Evaluating dead reckoning variations with a multi-player game simulator

Wladimir Palant
University of Oslo
palant@ifi.uio.no

Carsten Griwodz
University of Oslo
griff@ifi.uio.no

Pål Halvorsen
University of Oslo
paalh@ifi.uio.no

ABSTRACT

One of the most difficult tasks when creating an online multi-player game is to provide the players with a consistent view of the virtual world despite the network delays. Most current games use prediction algorithms to achieve this, but usually it does not go beyond applying the DIS [2] dead reckoning algorithm proposed in the mid-90s. In this paper we introduce a simulator called GLS that allows us to evaluate different aspects of DIS and its variations. We examine the impact of prediction and clock synchronization on game consistency. We also evaluate the convergence algorithm we introduce here. Furthermore we look into ways for compensating increasing delays to keep the player's view of the game state sufficiently consistent with other players.

1. INTRODUCTION

Multi-player online games become increasingly popular nowadays. It is simply more challenging to compete against other players than it is to compete against the artificial intelligence the game developers build into their games. One major problem developers of online games have to face is the network latency that makes fast user interactions difficult. Especially first-person shooting games have high demands on accuracy when displaying remote players. Shooting games require fast reactions from the players, and the players require accurate position of their opponents without noticeable delay from the game in return.

Early network games like Doom solved the problem easily – they would send changes in game state (position updates) at frame rate, at least 25 times per second. This way whenever a client had to redraw the screen it already received all the necessary position updates. This was an incredible waste of bandwidth, of course, something that was only possible for a small number of players on a local area network.

The US Army chose a different approach for its SIMNET project in the mid-80s. It should allow large-scale training of army units in a networked simulator. This effort was standardized as Distributed Interactive Simulation (DIS, [2]) in 1993. Here a technique called dead reckoning was used to limit the number of necessary position updates – the simulator would predict the current position of a remote object between position updates based on its velocity and acceleration. The formula for the position at which DIS will show a remote object is well-known (last term is optional):

$$p = p_0 + v_0\Delta t + \frac{1}{2}a_0\Delta t^2$$

Here p_0, v_0 and a_0 are the position, velocity and acceleration of the remote object received with the position update. Δt is the time that passed since the position update was sent (difference between current time and the timestamp of the position update). In a similar way the orientation of the remote object can be calculated, with the same assumption that it did not change its angular velocity/acceleration (or at least not too much).

DIS does not require position updates to be sent too often. It also deals with the fact that due to network lag a position update is already outdated when it reaches its destination – this is compensated by prediction. But DIS goes even further and defines a way to decide when a position update must be sent. Every player will predict his own position based on the last position he sent. If this position deviates too much (in terms of distance) from his real position he should send a position update. The interval between two subsequent position updates should not be longer than a certain time interval however (typically a few seconds). This restriction ensures that if one position update was lost on the network another one will follow shortly, so that the impact of the loss is limited.

Surprisingly, today the approach proposed by DIS is still used in most multi-player online games with only small variations. Only a few other prediction algorithms have been proposed, typically with a very limited area of application. In this paper we evaluate DIS and its variations in the scenario of a typical first-person shooting game. We present different evaluation criteria and show how the results differ from one to another. We furthermore introduce an extension to DIS that smoothes the displayed trajectory of the players and makes it more realistic.

2. RELATED WORK

Delay and jitter on the network have an impact on the user's experience of the game, they also influence the player's performance. While this paper only intends to measure the effects on computer players, there has been work on providing hard numbers for the impact on human players [3, 4, 5].

NOSSDAV '06 Newport, Rhode Island USA

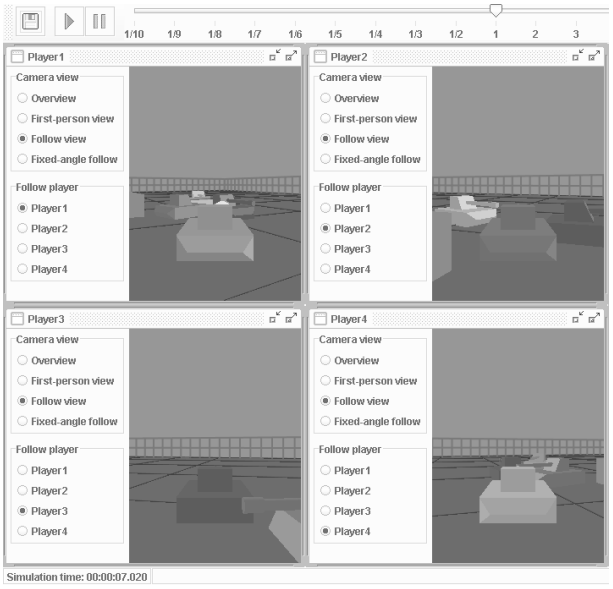

Figure 1: Simulator's graphical user interface

Furthermore [6] shows on the example of the online game MiMaze, how the game experience can suffer if network latency is not accounted for. And [7] demonstrates that this is not an abstract problem. In shooting games players from different continents come together and compete with each other, result being that pairwise network delays between players can be very high.

While we chose convergence to repair prediction mistakes (changing the state smoothly so that a correct state is restored) other approaches are possible. One such approach is Timewarp, removing the inconsistencies by restoring a previous game state [8]. Another technique, which we describe later in the paper, is position-history based dead reckoning (PHBDR, [9]).

3. SIMULATION SCENARIO

In order to run extensive tests on effects of network latency we developed a system called GLS (Game Latency Simulator, source code and the pre-compiled simulator are available on the Internet [1]). For this we chose to emulate an open source game called BZFlag [10]. This is a first-person shooter game with a very simple goal – drive a tank and shoot as many other tanks as possible. This made it easy to create good computer players and to evaluate results.

The simulator emulates several players, all with their own view of what happens in the game (the current results are based on a two-players scenario). The players send position updates to each other through the network. Network topology is not simulated, it is simply assumed that everybody receives a certain position update after some preset network delay. Therefore it is irrelevant whether the game uses a peer-to-peer approach or whether position updates go through a central server – the important thing is only that everybody concerned receives the update. For the results presented here we used constant network delay, simulating jitter and loss is also possible however. We assume that our clients always send correct data, cheating is not considered yet.

We created a GUI for the simulator (Figure 1) that allows us to see the views of all players at the same time. It is useful when we need to understand the numbers that we get from our experiments. The simulator is mostly run in batch mode however where we can speed up the processing at 1000 times the real-time speed while still processing every frame for every simulated client. The configuration file defines all the necessary parameters, and each of those can be overridden from the command line, for example the game field size, maximum velocity, number of obstacles and the simulation duration:

```
java batch.Main --field=Field(40,40)
                --maxLinearVelocity=1.5
                --obstacles=3
                --event1=ShutdownEvent(180000)
```

To sum up the advantages of our simulator over using an existing game:

- Simulations can be run much faster than at real-time speed
- Using a simplified world model where actions don't have unexpected side effects
- Algorithms can be swapped easily, even from the command line
- Can be run in batch mode with all important parameters specified on command line or in configuration file
- Network is simulated as well so that its characteristics can be adjusted easily

3.1 Physical model

The simulated world is modeled after BZFlag. Unlike flight simulators or racing games that have to simulate the real world properly, most first-person shooting games use a simplified physical model. The world of BZFlag is two-dimensional, the tanks cannot fly (they sometimes can jump however which we left out in our simulator). Orientation, movement direction and firing direction are always identical. Acceleration is meaningless, a player can change his linear and angular velocity instantaneously. Both linear and angular velocity are limited however (2 m/sec and $\frac{\pi}{2}$ sec^{-1} are our default settings). Negative linear velocity is allowed, the tanks can move backwards.

The physical model of popular games like Doom or Quake deviates only in a few details from BZFlag, so results produced with our simulator can be applied to those games as well. The instantaneous velocity changes have an effect on the players' strategies, the good players have to be fast and unpredictable. In these games it does not make sense to move slower than at maximum speed (and in fact in Doom and Quake you cannot) or not to change direction for more than a few seconds. If you do you are likely to get shot. That observation went into the design of computer players, who reconsider their current strategy every two seconds. It also imposes a limit on prediction quality.

3.2 Clock synchronization

One advantage of running the game in the simulator is that all clients automatically have synchronized clocks –

there cannot be disagreements on the current time as all clients are simulated on the same computer. We can always tell whether some event on two different clients happened simultaneously. That is also the reason we can compare the displayed position of a player with his real position – because we know where this player is *right now*.

This is something we cannot simply assume in a real-world scenario. Different clients run on different computers. Clocks on different computers usually differ by at least several seconds, so that the timestamp included in a position update will only tell us the time interval between this position update and the previous one from the same client. It is possible to synchronize the clocks on two computers with protocols like Network Time Protocol however, that is why our simulator will still assume synchronized clocks.

If clock synchronization is not available, the timestamp from the position update has to be ignored (it is meaningless). The best a game can do is to use the time when it received the position update instead. This option is also implemented in GLS and is used for one of the evaluations presented.

3.3 Handling of remote players

Whenever a player needs the position of another player, a prediction module will have to calculate current position of the remote player based on received position updates. The prediction module that we mainly use implements a modified version of the DIS dead reckoning algorithm. While it predicts the player's position as specified in the DIS standard, pre-reckoning [11] has also been implemented, meaning that position updates should not only be sent when the player's position deviates too much from his predicted position, but also when its orientation deviates from the predicted orientation by more than $\frac{\pi}{6}$.

We also implemented a convergence algorithm to correct the displayed position of remote players smoothly. When a new position update is received the predicted position of a player typically changes and does not match his displayed position any more. DIS only has the option to jump to the new position immediately (no convergence) or in a series of smaller jumps ("smoothing"). Our goal on the other hand was to produce something that would look like a genuine movement. Whenever the remote player needs to be displayed we use his previous displayed position but adjust it according to a linear and angular velocity that are calculated as follows:

- Calculate the remote player's predicted position, the target point
- Set α as the angle between the current displayed orientation of the player and the direction from his current displayed position to the target point
- If $maxAngularVelocity$ allows us to reach the desired orientation in this update step – set angular velocity accordingly. Otherwise the angular velocity should be $\text{sign}(\alpha) \cdot maxAngularVelocity$. That means that we always turn with the maximal possible velocity, which is usually the case in the real game as well.
- If the target point can be reached in this update step – set linear velocity accordingly. Otherwise use $\cos(\alpha) \cdot maxLinearVelocity$ for linear velocity.

3.4 Handling of remote shots

In GLS, whenever information about a shot is received the shooter will be displayed as firing. The shot is originating at the *displayed* position of the shooter and propagating linearly, with the predefined constant shot velocity (5 m/sec by default). Now the displayed position of a remote player is not necessarily correct, furthermore the target receives information about the shot with a delay. This has as consequence that the shooter and his target do not necessarily agree on whether the target was hit.

Several approaches have been proposed to ensure that there is always an agreement on the result of a shot. One representative suggestion is targeting [12]. The idea is simple – the information on the shot arrives delayed, so the target should compensate this by moving the first displayed position of the shot forward accordingly, so that the shot's position is the same in the view of both players. The disadvantage of this approach is that the target will not have time to react, especially in a fast game dominated by close combat. It also solves only one part of the problem as the shooter might be aiming based on wrong information about the target's location. We decided not to adopt targeting or any similar approach, keeping player's view consistent (realistic) is more important to us than a consistent global game state.

3.5 Evaluation methods

We used four different evaluation criteria for the results presented here:

- Average position deviation: this test simply compares the displayed position of a player in the view of another to his real location. This is done at frame rate (every 20 ms), the result is the average distance between the real and the displayed position – an objective criterion of the quality of displayed positions.
- Number of shots: this measures how many shots the computer players fire per second. A low number of shots indicates that the players have trouble catching each other, for example if the convergence algorithm is disabled and the opponent seemingly jumps from one place to another all the time. Typically this is a situation where real players get irritated and do not have much fun playing the game.
- Number of misses: here we measure how often the computer players miss their target. A high number of misses indicates that errors in displayed positions of remote players affect the "judgement" of the computer players making them miss more often than usual.
- Agreed hits: for this test we do not rely on computer player's aiming skills and make them fire constantly instead (one shot every 400 ms). We count the percentage of hits where both parties (shooter and target) agreed that the shot did in fact hit. This is a measure of game state consistency, low percentage of agreed hits indicates that different players often disagree on the game score.

4. EVALUATION RESULTS

In this section we present selected results from our large-scale tests with the GLS simulator.

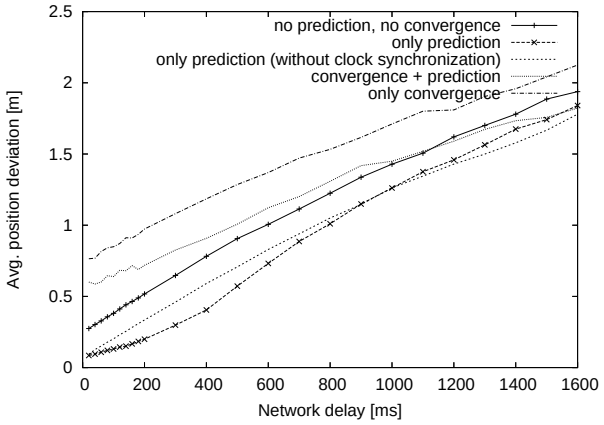

Figure 2: Position deviation for different scenarios

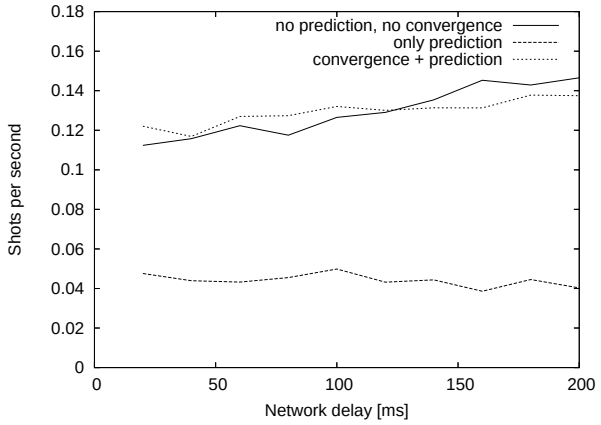

Figure 3: Shot numbers for different scenarios

4.1 Prediction and convergence

We examined the effect that prediction and convergence have in a fast-action first-person shooting game. As the player can change his direction quickly it is to be expected that prediction will not be very accurate (see section 3.1). With growing network delay the inaccuracy of the prediction will grow because we will have to predict over a larger time interval. This is best measured as average position deviation, see Figure 2 (every dot represents the average for 20 minutes of simulation, standard deviation isn't shown to keep it readable). We can see that while prediction gives us more accurate positions for low delays this advantage becomes smaller with higher network latency.

This figure also shows that the convergence algorithm comes with a price – you get less accurate absolute positions, of course, if you move smoothly to your new predicted position instead of jumping there. This price is highest for low delays, for higher delays we notice an advantage of convergence – it does not allow wrong predictions to influence the positioning of the player too much. If the prediction produces a position several meters away from the current displayed position of the player, it will take several seconds for the convergence algorithm to get there, and by that time a more accurate position update will probably arrive already. We can see that in our testing scenario the results with and without convergence become the same for 1500 ms one-way network delay.

There is also another way of dealing with bad predictions. We can drop time synchronization (see section 3.2) and ignore the timestamp of the position updates. Instead we use the timestamp at which the position update was received and predict the player's position starting from this time point. That means that we knowingly show player positions that are outdated. These positions are reasonably accurate however as we only need to predict for the time interval between two position updates. As can be seen in Figure 2 this also eliminates the rapid growth of position errors towards higher network delays.

It does not make sense to drop prediction and use only convergence instead. As Figure 2 shows, converging only between known positions of remote players produces clearly worse results than the combination of prediction and convergence algorithm.

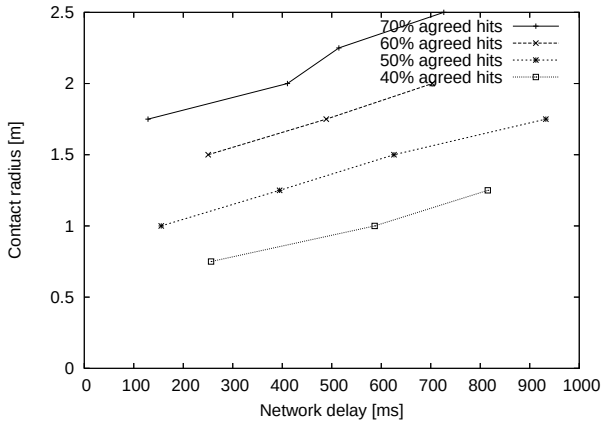

Figure 4: Contact radius required for constant number of agreed hits

Now one could get the impression that using prediction without convergence still gives you the best results for up to 900 ms network lag and therefore should be used unless the network delay really exceeds this number (which will usually be not the case). Figure 3 however shows another aspect of having no convergence algorithm – the opponent seemingly jumps every time a new position update is received, making it difficult to aim and shoot at him (in this figure and all the following every dot represents the average of 20 simulations with 10 minutes simulated time each). This results in a much lower number of shots (see section 3.5) compared to the scenario where convergence is used or neither convergence nor prediction are used and the tank simply stands still until the next position update is received.

4.2 Adjusting contact radius

Usually when first-person shooting games check whether a shot was a hit or not, they ignore the complex geometry of the player as he is displayed on screen. Instead they define the player's "center" and every shot that came within a certain distance ("contact radius") of this "center" counts as a hit. The contact radius might be changed for various purposes, for example in some games computer players

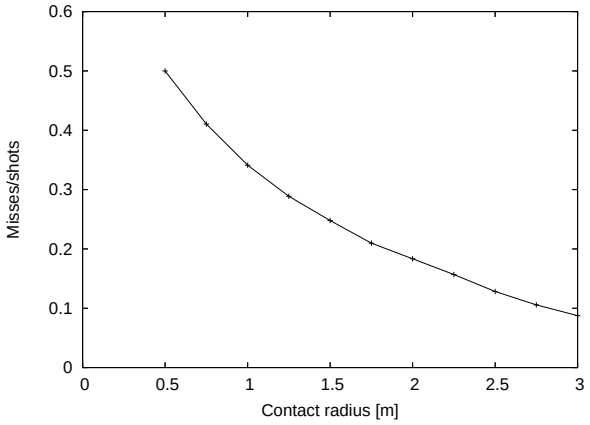

Figure 5: Relation between number of misses and contact radius

have a bigger contact radius then human players, thus giving human players an advantage (Doom is an example). Here we wanted to examine whether it is possible to adjust the contact radius to limit the effects of network lag on game consistency and user experience.

For the following evaluations both prediction and convergence were enabled. Our measurements of agreed hits (see section 3.5) confirmed that we get a higher number of disagreements on whether a shot hit his target or not when we increase the network delay. So we measured the percentage of agreed hits for different contact radiuses to decide how we have to choose the player's contact radius depending on the network delay to keep the percentage of agreed hits constant (see Figure 4).

Surprisingly it shows up that there is a linear relation between the network delay and the contact radius required. The figure seems to imply that for our data sample the dependency between contact radius and the number of agreed hits is also linear – 10% increase in agreed hits correspond to roughly 0.4 m increase in contact radius. This is not the case, of course, as it would mean 100% agreed hits for 3 m contact radius (the real number from our experiments is 89%).

Now we look at the results from the user's (in our case: computer player's) point of view. Interestingly enough, the percentage of shots that missed their target hardly changes with increasing network delay. It seems that the computer player's ability to aim is nearly unaffected by the network delay when the convergence algorithm is used. Also, as expected the number of misses decreases with increasing contact radius, this relation is shown in Figure 5.

4.3 Dependence of prediction algorithms on physical model: PHBDR

We also tried to run tests with our implementation of the PHBDR [9] algorithm. We found out however that this algorithms does not work with a simple physical model that does not consider acceleration.

PHBDR only uses the player's previous known positions to predict his current position. It takes either the two or three last position updates and calculates a first/second order curve though these points – the supposed movement trajectory of the player. While first order curves are relatively unproblematic, the second order curve defines an acceleration that the player must have to hit all three positions at the correct time. In our simulator this acceleration term was often extremely high and, as the result, the prediction based on it absolutely unrealistic.

5. CONCLUSION

Our experiment results show that prediction the way it is defined by DIS is still the best way to achieve accurate positions. However, this approach degrades with higher network delays which can be prevented either by using our convergence algorithm or by dropping clock synchronization. We furthermore show that while our convergence algorithm comes with a cost in terms of position deviation (which becomes less important towards higher network delays) it prevents much user irritation through sudden "jumps" of remote players. Overall it seems that the advantages of providing a reasonably smooth transition when repairing game state outweighs the disadvantages, even though the convergence algorithm we present here still can be improved. A convergence algorithm should especially always be used for games where the precise position is not so important like strategy and role-playing games.

We could also show that increasing the contact radius of the players with increasing lag can be used to keep the consistency level in a game on a constant level. The dependency between the contact radius and the network delay has been shown to be linear in our simulator and it is reasonable to assume that it is also linear in other shooting games. This means that it is enough to adjust the contact radius without changing the player's displayed size. Other finding was that the network delay has almost no influence on the player's aiming capabilities when our convergence algorithm is in effect.

On the negative side we had to state that PHBDR cannot be applied to our physical model. This algorithm was originally meant for flight simulator and assumes a physical model where velocity changes go with an acceleration – which is not true for most first-person shooting games.

6. FUTURE WORK

We intend to continue testing different scenarios and approaches with GLS. One particular area of interest are position updates containing the relative position to the receiver of the update. Players usually make their decisions based on relative positions and not on the absolute ones, so using relative positions in the prediction algorithm should improve the game experience e.g. by allowing more precise aiming. Also predicted relative positions for players interacting with each other should be more reliable than predictions for absolute positions.

Another interesting research area are the dependencies between prediction quality and the physical model with its parameters. We already did a basic evaluation of the effects a reduced angular velocity limit would have and this work should continue.

We also intend to run tests with more complicated network models that include not only delay but also jitter and losses. A measurement of required network bandwidth depending on the prediction and convergence algorithm used is also necessary.

Furthermore we need to extend our convergence algorithm to deal with obstacles. The tank should probably drive around an obstacle or if this would take too long (meaning that the current displayed position of the player is very far off) it could jump to a position on the other side of the obstacle.

It should be interesting to do some evaluation for scenarios with more than two players. While our simulator allows us to do so we only run a few very basic tests so far. And, of course, we should confirm the relevant results with human players. Unfortunately these are far less effective than computer players which is inconvenient for large-scale tests.

Something that we might want to look into is cheating prevention. The best algorithms are not worth much if they allow players to cheat easily, thus disturbing the game experience for everybody.

7. REFERENCES

[1] GLS – Game Latency Simulator. `http://www.ifi.uio.no/forskning/grupper/nd/projects/2004/mismoss/gls.html`

[2] IEEE Standard for Distributed Interactive Simulation – Application Protocols. *IEEE Std 1278.1-1995*

[3] Tristan Henderson. The effects of relative delay in networked games. *PhD thesis, University of London*, London. April 2003.

[4] Lothar Pantel, Lars Wolf. On the Impact of Delay on Real-Time Multiplayer Games. *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Miami Beach. May 12-14, 2002.

[5] Mark Claypool, Kajal Claypool, Feissal Dama. The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games. *Proceedings of the 13th Annual Multimedia Computing and Networking*, San Jose. January 18-19, 2006.

[6] Laurent Gautier, Christophe Diot. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. *IEEE Multimedia Systems Conference*, Austin. June 28 - July 1, 1998.

[7] Wu-chang Feng, Francis Chang, Wu-chi Feng, Jonathan Walpole. A Traffic Characterization of Popular On-line Games. *IEEE/ACM Transactions on Networking*, vol. 13, no. 3, June 2005.

[8] Martin Mauve. How to Keep a Dead Man from Shooting. *Proceedings of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, Enschede. October 17-20, 2000.

[9] Sandeep K. Singhal, David R. Cheriton. Using a Position History-Based Protocol for Distributed Object Visualization. *Technical Report STAN-CS-TR-94-1505*, Department of Computer Science, Stanford University. February 1994.

[10] BZFlag – a free multiplayer multiplatform 3D tank battle game. `http://bzflag.org/`

[11] Thomas P. Duncan, Denis Gračanin. Pre-Reckoning Algorithm for Distributed Virtual Environments. *Proceedings of the 35th conference on Winter simulation*, New Orleans. December 07-10, 2003.

[12] Chris Haag. Targeting, a variation of dead reckoning. `http://www.gamedev.net/reference/articles/article1370.asp`