locate the convex hull that contains it in BSP tree. From the convex hull, there is in fact a limited number of other convex hulls can be seen. Thus, if the set of possibly visible convex hull is known, the renderer can just render those convex hulls in certain order, be it from back to front for translucent objects, or from front to back for opaque objects. In such a case, the overhead would be the overlap of closest visible convex hull and the second closest, plus the convex hulls at behind view instead of many order higher number of polygons. As shown in Figure 5.1, the renderer (which is at players eye position) can see through the wall. The staircase behind the wall is thus drawn. But the view is only limited to the staircase. Nothing behind staircase is drawn.
Therefore, we define the *Potentially Visible Set (PVS)* as follows. For a given source geometry, PVS is the set of world geometries that can be fully or partially seen from the source geometry. In fact, FPS games (specifically, games based on Quake III engine) has extended the geometry to a more generally term, *Area*. PVS can be efficiently computed using the BSP tree as described in [138] and a more advanced version, which progressively computes PVS is described in [139]. As PVS is static information for a given map, it can be precomputed and stored in the same BSP tree which represents the map.

**Area, Cluster and Portal.** Figure 5.2 demonstrates the use of BSP and PVS in a level. The level is subdivided into many spaces, called *areas*. If an area is enclosed by faces, the area is marked solid. A solid area will never be considered during rendering because a player will never enter or see that area. There are 3 players in the level currently. The PVS set of an area is not affected by the position of the
player it contains. Area 3 can see area 21, but cannot see area 14. Thus, player 1 and 2 are potentially visible to each other. Player 3 is potentially visible to none of others.

Adjacent areas shared same surfaces are grouped together to form a cluster [140]. A cluster is something like a room. Inside the room, there is an area under table, an area under bed or an area that can walk on. But all these areas are inside a room. A room is separated from another room by a door. A door is called cluster portal, because player can travel from a cluster to another through it. Thus, a map is logically classified into clusters connected by cluster portals. This graph structure together with areas forms a hierarchical network.
Table 5.1. Map Size of MMOG games

<table>
<thead>
<tr>
<th>Game</th>
<th>Typical Map Size in miles square</th>
</tr>
</thead>
<tbody>
<tr>
<td>World of Warcraft</td>
<td>80</td>
</tr>
<tr>
<td>Ryzom</td>
<td>50</td>
</tr>
<tr>
<td>Asheron’s Call</td>
<td>500</td>
</tr>
<tr>
<td>Guild Wars: Nightfall</td>
<td>15,000</td>
</tr>
<tr>
<td>The Lord of the Rings Online</td>
<td>30,000</td>
</tr>
</tbody>
</table>

5.1.1.2 Limitations

PVS is computationally intensive and requires precomputation. Some modern games let the artist specify the cluster portals, known as artist defined portals to compute visibility. For games with dynamic maps, BSP and PVS cannot be applied easily. For large open space maps BSP is an overkill for memory and processing. These games use other representations such as octtree described in next section.

5.1.2 Quadtree and Octtree

Games which use huge open maps split the game world (maps) into cells and represent the maps usually in quadtree (for 2D maps), octtree (for 3D maps, cells are cubic), r-tree and their variants. MMOG games usually have such huge open maps. More details about octtree can be found in the Manuals of OGRE [141] and cube2 engines [142]. Map size of some of the popular MMOGs are given in the Table 5.1. Though the source needs to be verified, the Figure 5.3 from [143] gives some idea about the typical map sizes in several modern MMOG games.

Depending on the range the player is viewing determines how deep down the tree to be traversed for rendering. Each cell could then contain other data structures, eg. a large castle or dungeon could have a BSP for the inside. In computer graphics,
Figure 5.3. MMOG Map Sizes - A Graphical Comparison
accounting for *Level of Detail* (LoD) involves decreasing the complexity of a 3D object representation as it moves away from the viewer, or according to other metrics such as object importance, eye-space speed or position. LoD techniques increase the efficiency of rendering by decreasing the workload of vertex transformation which is one of the stages in graphics processing pipeline. The reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or moving fast. The Figure 5.4 shows the the circular regions for LoD, the darker areas are rendered with full details and the lighter ones have less details. MMOG games define *vision range* or *vision distance*, which is the distance upto which the player can view objects in detail. Beyond this range things get blur. Vision range is one of the key parameter in implementing LoD in these games. Within the vision range frustum culling is applied to further improve the efficiency of rendering.
5.2 Distance Based Approach

5.2.1 Game State Estimation

In distance based algorithm, we discretise the game world into tiles as depicted in Figure 5.5 [144,145]. We define *vision range* which is, the distance up to which a player can view the objects with sufficient details to interact with them. In some games, especially MMOG games with huge open maps *vision range* has already been defined as a part of the LoD implementation. Our algorithm just uses this value if it is already defined. In most games, vision range (also called as vision distance or vision radius or Area of Interest (AoI)) is dynamically adjusted based on the player’s environment. For example, if the player is in safe zone (like a sanctuary area), there won’t be any enemy object and, the player can interact only with the friendly objects. The interactions (such as, chatting, weapon exchange) with friendly objects happen mostly when they are in close proximity to the player. But, the interaction(such as, shooting) with enemies in hostile environments may happen even when their distance is longer. Vision range is set to a larger value when the player is in hostile environment when compared to friendly environment. The actual range is game dependent. In Figure 5.5, the visibility range of the client ‘a’ could be r1 or r2 depending on its current environment. In this report we consider r1=r2.

For each client in power save mode (mobile client), first the server checks for any entity within the vision range of the client. If there is an entity, then the *game state* for the client is marked as *critical* otherwise, *non-critical*. To make such checks faster and scalable, the entities are continuously registered to the tiles/cells which they are visiting during runtime. To check for the entities in the vision range, we just need to
check the whether there is any registered entity in each of the tiles within the vision range. We have two levels (Macro and Micro) of power management. The pseudo code presented in Algorithm 1 outlines our main algorithm.

**Algorithm 1** Main (Distance Based Approach)

```plaintext
for each server frame do
    for each active client do
        if client is sleeping then
            continue
        else
            do clientstate = Client.state()
            if clientstate = 'notCritical' then
                do Macro.power()
            else
                do Micro.power()
            end if
        end if
    end for
end for
```

If the game state is not critical the system enters *Macro Level* (gives potentially longer sleep duration) power management; Otherwise, it enters *Micro Level* (sleep duration is limited) power management. Macro and Micro level power management algorithms predict the duration for which the client’s wireless interface can be put into sleep mode and sends sleep command to client with duration. On receiving the sleep command with sleep duration the client puts its wireless interface into sleep mode for the specified duration and automatically wakes-up at the end of sleep duration. Macro and Micro level prediction algorithms are described in the following sections.

### 5.2.2 Macro Power Management

At the macro level, the server makes power management decision and sends *sleep* command to the client. Its decision relies on the position of all players. Depending
on the game genre number of players in the game, it can use either single-ring (games
with low number of players, such as FPS games) or dual-ring algorithms (games with
huge number of players, such as MMOGs) given below.

**Single Ring Algorithm (SRA).** Single ring algorithm is based on the *relative
velocity* between the players (interactive entities in general). Single ring algorithm
estimates the *Potential Sleep Duration* (PSD) as presented in *Algorithm 2*.

**Algorithm 2** Single Ring Algorithm (SRA)

```
for each entity i do
    currentProximity_i = getEuclideanDistance(currentClient, entity_i)
    \(\triangleright\) get current proximity of all interactive entities

    pastProximity_i.add(currntProximity_i)
    \(\triangleright\) adds current value to history and removes oldest value

end for

interestingEntities_{1..n} = getNearest(n entities)
\(\triangleright\) get n nearest entities; we are interested only on n nearest entities

for each interestingEntity j do
    relativeVelocity_j = calculateRelativeVelocity(pastProximity)
    \(\triangleright\) compare the historical proximity to determine new relative velocity (*bi_directional*).
entities coming closer or going away?

    PSD = (currentProximity_j - AoI Visibility Radius)/relativeVelocity_j;
    \(\triangleright\) calculate potential sleep time. That is, s1 or s2 in Figure 5.5

    if PSD <= 0 then
        do return (0)
        \(\triangleright\) If PSD <= 0, return 0 and exit SRA. Entities found inside the ring
    end if
end for

return smallest(PSD_{1..n})
\(\triangleright\) return the PSD of the entity which is expected to reach the client’s AoI Visibility
Radius first
```
The algorithm finds the nearest \( n \) interactive entities and estimates the time required for them to reach the current client’s vision range. The smallest these reach-time values is set as the PSD. The PSD is the safest duration and there is very less chance for important game state changes during this period.

Computing \textit{Euclidean} distance requires a square root operation, which even on modern computers is expensive. As our computation is concerned only with comparing distances, comparing square of \textit{Euclidean} distances is equivalent comparing the distances. Hence, we compute square of \textit{Euclidean} distances to all other players using the formula below and pick the \( n \) nearest players. For these \( n \) nearest players we compute the actual distance. \( \text{Distance}^2 = (X^2 + Y^2 + Z^2) \) - for three dimension; \( \text{Distance}^2 = (X^2 + Y^2) \) - for two dimension; where, \( X,Y \) and \( Z \) are \( \text{abs}(x_1 - x_2), \text{abs}(y_1 - y_2), \text{abs}(z_1 - z_2) \); \((x_1, y_1, z_1)\) and \((x_2, y_2, z_2)\) define the position of the client and an interactive entity.

For a game with \( m \) interactive entities, the algorithm takes \( O(m^2) \) time to find the nearest \( n \) entities. In MMOGs such as WoW (World of Warcraft) \( m \) is high. On average, in WoW there are 100 to 900 active players throughout the day across all realms (all servers) with an average of 500 players [146]. On “Earthen Ring - RP” realm alone it ranges from 100 to 1100 active players throughout the day. If \( m \) is big, to make the algorithm scalable, Dual Ring algorithm is used. Dual ring algorithm is described below.

\textbf{Dual Ring Algorithm (DRA).} Dual ring algorithm is based on \textit{incremental scanning} mechanism. It starts with vision range of the player and gradually increases
the search area for other entities. It checks for entities in the area from vision range of the player to distance $s$ as shown in Figure 5.6a. Where $s$ is 100ms time step from the vision range of the player (p1) and it is computed as,

$$s = visionRange + (2 \times MS \times 100ms)$$

where $MS$ (Movement Speed) is estimated average player Movement Speed. $2 \times MS$ approximates the relative speed of two players. It gives the worst case value where two players move towards each other.

To compute $MS$, the server logs the position of each player for a charted period and computes the maximum speed of each player. An average of all these player speeds is computed to update $MS$. We currently did this offline in our implementations, which can be integrated to run-time algorithm and run to keep $MS$ accurate.
If there is no entity in the range $s$, then the algorithm increases $s$ by another 100ms time step and checks again. This is repeated until an entity is found or max sleep threshold (game dependent parameter [111]) is reached. PSD is set to either the value of $s$ prior to finding the first entity or max sleep threshold. We have selected 100ms time-step as the smallest possible value for $s$ as sleep duration below 100 ms are too small to save any significant energy saving for 802.11b/g interface [111].

5.2.3 Micro Power Management

If game state is critical, the main algorithm leads to Micro Level (sleep duration is limited) power management. At micro level the algorithm relies on orientation of the players. Player orientation is suitable for game with maps having huge open areas (eg. common maps of MMORPG games).

**Player Orientation Based Approach.** In most games the orientation of a player is defined by a float value (range from $-\pi$ to $\pi$) and the field of view of the player is $2\pi/3$. The entities which are inside the vision range of a player still cannot be seen if they are not in the field of view of the player as shown in Figure 5.6b. Micro power management exploits this property to improve the efficiency of our system. We computes the duration required for the player to reach other players' field of view as given below.

1. We calculate the angle $\phi_1$ between the directions vector of the player and the vector from the payer to entity (n and BA vector in Figure 5.6c). From which we compute, $(\phi_1 - \pi/3)$ which is the angle the player need to turn to see this entity.

166
2. Similarly we calculate the angle $\phi_2$, which the entity needs to turn to see the player. We calculate $\phi_2$ only if the entity has vision. The $\min(\phi_1, \phi_2)$ will be the angle displacement between the character and the entity.

3. After calculating the angle between the character and all entities inside vision range or distance, we can find the minimum angular distance, $\phi_{min}$.

4. Based on this value, we find the $Potential_{Sleep\_Duration}$ PSD as follows.

\[ PSD = \frac{\phi_{min}}{(2 \times \alpha)} \]

where, $\alpha$ is the angular velocity of players. $2 \times \alpha$ approximates (worst case) relative angle velocity of 2 players.

$\alpha$ is a game dependent parameter. We have computed this offline in our implementations. We turned the player $360^\circ$, several times by continuously pressing the right arrow (right turn/rotation key) and estimated the time taken to rotate $360^\circ$. From this value we have obtained the value for $\alpha$.

**Sleep Command.** The server sends sleep command with the computed PSD (that is, duration to sleep) to the client. The client side logic is described below in Section 5.5. In both macro and micro power management cases we ensure that the client has received at least one complete update before the next sleep to avoid longer inconsistencies.
a. Dual Ring Algorithm

b. Field of View

c. Angular Distance

Figure 5.6. Distance Based Approach
5.3 Visibility Based Approach

Distance based approach works well with MMOG games where the maps are open and huge. For FPS games we propose visibility based approach [147] as the game world is usually small and closed with lots of occlusions. If two players do not see each other, they probably wont have any interaction. Hence, these two players need not be updated about each other. Conceptually, our approach does the following:

- For each player, it identifies all the possible areas the player can be in $\beta$ ms if they are moving at the maximum speed.

- For all pairs of players, check if two players can be visible to each other in $\beta$ ms.

- For each player, if they are not visible to all other players in $\beta$ ms, then a sleep interval of $\beta$ ms is a "safe" interval for this player to sleep.

![Inter-player Visibility With Obstacles](image)

**Figure 5.7.** Inter-player Visibility With Obstacles
Figure 5.7 shows an example on how visibility varies with the players’ positions over time. The shaded areas around each player’s number indicate the possible areas that a player can move to in $\beta$ ms while the black areas indicate obstacles that obstruct visibility. From the figure, we see Players one and two will not be visible to each other even after $\beta$ ms — no matter how either player moves. Hence, without considering player three, both players can safely sleep for $\beta$ ms. However, when player three is taken into consideration, only player one is not visible to both players two and three. As a result, only player one can sleep. Players two and three cannot sleep as there is a possibility that they could enter each other’s AoV. Hence, we conservatively avoid sleeping players two and three to ensure minimal quality loss.

To apply this in practice, we need to consider every point in the game’s current map. We do this by first discretising the game map into small regions. It then becomes theoretically possible to check every map region to see if any player can possibly see any other people. However, this brute force approach is impractical for any reasonably large map.

Our approximation-algorithm solution reduces the problem size in both space and time. For time, we only consider time quantum of $\beta$ ms. For space, we divide the game space into hexagonal or square grids, where the grid element width ($p$) is the maximum distance a full-speed player can cover in $\beta$ ms. In the rest of this section, we use a 2D hexagonal grid, even though our approach works for 3D maps as well.
Using the grid defined above, we identify the possible future locations in multiple time steps (of β ms). Figure 5.8 shows the possible locations from the current player’s position of zero (centre of figure) that can be reached up to three β time-steps later. The darkest shading shows the area reachable in one time-step, followed by two steps (lighter shading), and then three steps (lightest shading). Note: the player could stay still and not move in each time-step.

Assume that there are a total of m grid elements in the map. To reduce computation, we consider only one particular point in each grid. In this report, we chose the centre of the grid as the reference point. Since a player can be anywhere in the grid, the central location is a reasonable estimate. This approximation of checking only a single point in a grid obviously introduces error but is necessary for the approach
to be practical. If the grid size is sufficiently small, the error will be small as well. However, the price for a smaller grid size is a larger matrix (explained below).

The visibility computation can then be performed off-line as the map layout (where the walls are etc.) and grid representation is mostly static. The visibility computation generates a visibility matrix $V$ of size $m^2$. The matrix entry is '1' if two grids (in particular the centre positions of those two grids) are visible to each other, and '0' otherwise. For dynamically layed-out maps, we can pre-compute matrixes for the different layouts and use the appropriate matrix at runtime.

5.3.1 Dynamic Lookahead

![Figure 5.9. Basic Grid Weights (One Time-Step)](image)

To effectively use these grids, we need to also incorporate the player’s movement patterns. In particular, we need to assign weights to all the likely positions that a player could be at $\beta$ ms later. We can then use these weights to determine the probability of the player being able to see other players located at specific grids.
In our approach, we exploit the player’s velocity to set the grid weights. This is a reasonable assumption as velocity is often part of the game state. To illustrate how we incorporate velocity, Figure 5.9 shows the case where the player is moving up from the centre. Let $w_i$ be the weight of moving to grid $i$ from the current position. The grid weight for moving in the forward direction is four. The forward plus sideways directions have weights of two, and the rest of the directions have their weights set to one. We do not claim that our chosen weights are optimal, only that velocity information can be utilised to improve prediction. We defer the choosing of optimal weights to future work.

These weights allow us to more accurately determine if a player will see other neighbouring players. In the “likely to go north” case, the player is far more likely to see players located in the higher grids and less likely to see players located in the lower grids. In our implementation, we pre-compute these velocity weights and apply them at runtime according to the player’s actual velocity.
The weights shown above are for just one time-step. As mentioned earlier, we need a small grid size to reduce the grid-to-grid visibility check inaccuracy. However, small grid sizes only allow small sleep intervals (as players can move through grids quite fast) — limiting the overall power savings. To overcome this and increase the energy savings, we need to extend our weights to consider multiple $\beta$ time intervals, as shown in Figure 5.8 (for up to 3 $\beta$).

This problem is similar to how mobility can be predicted in cellular networks [148]. In our approach, we assume that, at each future time step, the player’s movement patterns will use the same weights as the one time step weighted grids. Figure 5.10 shows an example of a possible weighted grids for three times steps and for a player moving forward.
5.3.2 Determining Sleep Times and Intervals

We now present our final algorithm that determines when and for how long to sleep a player’s network interface. Consider two players '1' and '2' at grids $x_1$ and $x_2$ respectively. In $t$ time steps, let the grids reachable for players 1 and 2 be $L^1_t$ and $L^2_t$ respectively.

For each point $l_1 \in L^1_t$, check its visibility to each point in $L^2_t$ (using the visibility grid $V$). Set the value $v(l_1, l_2)$ to 1 if their positions are visible to each other and 0 otherwise. For each pair $(l_1, l_2)$, the cost is computed as $w_{l_1} \ast w_{l_2} \ast v(l_1, l_2)$. We compute the likelihood of player 1 seeing player 2 as the following normalised sum:

$$S_t(1, 2) = \frac{\sum_{l_1 \in L^1_t, l_2 \in L^2_t} w_{l_1} \ast w_{l_2} \ast v(l_1, l_2)}{\sum_{l_1 \in L^1_t, l_2 \in L^2_t} w_{l_1} \ast w_{l_2}}$$

(5.3.1)

Let $N$ be the set of all players, $T$ be the set of all discrete sleep time intervals considered and $\alpha$ be a control parameter. The parameter $\alpha$, $0 \leq \alpha \leq 1$, controls the aggressiveness of the power saving mechanism, with 0 being the most conservative and 1 the most aggressive (always sleep). By varying $\alpha$, we can obtain different trade-offs in power consumption versus accuracy. Dynamic lookahead method is outlined in Algorithm 3.

This algorithm is executed every $\beta$ ms on the server, but is transmitted to the client only when the client is awake. To make the algorithm scalable, distance based filtering can be applied to reduce the number of players to check for visibility. However, for
Algorithm 3 Dynamic Lookahead

for each player $i$ do
  for each time slot $t \in T$ do
    for each player $j \in N \setminus i$ do
      compute $S_t(i,j)$
    end for
    $t$ is feasible if $S_t(i,j) \leq \alpha, \forall j$
  end for
  Select the largest feasible $t$ as user $i$'s sleeping interval, otherwise user $i$ does not sleep.
end for

huge open maps the algorithm demands more memory to store cell-to-cell visibility matrix. In addition, computing visibility matrix immediately after loading the map introduces significant amount of delay to the start-up phase of the game.

5.4 3D Renderer’s View Based Approach

As described in Section 5.1, most 3D games use 3D spatial subdivision of the game map (such as, Binary Space Partitioning, Octree) to accelerate the rendering of 3D environments and for collision detection among other uses. Potentially Visible Sets (PVS) is a form of occlusion culling used by renderers, whereby a candidate set of potentially visible polygons are pre-computed, then indexed at run-time in order to quickly obtain an estimate of the visible geometry. At a higher level, PVS gives a set of potentially visible regions in a game map form the current region of the client. PVS can be efficiently precomputed and stored from the BSP trees [138] [139]. We exploit the this precomputed PVS data intelligently to determine the game state without any additional processing overhead. This approach works well for games with small, closed a highly occluded maps (eg. Maps of Quake III, Unreal engine based games). We also describe the possibility of generalising this approach by catering for
maps with huge open areas where visible area is too large (eg. Maps of Ryzom, WoW).

We augment the approach with a distance based scheme to minimise the visible range in huge open areas. Rest of this section describes this approach in detail.

5.4.1 Visibility and Spatial Subdivision Scheme

Naturally, if 2 players do not see each other, they probably wont have any interaction. Thus, visibility is an aspect that needs to be considered. The visibility here means whether there is any object intersects a line that spawned from two players position. This information can be naively computed by shooting rays in all directions from a players position at run-time. But the overhead for this method is too high. Most of the rays wont hit anyone in FPS game. Hence we do Spatial Subdivision of the map into convex hulls (that is, regions or clusters) as described in Section 5.1 and compute visibility. Then from the location (convex hull) of player, the criticality is determined using PVS of areas is the ideal tool we can use in our algorithm. Advantages of using 3D Spatial Subdivision and PVS over naive 2D Grid based Subdivisions.

1. *It is fast.* Since PVS is pre-computed, the querying time is constant. It doesnt incur any overhead at run-time.

2. *It can be stored efficiently.* PVS is map-related. The information is highly crafted to the internal structure of in-map buildings. Thus, the memory required to store the information is optimized. Only the necessary space needs to have visibility information which can be stored in the same BSP tree of the game map.
3. *It is conservative.* As discussed before, in certain situation, even though two players cant see each other, they may still be potentially visible to each other. Thus, a prediction based on potential visibility can have less error.

However, there is a shortcoming for this scheme as well. That is, in a big cluster or region, for example, outdoor battle field, every point inside the field will be marked visible to other points as the field is a single area by itself. Thus, if we use PVS alone the player will have very little chance to save power in outdoor environment. Hence, we augment our approach with a distance based scheme to cut-off the range in widely open areas up to some discernible distance. To get the best estimation of the distance, we use *path distance* as described below instead of simple *Euclidian distance*.

By combining PVS with distance based scheme our system can efficiently save energy for all kind of game maps (indoor maps with lots of occlusions and open outdoor maps with few occlusions). Hence, it can be extended to many game genres.

5.4.2 Two-Level Scan Algorithms

The algorithm is divided into two parts, namely *macro scanning* and *micro scanning*. Macro scanning mainly serves to compensate the limitation of PVS scheme. It also helps to pre-filter irrelevant players, thus keeping the size for micro scanning within a scalable level. Then, Micro scanning queries the PVS for the area where the player is and sleep the client for a period of time if no other player is potentially visible.

The pseudo code presented in Algorithm 4 outlines the main algorithm.
Algorithm 4 Main (3D Renderer’s View Based Approach)

\begin{verbatim}
for each server frame do
    for each active client do
        if client is sleeping then
            continue
        else
            do $t = Macro\_scan()$
            if $t > 0$ then
                sleep for $t$ msec
            else
                do $t = Micro\_scan()$
                if $t > 0$ then
                    sleep for $t$ msec
                end if
            end if
        end if
    end for
end for
\end{verbatim}

5.4.2.1 Macro Scanning Algorithm

Macro scanning is basically a distance based scan. It uses vision range or maximum perceivable distance (MPD) as a ruler to scan any other player inside range of current player. The MPD is measured in an open game map. It is the distance beyond which the player cannot see other players or cannot affect other player in any way. For simplicity we just consider view distance and ignore weapon distance. If the game has defined weapon ranges, that information can be exploited.

In the macro scanning, we have two objectives. One is to check if there is any player inside MPD. If nobody is inside the MPD, we can sleep for a while based on the distance of next player or portal, whichever is nearer to MPD. Second objective is to store the list of players who are inside the MPD, so later in micro scanning, this list of players can be scanned through instead of scanning every players. This
Breath First Search (BFS) is used to traverse clusters. We use path-distance to measure the distance from the cluster of the player to other clusters as shown in the Figure 5.11, so only the necessary clusters are added to the queue to scan. Note that if the cluster radius is bigger than MPD, then the MPD line is effectively a straight line distance. Detailed pseudo code for macro scanning is presented in Algorithm 5.

The distance from a player to the current player’s MPD can be easily calculated from the positions of the players. Sleep time is then calculated by dividing distance by players movement speed as shown in Equation (5.4.1). However, we cannot simply get the move speed of one player only. The time calculation should used relative speed of two players. Thus, player orientation and moving direction needs to be
Algorithm 5 Macro_scan

clusterQueue: initialize to contain the current player’s \((player_i)\) cluster
playerQueue: initialize to empty, will pass to micro scan
sleeptime: initialize to maximum sleeptime+1

\[
\textbf{while} \text{ clusterQueue is not empty do} \\
\text{cluster} = \text{pop clusterQueue} \\
\text{for each } player_j \text{ in cluster; } j \neq i \text{ do} \\
\quad \text{if } player_j\text{’s distance to } player_i > \text{MPD then} \\
\quad \quad \text{calculate time } player_j \text{ takes to MPD of } player_i \text{ using MovementSpeed} \\
\quad \quad \text{if time < sleeptime then} \\
\quad \quad \quad \text{sleeptime} = \text{time} \\
\quad \text{end if} \\
\quad \text{else} \\
\quad \quad \text{put } player_j \text{ in playerQueue} \\
\quad \text{end if} \\
\text{end for} \\
\text{for each portal}_k \text{ of cluster that has path-distance to } player_i < \text{MPD do} \\
\quad \text{clusterOther} = portal_k - > \text{otherCluster} \\
\quad \text{add clusterOther to clusterQueue} \\
\text{end for} \\
\text{for each portal}_q \text{ of cluster that has path-distance to } player_i > \text{MPD do} \\
\quad \text{calculate the time taken from the portal}_q \text{ to MPD} \\
\quad \text{if time < sleeptime then} \\
\quad \quad \text{sleeptime} = \text{time} \\
\quad \text{end if} \\
\text{end for} \\
\text{end while} \\
\text{return sleeptime}
\]

considered. Thus computing for every player at every frame would incur too much overhead. Moreover, its not necessary to have exact relative speed to obtain a good sleep time for this approach. Our approach is adaptive and just needs an estimated average movement speed (here after we simply refer it as move speed). If move speed is lower than actual move speed of the players, there are more possibility for errors. This estimated move speed will be dynamically tweaked based on error rate, which we describe in the evaluation section 5.8.6.1.
$$Sleep\ Time = \frac{MPD_{distance}}{2 \times MS}$$

where, $MPD_{distance}$ - is distance from a player to MPD of the current player; $MS$ - is estimated move speed of players.

If the number of players is large (as in MMOG games), incremental scanning such as dual ring algorithm can be applied as described in Section 5.2.3 to improve scalability.

5.4.2.2 Micro Scanning Algorithm

As our spatial subdivision scheme is 3D and stored in the same BSP tree of the game map, Micro scanning becomes simple and efficient. We just check against the PVS for each player in the player queue that created by Macro Scan. The PVS is pre-computed data from BSP tree. Its structure is in the form of bit-array. For example, in Quake III, Area $x$ is visible to area $y$ if $(1 << y \mod 8)$ bit of $PVS[x \times \text{no. of areas} + y / 8]$ is set. Note:- $<<$ is bitwise left shift operation. Pseudo code for micro scanning is described in Algorithm 6.

**Algorithm 6 Micro_scan**

playerQueue: passed as parameter
currentPlayer: index no. of current player

```plaintext
for each player inside playerQueue do
    if $(1 << \text{player} \mod 8) \& PVS[\text{currentPlayer} \times \text{no. of areas} + \text{player} / 8] == 1$ then
        return NO_SLEEP
    end if
end for
return fixed_sleep_time
```

182
This approach is implemented in Quake III. Quake Engine has provided us with the PVS data. However, for other games we generate and store the PVS data in the BSP tree of the game map. In a similar format as in Quake III. As described earlier more details about the PVS can be found in [138,139]. For games which already have PVS data, our system become much thinner by simply exploits this data. For micro scanning, the time to sleep will always be the Fixed Sleep Time (FST) when there is no other players in current player’s PVS. If FST is too high, there are more possibility for errors. The FST value is dynamically will be dynamically tweaked based on error rate, which we describe in the evaluation section 5.8.6.1.

5.5 Wireless Interface Control at Client Side

The client side logic is simple as most of the decisions are made at the server side in all the above approaches. The server sends \textit{sleep} command with \textit{duration}. We call this duration as \textit{Potential Sleep Duration (PSD)}. On receiving sleep message the client side RC computes the Effective Sleep Duration (ESD) as given below and puts the wireless interface into sleep mode. As described in [111], there are two constraints for sleep duration: \textit{maximum sleep duration} that a game can tolerate (using techniques such as Dead Reckoning) and \textit{minimum sleep duration} that is really required to save energy (due to \textit{mode switch power cost} and \textit{mode switch latency} of the wireless interface). Our measurements on the average current consumption in ON and SLEEP states for various wireless interfaces along with the corresponding mode switch penalties are presented in Table 5.2.

\textit{Computing Effective Sleep Duration (ESD)}

183
if (PSD < minSleepDuration)
    ESD = 0;
else if (PSD > maxSleepDuration)
    ESD = maxSleepDuration;
else
    ESD = PSD;

Table 5.2. Power Characteristics of Different Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Current (mA)</th>
<th>Mode Switch Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ON</td>
<td>SLEEP</td>
</tr>
<tr>
<td>old 802.11b</td>
<td>600</td>
<td>10</td>
</tr>
<tr>
<td>new 802.11b</td>
<td>300</td>
<td>6</td>
</tr>
<tr>
<td>802.11g</td>
<td>260</td>
<td>4</td>
</tr>
<tr>
<td>802.11g*</td>
<td>260</td>
<td>4</td>
</tr>
<tr>
<td>3.5G(HSPA)</td>
<td>201</td>
<td>8</td>
</tr>
<tr>
<td>ZigBee</td>
<td>50</td>
<td>2.5</td>
</tr>
</tbody>
</table>

* - using atheros chipset; Current - Current Consumed

5.6 Algorithm Selection

We have described three approaches for wireless interface power management. Each of these approaches have their pros and cons. They perform better in one type of game and game map and may fail to save significant power in another type of game map. The following table (Table 5.3) gives summary of algorithms and the maps where they perform well. The table uses the following definitions.

The three approaches Distance Based Approach, Visibility Based Approach, Renderer’s View Based Approach are referred with their abbreviations DBA, VBA and RBA respectively. DBA has Single Ring and Dual Ring algorithms (both algorithms have micro and macro power management modes), we refer them as DBA-SR and DBA-DR. RBA has micro and macro scanning in which macro scanning is optional. Macro scanning is basically used to filter the area for micro scanning. However, for
small maps we don’t need macro scanning, hence we have another class RBA-Mi, which is RBA with only micro scanning.

**Open, Closed and Mixed map.** We define Open Map as the map which consists of only a few loosely distributed buildings and most of the area in map is flat planes. Example of an open map is the Simpsons map of Quake III. Closed Map is defined as the map that is mainly constructed by architectural structures, such as the interiors of a castle. It consists of a lot of rooms separated by door-ways. These are highly occluded maps. Example of a closed map is Q3DM7. A Mixed Map is a combination of both. Usually huge Open maps with some Closed areas inside. For example, huge flat area with some cities (with several architectural structures) at spare distances.

**Small and Huge map.** A map of size around ten or less than ten miles as Small Map; they are common in FPS games. A map which spans several cities (several hundred miles) as Huge Map; they are common in MMOG games.

**Few and Large number of Players.** Games like Quake allows a maximum of 16 players in a map. We define anything below 100 players as Few Players. When there are several hundreds and thousands of players we refer it as Large number of Player.

## 5.7 Implementation

### 5.7.1 Network Characteristics of FPS and MMOG Games

In all these games the game’s logic is executed completely by the server with only the server having complete knowledge of the entire game world — individual clients
Table 5.3. Selection of Algorithms

<table>
<thead>
<tr>
<th>Map Type</th>
<th>Number of Players (n)</th>
<th>Common Game Genre</th>
<th>Algorithm option</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closed, Small</td>
<td>Few</td>
<td>FPS</td>
<td>VBA, RBA-Mi</td>
<td>Simple, Faster. VBA is also Conservative (low errors)</td>
</tr>
<tr>
<td>Closed, Small</td>
<td>Large</td>
<td>FPS</td>
<td>RBA, VBA-Mi</td>
<td>Simple, Faster</td>
</tr>
<tr>
<td>Closed, Huge</td>
<td>Few</td>
<td>FPS</td>
<td>VBA</td>
<td>Simple, Faster, Conservative (low errors)</td>
</tr>
<tr>
<td>Closed, Huge</td>
<td>Large</td>
<td>FPS</td>
<td>RBA</td>
<td>Scalable due to Macro scan</td>
</tr>
<tr>
<td>Open, Small</td>
<td>Few</td>
<td>FPS</td>
<td>RBA</td>
<td>Macro scan helps to save energy</td>
</tr>
<tr>
<td>Open, Small</td>
<td>Large</td>
<td>MMOG-FPS, MMOG</td>
<td>RBA</td>
<td>Macro scan helps to save energy</td>
</tr>
<tr>
<td>Open, Huge</td>
<td>Few</td>
<td>FPS, MMOG</td>
<td>DBA-SR</td>
<td>Simple, Accurate Sleep Time</td>
</tr>
<tr>
<td>Open, Huge</td>
<td>Large</td>
<td>MMOG</td>
<td>DBA-DR</td>
<td>Scalable, Simple</td>
</tr>
<tr>
<td>Mixed, Small</td>
<td>Few</td>
<td>FPS</td>
<td>RBA</td>
<td>Macro(Distance) and Micro(Visibility)</td>
</tr>
<tr>
<td>Mixed, Small</td>
<td>Large</td>
<td>MMOG-FPS</td>
<td>RBA</td>
<td>Macro(Distance) and Micro(Visibility), Macro scan makes it Scalable</td>
</tr>
<tr>
<td>Mixed, Huge</td>
<td>Few</td>
<td>MMOG-FPS, MMOG</td>
<td>DBA-SR, RBA</td>
<td>DBA-SR is Simple, gives Accurate Sleep Time. RBA caters both areas</td>
</tr>
<tr>
<td>Mixed, Huge</td>
<td>Large</td>
<td>MMOG</td>
<td>DBA-DR, RBA</td>
<td>Both are scalable, DBA-DR is Simple. RBA caters both areas.</td>
</tr>
</tbody>
</table>
only know about the small portion of the world that they can currently see. This information difference not only reduces the server to client bandwidth traffic but also serves as an effective anti-cheating mechanism. While the client may perform some state estimation using techniques such as Dead Reckoning and Interpolation to handle small amounts of expected network jitter, it needs to constantly synchronise its game state with the server through regular client to/from server update packets.

On average, the traffic rate from the server to clients is 16Kbps (≈20 packet/sec) and 21Kbps (≈40 packet/sec) in the reverse direction for FPS games; and 15Kbps (≈15 packet/sec) and 16Kbps (≈20 packet/sec) in the reverse direction. The key information sent from a client to the server is the player’s current position and action (e.g. shooting). As game state consistency is server maintained, the key consequence of sleeping a client’s network interface is that the player’s position updates could be delayed significantly. As discussed previously, we consider this delay to be a problem only if it results in state inconsistency between two or more players. Hence, the decision for the client to go to sleep is server initiated and only if the server believes that it will not result in important state inconsistency.

5.7.2 Sleep Command

We have implemented our algorithms in the server side of all these games as the server knows the entire state information about the client. We have used Ubuntu 10.10 with Eclipse as our development platform. The server makes the estimates the client’s state and the time period that the client can sleep without affecting the quality of game play adversely.
In our implementation, we added a new sleep command on top of the existing client/server communication in the game. The server uses this sleep command to tell specific clients how long to sleep. Upon receiving this command, the client will send a signal to the wireless hardware to make it sleep for the specified time period.

To turn the wireless card into sleep mode and wake up fast we used Madwifi driver [149], which is an the advanced Linux based WiFi driver for WiFi cards with Atheros chipset. MadWifi comes with Atheros Hardware Access Layer (HAL). The hal provides hardware support for wireless network adapters. This code is part of the atheros driver but configured separately to allow fine-grained control over the set of chips supported. Moreover, only cards based on Atheros chipset AR5210, AR5211, AR5212, AR5213, AR2413, AR2417, AR2425, AR5413, AR5416, AR5418, AR5424, AR9160, and AR9280 chips (and companion RF/baseband parts) support HAL future. For more details on this can be obtained through the ubuntu man pages [150].

5.8 Evaluation Methodology

We have used the same testbed (Figure 3.13) described in previous chapter with the setup shown in (Figure 5.12) to measure the power consumption by wireless interface.

For the experiment the game clients used the NEC PA-WL/54AG PCMCIA wireless interface card (Figure 5.14) for connectivity, which is based on Atheros chipset (AR5212) and supports Hardware Access Layer (HAL) [150]. The power characteristics of the the card are shown in Table 5.4. NEC PA-WL/54AG PCMCIA hardware
Figure 5.12. Setup for Wireless Interface Power Measurement

is optimized for low standby mode power consumption. As shown in the Table 5.4, in power save mode, the card consumes only 28 mA compared to 248 mA when it is in receive or transmit mode. Moreover using the card, turning on the network interface takes only 10-50ms for its computer to connect to the wireless network. The PCM-CIA card was connected to the Laptop, through a PCMCIA extender card, built by Accurite Technologies (Figure 5.13), that extended the PCMCIA bus and provided leads for current measurements. The measurements for wireless interface was done in both Quake III and Ryzom games.

Table 5.4. Power characteristic of the card used

<table>
<thead>
<tr>
<th>Mode</th>
<th>NEC PA-WL/54AG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standby Mode</td>
<td>4.66 mA</td>
</tr>
<tr>
<td>Power Save Mode</td>
<td>28 mA</td>
</tr>
<tr>
<td>Receive Mode</td>
<td>248 mA</td>
</tr>
<tr>
<td>Transmit Mode</td>
<td>248 mA</td>
</tr>
</tbody>
</table>
Figure 5.13. PCMCIA Extender card (Accurite Technologies)

Figure 5.14. PCMCIA NEC PA-WL/54AG WiFi card used in the Experiments
5.8.1 Evaluation Objectives

The primary objective of our evaluation is to find potential power savings achievable by our algorithms and the effects of our algorithms on game play quality for different games, different maps and with different algorithm parameters. We performed three kinds of evaluation, namely: trace-based simulations, actual system performance measurements, and user studies.

Our evaluation aimed to answer the following questions:

1. **Base Effectiveness**: What is the potential power savings achievable by our algorithms in optimal hardware conditions (no penalty)? We present these results in Sections 5.9.2 and 5.9.3.

2. **Player Density**: What is the effect of player density on power saving? We present these results in Sections 5.9.1, 5.9.2 and 5.9.3 for our three different approaches.

3. **Repeatable & Identical Game Runs**: Game state varies from run-to-run and no two games runs are identical. However, in order to compare different algorithms in our visibility based approach (such as, static, dynamic), we need to evaluate different algorithms under the “same” condition. How can this be done? We used simulations as described in Section 5.8.5 and the results are given in 5.9.2.

4. **Players’s perception of Quality**: How do we evaluate quality objectively and how do we relate the objective metric used to a player’s perception of the
5. **Simulation vs Real Power**: How much power can we save in a realistic game playing environment with minimum impact on gameplay? We did real power measurements using the Testbed in Section 3.6.1 and the results are shown in Section 5.9.2.7 and 5.9.1.

6. **Impact of Measured Errors**: How much of the error is actually visible to the user? We answer this with the results of user study in Section 5.9.2.9.

7. **Effect of Game Type**: How effective is our system in saving power for different game types? We evaluated and presented results of our algorithms for FPS (Sections 5.9.2 and 5.9.3) and MMOG (Section 5.9.1). In all cases we can save significant amount of energy.

The general error metric we used for objective assessment in all our

### 5.8.2 Defining a Quality & Power Metric

**Definition of Error**

We defined a miss or an error as the case when a client, on waking up from sleep, finds that there is,

- some other client in its *vision range* or *vision radius* for distance based approach,

- some other client who is *directly visible* from its position for visibility based approach and
• some other client in its *Potentially Visible Set (PVS)* and *Maximum Perceptible Distance (MPD)* for renderer’s view based approach.

**Sources of Errors**

There are 4 scenarios that an error can happen:

1. other players teleport to near the current player (instantly appear). [Applicable to all three approaches.]

2. other player runs fast toward MPD or Vision Range during sleep (move speed or velocity prediction fail). [Applicable to all three approaches.]

3. the boundary area in PVS test is too small and other player is just near to that area and moving towards current player (fixed sleep time is too long). [Applicable only to renderer’s view based approach.]

4. visible edges of the grid are identified as invisible as only the mid-points of the grids are checked for visibility (the grid size is large) [Applicable only to visibility based approach.]

Errors should be reduced as it degrades a user’s game play experience. There is a trade-off between choosing longer and shorter sleep time. As discussed above, if we put the network interface into sleep mode for longer period, there will be more chances errors to occur. However, if we always choose a shorter sleep time, there will be much overhead. This is because, according to our experiments, we found that the wireless card always incurs a switching-delay to switch from ‘sleep’ to ‘wake-up’ mode. This switching-delay is about 50 to 100ms, and it cannot be eliminated, though
with a better card, the delay may be reduced. Higher number of short sleeps, results in higher number of switching, which in turn introduces additional delay affecting the game quality and amount of energy saved.

The errors were determined by continuously logging the player-to-player visibility, distance, view angle, every player’s PVS, sleep time and sleep duration during runtime.

The error rate and quality of the game are thus computed as,

\[
ErrorRate = \frac{\text{Total sleep time with errors}}{\text{Total sleep time}} \tag{5.8.1}
\]

\[
Accuracy = 1 - \frac{\text{Total sleep time with errors}}{\text{Total sleep time}} \tag{5.8.2}
\]

This accuracy metric has the advantage that it can be easily measured from the logged data. However, there exists the problem of how well does this accuracy metric map to a human player’s perceived game play experience? Since we log (and hence check) a player’s position only once every time quantum (of say 50ms), an error does not automatically imply a bad player experience as the two players might have just been visible to each other for much less than the time quantum. In addition, since network games are designed to tolerate some amount of network delay and jitter, very high accuracy may not be necessary. We evaluated the impact of accuracy levels on human quality perception through a small scale user study (described in Section 5.8.3).

As described in Table 5.2, different wireless devices have different penalties for turning it ON and OFF. The penalties are due to mode switch power cost and mode


switch latency. Hence, to have an absolute (best case value) power saving numbers, we have used the following equation (Equation 5.8.3):

\[
\% \text{ Power Saved} = \frac{\text{total sleep period}}{\text{total game period}}
\]  

(5.8.3)

This percentage can be converted to actual power numbers by multiplying it with the actual power consumption of the specific wireless interface (subtract the penalties for accurate values). We have also done actual power measurements for specific wireless card (Atheros chipset based NEC PA-WL/54AG PCMCIA) to show these additional penalties on power saving. To measure the actual power saving, we logged the current passing through wireless card during the game play and used the following formula (Equation 5.8.4) to compute power saved.

\[
\text{Energy Saved} = 1 - \frac{\text{Average Current Observed}}{\text{Average Operating Current}}
\]  

(5.8.4)

Determining Maximum Perceptible Distance

Maximum Perceptible Distance (MPD) is measured by printing out the distance of two players contineously and then, taking the distance when current player is at a position that can hardly see the other player. Figure 5.15 illustrates this perceptible distance in the map Simpsons_q3.pk3. We found 2500 game units (game unit is game dependent) is a good MPD for Quake III.

5.8.3 Small Scale User Study - Methodology

We did a small scale user study with 10 graduate students from our lab to evaluate one of our power management approaches (Visibility Based Approach). We used
Quake III game in Ethernet LAN environment for this study. We modified the game server and client to stop sending packets to each other during sleep periods. This isolated users from the effect of variable WiFi interface wake up times — allowing them to focus solely on the effects of our algorithm. The objective of this study is to find the effects of power saving algorithm on game play quality.

The study procedure was as follows:

First, the participants filled up a basic demographic information in the survey form, attached in Appendix D. Then, they played the original unmodified version of Quake III to get an idea about the default quality of the game in terms of any network delay and jitter in LAN environment for 10 to 30 minutes. For those who were not familiar with Quake III, we trained them to play the in game. After that,
they were presented six power optimised versions of the Quake III game. The power optimised versions had our power management algorithm running with different $\alpha$ values. The users played each version for about 10mins in random order without knowing $\alpha$ values and given their observation on the quality of the game play with respect to network related artefacts by filling up the questions in the survey form.

5.8.4 Experiments

For each of our experiments, we ran the game client and played for around five minutes to capture the date. Players controlled their characters to move around the virtual world, found and attacked monsters/enemies in the game. We vary the maps, number of players (2, 4, 8, 16...) and networks (WiFi, 3G) in each of these experiments. We used up to 4 human players for the rest we added computer controlled bots as required. For Ryzom and Planeshit we used the default maps, they are huge open maps. For Quake III we used two maps developed by Quake III Arena team ($q3dm1$, $q3dm7$, Simpsons), and two developed by us ($longroom$, $bigroom2$). $q3dm1$ and $q3dm7$ are comparatively small maps, while $longroom$ is roughly twice their size, and $bigroom2$ is twice as big as longroom. $q3dm1$ and $q3dm7$, have different kinds of obstacles and 2 big areas for the players to move around in. $longroom$ and $bigroom2$, on the other hand, have more uniform obstacles (walls), and the obstacles partition the game area into roughly similar sized rooms. Simpsons is a open map with less occlusions.

All the power saving values presented in the Section 5.9, are based on estimation using the Equation 5.8.3 with the logged game play data, except those presented in
Sections 5.9.2.7 and 5.9.1 which are real power measurements which includes all the penalties.

**Characteristics of Different Wireless Interfaces**

However, to accurately simulate real-world network latencies and jitter, we collected our game traces using two different wireless interfaces that are commonly found on modern smartphones — 1) 802.11n WiFi, and 2) two different 3.5G cellular HSPDA interfaces (2Mbps and 7.2Mbps).

Our experimental testbed to collect these network traces is shown in Figure 3.13. We used laptops to run the Quake III Arena game client as we did not want the limited graphics capabilities of cell phones to hinder our network-interface focused experiments. Our latencies to the game server were as follows: 3.55ms average RTT (stdev. of 3.58ms) for WiFi with a 0% packet loss, 84.35ms average RTT (stdev. of 34.63ms) for a 2Mbps 3.5G connection with a 4% packet loss, and 62.29ms average RTT (stdev. of 8.91ms) for a 7.2Mbps 3.5G connection with a 1% packet loss.

### 5.8.5 Additional Evaluations - Visibility Based Approach

As we had multiple variations of the algorithm (eg. static sleep time, dynamic sleep time) in our visibility based approach, to compare them fairly we used traces of the game play for some experiments, which are described below.

#### 5.8.5.1 Using Traces for Repeatability

To address the first question, we separated the trace collection from the algorithm evaluation. In the trace collection phase, we did not run any power conservation mechanisms. Instead, the game was played “normally” and the player location,
player-to-player visibility information (a record of which players are visible to each player at that point in time), and direction information was logged every 100ms. We then compared different lookahead mechanisms using the same collected traces. This allowed us to have a consistent and systematic way of comparing different algorithms.

5.8.5.2 Running the Simulations

We now describe how we conducted repeatable experiments using a custom-built two-stage Java processor. The first stage takes the visibility grid (described in Section 5.3) for the particular map being used and the player traces and generates a predicted visibility map for various $\beta$ values. In our experiments we used $\beta$ values of 200, 400, and 600ms. We collected different traces for WiFi and 3.5G environments and did not mix interfaces (i.e., in WiFi traces, all clients were using WiFi).

In the second stage, we analyse the predicted visibility map and determine the expected power savings and accuracy for different algorithm variants for that particular trace.

5.8.6 Additional Evaluations - Renderer’s View Based Approach

As discussed above, there are sources of error in all the three approaches. However, the renderer’s view based approach allows us to control the error rate during run-time according to a user defined quality threshold, acceptable error threshold, by tweaking other variables which contribute to the error. By tweaking these variables we can achieve optimal power saving under the given quality threshold. It provides a better automated way to trade-off between quality and power saving. Due to time limit, we used simple Additive Increase and Multiplicative Decrease (AIMD) based controller
and defer using most robust controllers like, Proportional-Integral-Derivative (PID) controller as future work.

5.8.6.1 Using AMID for Error Control

It is clear now that we need to balance the error rate and the total power saved (alternatively, total sleep time). The error rate always grows together with total power saved. The level of growth really depends on how much user can bear with the affected game experience. Hence, the reasonable way is to have multiple levels of error threshold controllable by the user. The user can choose or request the server (to avoid cheating) for the one that is most appropriate to his situation. In such a way, we can have a fixed error threshold when running the algorithm. Next, we need to tweak two variables Fixed Sleep Time (FST) 5.4.2.1 and Move Speed (MS) 5.4.2.2 to optimal level, in order to save maximum energy for a given error rate threshold. Our AIMD scheme adapted from TCPs congestion control algorithm to control the complex variables. It enables fast convergence of error rate to expected threshold and automatic adjustment of FST and move speed.

As the game activities are much random and human behaviours in games are hard to predict, such feedback based control systems are much robust and adaptive to different requirements. Though statistical methods (eg. n-gram algorithm) [151] can be used to predict MS of human-controlled character, their accuracy levels are less than 70%. Hence, cannot be used to control the network interface. However, these statistical algorithms are widely used to augment the intelligence of AI-controlled characters to mount a more challenging opposition as higher level of accuracy is not
requied in such cases. Higher level of accuracy will make AI-controlled characters unbeatable and do not give any sporting chance to the human player.

The procedure we want to take in determining sleep time is presented in Algorithm 7. The flowchart in Figure 5.16 summarises this feedback based procedure for controlling error rate.

**Algorithm 7 Run-Time Error Controller**

Let fixed sleep time initialize to 200ms, move speed initialize to 500 game unit/s. (Game unit is game dependent). Move speed here is the normal average movement speed in game. 500 units/s is obtained from multiple game play experiments.

Let user choose an Error Threshold (ET) between 1%, 3% and 5%. Higher threshold results in lower quality and higher power saving. It determines the trade-off between quality and power.

Run the middleware algorithms (macro and micro scanning).

for each server frame do

Compute running-average value of error rate for last 'N' frames.

if Average error is less than ET then

Increment FST by $X_1$ and
Decrement move speed by $X_2$

▷ Note: This is additive

else

Decrement FST by $1/Y$ of its value and
Increment move speed by $1/Y$ of its value

▷ Note: This is multiplicative

end if

end for

Note: After several set of experiments with Quake III game we have arrived at the values 2, 2 and 4 for the game dependent variables $X_1$, $X_2$ and $Y$ respectively. These values gave reasonably fast convergence and well controlled error level. One can guess that as we reduce the errors faster (multiplicative), the algorithm gives high priority
for maintaining the required quality than the aggressiveness in power saving (which is additive). This is a required behaviour, as it is better to shutdown than trying to play at unexpected quality levels.

**Calculation of Running Average Error:** The error control algorithm needs calculation of average error for a specific time interval. We assume players perception of game quality will be based on past one minute of game play; we defer the user study on this to future work. Thus, to compute running error, we only concern sleep errors for the past one minute. For the worst case, if the delay between sleep is 100ms, the client sleeps 100ms each time, there could be 300 sleeps within one minute. Thus, we keep a 300-sized queue of sleep information. Sleep information simply contains the duration of the sleep and an error flag. Every time the client sleeps, we push

---

**Figure 5.16.** Error Control Loop with AMID
the latest sleep information to the queue. If the queue is full, we pop the oldest sleep information. The Equation (5.8.5) defines the Running Error or Average Error Rate (AER) function used in Section 5.8.6.1. AER function is the same as the error function defined in Section 5.8.2, except that now it computes only for past one minute.

\[ AER = \frac{Error_{\text{sleep}}}{Total_{\text{sleep}}} \]  \hspace{1cm} (5.8.5)

where, \( Error_{\text{sleep}} \) is sum of erroneous sleep time in Queue and \( Total_{\text{sleep}} \) is sum of total sleep time in Queue.

5.9 Evaluation Results

5.9.1 Distance Based Approach

As described earlier, we have implemented distance-based approach in Ryzom game. Both macro and micro power management methods are implemented. For macro we have used Dual Ring Algorithm as it is highly scalable which is a necessary requirement for MMOGs. As it is a simple and straightforward approach we briefly show two important aspects of this approach here. First, we show how the algorithm behaves in sparse and dense environments, where sparse environment and dense environment are related to player density (number of players in a given space). When there are roughly more than four players in same area the environment is considered dense. The size of the area is map dependent. Second, we show the contributions from macro and micro level power management of the algorithm.
All the measurements in this section are real measurements using NEC PA-WL/54AG card in our Testbed 3.6.1.

5.9.1.1 Sparse Environment - Low player Density

Figure 5.17 shows the energy saved in sparse environment. When the predicted movement speed of players is 600 units/ms (maximum speed), we can save up to 15% consuming power on the network device and the error rate for it is nearly 0%. The error occurs if during the time players turn off their network interface, there are entities entering their vision region. When the estimated movement speed is 200 units/ms, the energy saving is up to 22%, however the error rate increases to 10% which is quite high. For 400 units/ms estimated movement speed, the error rate is only 2%, which is acceptable, and we can save up to 17% energy.
5.9.1.2 Dense Environment - High Player Density

In dense environment, such as more opponents and monster areas, figure 5.18 represents the energy saving using our algorithm. At estimated movement speed 400 units/ms, our solution saves 15% energy consumed by the network interface.

5.9.1.3 Contribution from Micro and Macro Power Management

In general, figure 5.19 indicates the contribution from distance (macro) and angle (micro) based algorithms. With low predicted movement speed, the contribution of distance based algorithm increases since there is higher chance there are not any entities which are coming near a player.

5.9.2 Visibility Based Approach

As described earlier, we have some variances of the visibility based algorithm. The simple algorithm takes static sleep time, whereas dynamic algorithm automatically
selects one of sleep durations from 200ms, 400ms and 600ms. We describe the results with all variations in this section.

All the power saving values presented in the Section 5.9, are based on estimation using the Equation 5.8.3 with the logged game play data, except those presented in Sections 5.9.2.7.

5.9.2.1 Baseline - No Prediction

We first provide baseline power consumption values, using trace-based evaluation, for the range of scenarios we considered. In our baseline, the client goes to sleep, for fixed sleep intervals of 200ms, 400ms, and 600ms, only when no player is visible (no prediction is performed). Since only the player’s current location is considered, this scenario can be thought of as the most aggressive case where the power saved is expected to be the largest and the accuracy the lowest. The simulation results are shown in Figure 5.20 for the q3dm1 map.
Figure 5.20. Results with no Prediction (q3dm1 Map)
A few observations can be made. First, substantial amounts of power (up to 50%) can be saved when the number of players is small and the sleep interval is large. However, this results in low accuracy value (<70%) when the number of players increases. We show, in the rest of this section, that our dynamic lookahead algorithm (described in Section 5.3.1) can achieve large power savings with low accuracy loss.

5.9.2.2 Experiments over Various Networks

In this section, we show how our dynamic lookahead algorithm balances the accuracy and power savings tradeoff. We compared our dynamic algorithm with 3 static lookahead algorithms that always sleep for fixed lengths of 200ms, 400ms, and 600ms (when sleeping is possible) respectively. In order to study the effect of each algorithm’s aggressiveness, we vary the parameter $\alpha$ (Section 5.3.2) from 0 to 1. We performed the measurements using both the WiFi and 3.5G network traces.
Figure 5.21 shows the results for 3.5G cellular networks for the map $q3dm1$ using 4 players. “Dynamic” refers to our dynamic algorithm where the maximum possible sleep interval is selected. Each line shows a smooth variation of $\alpha$ from 0 (best quality and the leftmost point) to 1 (worst quality and the rightmost point).

Using the most conservative settings ($\alpha = 0$) where a player only sleeps when the system believes that no other player will be visible within the chosen sleep time interval, the power saved is 14%, 11%, 9% and 7% for dynamic, static 200ms, static 400ms and static 600ms respectively. The accuracy for all 4 cases is above 94%. Note: when $\alpha = 1$, the algorithm converges to the same no-prediction algorithm described above and thus the results are identical to those presented earlier.

The results show that by varying $\alpha$, different trade-offs between accuracy and power saved is possible. In addition, the static 200ms algorithm, due to the short sleep
interval, cannot achieve large power savings. For almost all $\alpha$ values, the dynamic approach provides the highest power savings with the highest accuracy.

The results for WiFi, shown in Figure 5.22 (for the same map and number of players as 3.5G), are similar to the 3.5G measurements and shows that our approach can work over networks with fairly different delay and jitter (3.5G has larger jitter than WiFi) characteristics.

We obtained similar patterns for both interfaces when using the other two maps and player numbers (2, 8, and 16). We omit those results due to space constraints.

5.9.2.3 Effect of Velocity

As described in Section 5.3.1, we used the player’s velocity to predict the chance of seeing the other players. We conducted an experiment to determine how much this extra information helps us to save power, or maintain accuracy. In Figure 5.23, we plot the dynamic mode power savings for 3 modes — no velocity (uniform weights), correct velocity, and wrong velocity, in which we reverse the direction in our calculations. We observe that in all 3 cases $\alpha = 0$ and 1 do not show any improvement, since by definition they do not care about the velocity. However, we observe that using velocity helps everyone else. Overall, using velocity allows us to save more power with better accuracy.
Figure 5.24. Effect of Different sleep intervals

Figure 5.23. Effect of Velocity
5.9.2.4 Effect of Different Sleep/Wakeup Intervals

In this section, we study the effect of using different minimum wakeup and sleeping intervals. Our current choice for these values was determined by actual WiFi interface power measurements.

To test the effect of these values, we reduced the sleeping intervals to 100ms, 200ms, and 300ms (from the default values of 200ms, 400ms, and 600ms) and the minimum sleep interval from 100ms to 50ms. These values were chosen to maintain the same ratios (of 2, 4, and 6) between the minimum sleep and wakeup intervals as our default values. Figure 5.24 shows the results. As expected, compared to the default values, using shorter sleep intervals (100, 200, and 300ms) with a shorter minimum wakeup time (50ms) results in significantly better power savings, with similarly high accuracies, as the algorithm is able to make finer granularity sleep decisions. Hence, using network interfaces that allow shorter minimum wakeup times, such as ZigBee, can result in improved power savings.

However, if we reduce the ratio between the minimum sleep and wakeup intervals, we experience a performance loss. This is shown by the middle line in Figure 5.24. That line shows the effect of using sleep intervals of 100, 200, and 300ms with a minimum wakeup time of 100ms. This results in a sleep-to-wake-up ratio of 1, 2, and 3 compared with 2, 4, and 6 previously. This lower ratio impacts our power savings as the network interface is now required to stay awake much longer relative to the time it can sleep. We plan to investigate the effect of other ratios in the future.
5.9.2.5 Effect of player density

Since our algorithm only sleeps the network interface when the player has no-one in their current and predicted future AoV, it is obvious that player density in the map plays a crucial role in determining how well the algorithm performs. We investigate this effect by running different traces created with different number of players playing on different sized maps. We then compute a player density value as follows:

\[
\% \text{ Player Density} = \frac{\text{no. of Players}}{\text{Map's Visibility Grid Area}}
\]  

(5.9.1)

Figure 5.25 shows our results. To make the plot more readable, we normalised the player density by setting the lowest density to 1. Hence, the scenario with the highest density has about 14 times more players per unit area than the scenario with the lowest density.
Figure 5.26. Dynamic Versus Static Algorithms

The 5 curves represent 5 different $\alpha$ values. While it is clear that a more aggressive setting (larger $\alpha$) always saves more power (at the expense of accuracy as shown earlier) and vice versa, there are clear regions where the power saved percentage falls into different effectiveness zones. For example, player densities between 2 to 3 can save up to 42% of the interface power while densities between 4 to 7 can save at most 25% of the interface power. In addition, this Figure re-iterates that while the absolute power saved percentages can vary greatly over the range of player densities, the relative ordering of the savings achievable with different $\alpha$ values remains the same (lower $\alpha$ values save less power than higher values).

5.9.2.6 Benefit of Our Dynamic Algorithm

Figure 5.26 plots the power savings achievable by our dynamic algorithm and the 3 static algorithms for the full range of player densities. $\alpha$ is set to 0 (best accuracy) for all four algorithms and player densities are normalised as mentioned previously.
We see that at various player density points, the best static sleep algorithm keeps changing (the lines for the static algorithms cross at various points). However, the dynamic algorithm performs consistently the best over the entire range (≈50% savings at the lowest density). This result clearly shows that our dynamic algorithm achieves performance that cannot be achieved using a single sleep interval over a variety of maps and player numbers.

5.9.2.7 Real Power Measurements

In the trace based simulation, the power savings are estimated by computing the fraction of total game time in which the interface is put to sleep. In this section, we evaluate the correctness and accuracy of this approximation by comparing it with actual power measurements.
Using the setup described in Section 3.6.1, we measured the actual power savings as well as the savings as predicted by our simulation. Figure 5.27 shows the results. A few observations can be made.

First, while the actual power savings are less than the predicted savings, the trend or behaviour is very similar. One reason why the actual power savings is lower than that predicted by simulation is that in simulation, power is considered to be saved over the entire sleep interval. However, in practice, the network interface takes some time to turn on/off and that consumes energy.

To quantify this effect, we measured how long the network interface was sleeping in our simulation results versus how long it actually slept in our real measurements. We found that in the real measurements, the network interface was asleep for only 72% of the time that it was asleep in the simulation. If we scale the simulation’s predicted power savings by this factor of 0.72, as shown in Figure 5.27, the simulation results mostly agree with the real measurement. This constant scaling factor helps to prove validity of the simulation as the power saving values obtained from it can be converted to actual values by applying a constant scaling factor of 0.72. Note: this scaling factor is specific to the WiFi interface used in our real measurements and will have to be recomputed for other interfaces. Based on this observation we will improve our simulator in future to input scaling factor as a hardware specific parameter.

5.9.2.8 Another Perspective of Power Savings

In order to put our algorithm’s power savings potential in perspective, we converted the power savings into improvements in overall battery lifetimes. To do this, we played Quake III Arena on a HTC Desire HD smartphone, for five minutes, and
determined the percentage of battery power drained. This allowed us to calculate how long the game could be played until the battery completely drained.

With this information, we were able to calculate that our algorithm will extend the battery life by about 7.5%, using a network interface power savings of 21% on average (the real measurements show a 15% to 27% power savings for medium player density) and the fact that network overall consumes 35% of entire device power consumption. This translates to about 15 minutes of extra battery life for the HTC Desire HD smartphone (when it is used solely to play network games) or about 36 hours of additional standby time.

![Graph showing Quality Loss Versus Alpha](image)

**Figure 5.28.** User Study - Quality Loss Versus Alpha

### 5.9.2.9 Impact of Errors on Perceived Quality

Finally, we performed a user study with 12 players to evaluate how the accuracy values defined by our metric affects a human player's perceived game play experience. The participants in our study were all non-author Masters and PhD students from
the same lab. We first trained the players on an unmodified game, and then had them play the modified game several times with different $\alpha$ values. For each game play, we asked them to rate how noticeable, if at all, were any network related artefacts in the game, compared to the unmodified version, on a 5-point Likert scale. The players were not aware of the $\alpha$ value used during their game play and the order in which they played the games was randomised.

Figure 5.28 shows the user study results. We excluded two players’ results that had outlier ratings (always good or contradictory). The results show that most users found the system very playable. The average rating reduces slowly with increasing $\alpha$ values, indicating that the errors, while noticeable, were not highly damaging to the gameplay experience.

**Summary** The previous section shows that our dynamic AoV look-a-head algorithm is quite effective at saving power in fast paced FPS games — achieving up to 50% power savings, in our trace-based simulation, with almost no quality loss ($> 95\%$ quality) in low player density situations. However, for high density situations it needs additional filtering mechanisms such as view angle or distance.

### 5.9.3 3D Renderer’s View Based Approach

We measured the power and convergence rate for various map types, player density and error thresholds in Quake III game.

All the power saving values presented in the Section 5.9, are based on estimation using the Equation 5.8.3.
5.9.3.1 Effects of Map Type

Our first experiment is to measure the effects of map type on energy saving. We fix all the variables except map type. The settings for the experiment are shown in the Table 5.5.

**Table 5.5. Effects of Map Type - Experiment Variable Setup**

<table>
<thead>
<tr>
<th></th>
<th>Open Map</th>
<th>Closed Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map:</td>
<td>Simpsons</td>
<td>Q3DM7</td>
</tr>
<tr>
<td>Player no.:</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Error Threshold</td>
<td>3%</td>
<td>3%</td>
</tr>
<tr>
<td>MPD:</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>Fixed Sleep Time</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Move Speed:</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Game Length:</td>
<td>15 minutes</td>
<td>15 minutes</td>
</tr>
</tbody>
</table>

The graph in Figure 5.29 shows the test results. The numbers, 0.303 and 0.344 are the total energy saved. Internal counters are set up at server to record the contributions of Macro Scan and Micro Scan. Overall, Closed Map saves 4% more energy than Open Map. This shows our algorithm can function well on both big map and small map. In Open Map, most of the energy saving comes from Macro Scan while for Closed Map, most of the energy saving comes from Micro Scan. This is expected, because PVS doesn’t work much on an Open Map as there are very little geometric occlusions. Distance between players in an Open Map is very much likely to be greater than MPD, thus Macro Scan can save considerable amount of time. But in a Closed Map, most of time players are within MPD, thus, it not possible for Macro Scan to return a valid sleep time. This experiment shows the importance of using both Macro Scan and Micro Scan to achieve maximum energy saving.
5.9.3.2 Effects of Energy Threshold

From previous experiment we can observe that the amount of energy saved for different map types is roughly equal. However, there is a significant difference in contribution from Macro Scan and Micro Scan algorithms in saving energy. As they are equally important in energy saving, in the following experiments we only concern about the total energy saved. We keep one of the variables (map type) constant in order to reduce the complexity of analysis. We choose to use the Closed Map for the remaining experiments. So with other parameters unchanged, we have two variables to study, namely No. of players, Error Threshold (ET).

In this experiment we want to measure the effects of different error thresholds on the energy saving. Thus, we fix the No. of players to 8, and run the experiments with ET of 1%, 3% and 5%. The result in Figure 5.30 shows the percentage saved for each level of ET. The general trend is that the higher the Error Threshold, the

\[\text{Energy Saved} \]

Figure 5.29. MapType vs. Energy Saved
more Energy can be saved. From ET 1% to 3%, there is a 10.34% increase in the amount of energy saved. From ET 3% to 5%, there is a 21.82% increase. The non-linearity of increase is explained by the unpredicted fluctuation of game-play. The rationale behind the increase is that the higher of ET, the higher of Fixed Sleep Time, hence the more client can sleep for each duration resulting in more sleep times. This experiment just proves this.

5.9.3.3 Effects of Player Density (number of Players)

Next we want to measure the effects of No. of players on energy saving. This time, we fix ET to 3%, and change the No. of players to 2, 4 and 8.

As shown in Figure 5.31, fewer players in a map result in more energy saving. For the 2-player game, the players are in fact alone for most of the time. Intuitively, the wireless interface should sleep up to 80% of time given that the map Q3DM7.pk3 is quite big. However, only 57.43% of energy is saved. This is due to mode switching
overhead. As mentioned earlier, there is about 100 milliseconds lag between each sleeps, thus this amount of time is not possible to save. Moreover, as the card turns on/off too often, the heating of the hardware causes some slower response. This can be improved with better wifi cards.

5.9.3.4 Effects Error Controller on Optimising Algorithm Parameters

The Error Controller is supposed to continuously adjust the Fixed Sleep Time (FST) and Movement Speed (MS) to optimal values to save maximum energy while meeting the quality requirement (Error Rate). The following graph in Figure 5.32 shows how the FST is adjusted to optimal values overtime when planing with 8 players and 5% error threshold. (This data is from the same game play used to plot Figure 5.31 for 8 players). Again, from the same game play data, we have also plotted the graph in Figure 5.33 which shows the corresponding Average Error Rate (AER) over the time.
Figure 5.32. Fixed Sleep Time of Micro Scan varies over Time

Figure 5.33. Error Convergence Over the Game Play Time for ET=5%
From the Figure 5.32 we can see, the FST increases linearly for the first 20 samples. Then it jumps down from about 550ms to about 300ms. This is because the AER at this instant is higher than the threshold 5%; the FST needs to be shortened in order to keep the AER within the threshold. We shorten FST by multiplying its value by a factor of 0.75. This Additive-Increase-Multiplicative-Decrease (AIMD) method of TCPs congestion control algorithm allows AER to be quickly reduced. The flat portion of the graph means the AER is still higher than threshold. At this moment, FST stops increasing. Whenever another error occurs, the FST is reduced again until the AER is lower than threshold. The graph in Figure 5.33 shows the effect of this adjustment. The AER starts from 0. It then increases until slightly higher than 5%, where FST is forced to be multiplicatively reduced. The AER is well controlled within the 5% threshold which proves our AIMD method to be effective.

5.9.3.5 Effects Error Controller on Average Error Rate

The following graphs in Figure 5.34 and Figure 5.35 show the convergence of AER for games with 3% and 1% threshold respectively. The average convergence time is less than 4 seconds. This can be shortened further by optimising the sample size and using weighted average for AER.

Summary The amount of energy saved is satisfactory. With 8 players we are still able to save 17% of energy. With 2 players we can save up to 57%. Our experiments conform to theoretical analysis. The energy-save is determined by three factors namely Map Type, No. of Players and Error Threshold. Larger map doesn't affect power saves much which in turn proves that our algorithms scalability. Macro
Figure 5.34. Error Convergence Over the Game Play Time for ET=3%

Figure 5.35. Error Convergence Over the Game Play Time for ET=1%
scan indeed functions well on this part. Our AIMD algorithm also effectively controls the Errors thus ensuring consistent quality of game.

5.10 Summary of all Results

We have shown the results of all three algorithms. In general all approaches save significant amount of energy (17% to 57% energy of the wireless interface). Previous dead-reckoning based approaches [27] [31] cannot save significant amount of energy if multiple objects are controlled by extrapolation (state prediction). The methods described in these previous works can save around 36% of wireless interface energy when there are less than two objects controlled by dead-reckoning. In our approaches we can put the wireless interface into sleep mode even when the dead-reckoning threshold is met and our approaches are not affected by number of objects controlled by dead-reckoning. Our experiments also show that very short sleeping intervals obtained by dead-reckoning based algorithms are not practically feasible due to the mode switch latency. It is also interesting to note that in games with highly occluded maps our visibility based approaches can save more energy (up to 57%) than huge open maps (about 17%). This is because occluded maps provide more chances for sleeping due to occlusions. In open maps, due to long default vision range, there is a high probability for atleast one opponent to appear in the player’s vision range and the chances for sleeping is low. Our three algorithms have their own merits and demerits. For a given game genre we can select the appropriate algorithm which gives highest benefit based on the criteria suggested in Section 5.6.